ABB Robotics

Application manual
RobotWare Machine Tending

Power and productivity
for a better world™

Trace back information:
Workspace Main version a58
Checked in 2014-02-19
Skribenta version 4.0.378

Application manual
RobotWare Machine Tending
RobotWare 5.60

Document ID: 3HAC044398-001
Revision: A

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

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

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

Additional copies of this manual may be obtained from ABB.

The original language for this publication is English. Any other languages that are supplied have been translated from English.

ABB AB
Robotics Products
SE-721 68 Västerås

Table of contents

This page is intentionally left blank

Overview of this manual

About this manual

This manual explains when and how the option RobotWare Machine Tending (referred to hereinafter as **RWMT**) may be used.

Usage

In this manual, you can look up how to use the option RobotWare Machine Tending. Furthermore, you will get detailed information on the syntax of the RAPID instructions and functions and the parameters.

Who should read this manual?

This manual is primarily intended for experienced programmers.

Prerequisites

The reader should be well versed in

- industrial robots and their basic terminology
- the RAPID programming language
- the system parameters and their configuration.

References

References	Document ID
Technical reference manual - RAPID overview	*3HAC16580-1*
Technical reference manual - RAPID Instructions, Functions and Data types	*3HAC16581-1*
Operating manual - IRC5 with FlexPendant	*3HAC16590-1*
Technical reference manual - System parameters	*3HAC17076-1*
Application manual - Motion functions and events	*3HAC18152-1*
Operating manual - RobotStudio	*3HAC032104-001*
Operating manual - RobotWare Machine Tending	*3HAC044397-001*
Operating manual - Machine Tending PowerPac	*3HAC044396-001*
Application manual - FlexPendant SDK	*3HAC036958-001*

Revisions

Revision	Description
-	First version, RobotWare 5.15.
A	RobotWare 5.60. New features are added.

License agreement

License agreement for RobotWare Machine Tending

1 ABB is the only owner of the copyright and usage rights in the software option RobotWare Machine Tending that is delivered.

2 ABB assigns to the licensee a simple, non-transferable, exclusive, but unlimited right to use the option RobotWare Machine Tending.

3 The license entitles the user only to the **proper use** of the software option RobotWare Machine Tending on a robot controller. The licensee is not allowed to replicate the option RobotWare Machine Tending or parts of it and make these accessible to third parties or the use the software or parts of it on other robot controls. Taking a back-up copy exclusively for own use on the original hardware is exempted from this.

4 Modifying, translating, reverse engineering or decompiling or disassembling the software option RobotWare Machine Tending is not allowed.

Product documentation, IRC5

Categories for manipulator documentation

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

All documents listed can be ordered from ABB on a DVD. The documents listed are valid for IRC5 manipulator systems.

Product manuals

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

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

Technical reference manuals

The technical reference manuals describe reference information for robotics products.

- *Technical reference manual - Lubrication in gearboxes*: Description of types and volumes of lubrication for the manipulator gearboxes.
- *Technical reference manual - RAPID overview*: An overview of the RAPID programming language.
- *Technical reference manual - RAPID Instructions, Functions and Data types*: Description and syntax for all RAPID instructions, functions, and data types.
- *Technical reference manual - RAPID kernel*: A formal description of the RAPID programming language.
- *Technical reference manual - System parameters*: Description of system parameters and configuration workflows.

Application manuals

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

Continues on next page

An application manual generally contains information about:

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

Operating manuals

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

The group of manuals includes (among others):

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

Safety

Safety of personnel

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

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

Safety regulations

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

This page is intentionally left blank

1 What is RobotWare MachineTending?

Usage

RobotWare Machine Tending (RWMT) is a software option that makes it easier to access the robot and the system peripherals in the handling applications, both for the system operator as well as for the integrator.

For easy operations, a user interface (GUI) is provided on the FlexPendant of the IRC5-control.

For the Integrator, a set of RAPID data types, instructions and functions have been provided so that the RWMT can be integrated with the application program.

With the help of a process configuration, RWMT can for instance also be modified with respect to the graphic views as well as with respect to the existing signal interfaces.

This page is intentionally left blank

2 System prerequisites

Hardware

Robot controllers for the generation IRC5 with Flex Pendant generation 2, 3, or higher.

Software

The following software options are required in each case for configuring respectively using the RWMT and must be considered while ordering the robot or obtained separately from ABB:

- ABB Robots with IRC5 controllers and operating system RW5.60.
- RobotStudio (See *Operating Manual - RobotStudio* listed in the section *References on page 11*.)
- Software option **[1167-1] RW MachineTending**

The following software option is required if interface signals or RAPID variables have to be set to a specific value when operation mode changes, as is necessary, for instance, in the case of the EUROMAP-interface in injection molding:

- Software option **[623-1] Multitasking**

This page is intentionally left blank

3 Installation

3.1 Setup

Installation on the PC

The following procedure is needed for installing the necessary data for RWMT on a PC:

- Make sure that the pre-requisite RobotStudio 5.60 or upper is installed already on the PC.
- Install RWMT by executing its setup program.

The setup program (setup.exe) is shipped with the robot controller. The necessary data for RWMT can be installed by executing the setup program on the PC.

The setup program adds the Additional Option RWMT to the RobotStudio mediapool. The mediapool is located per default under *C:\Program Files\ABB Industrial IT\Robotics IT\Mediapool*

3.2 System generation

Creating a robot system

With the help of RobotStudio, a robot system can be created with the option RobotWare Machine Tending. The key for the controller module must contain the options that are described in the chapter *System Prerequisites* at least.

Step	Action
1	Start **Robotstudio** and open online window.
2	Create robot system with the help of the keys for the controller and drive module.
3	Enter the key for the option **RobotWare Machine Tending** in the window **Add additional options**.
4	Transfer new system to the robot controller.

More information on creating a robot system can be obtained from *Operating Manual - RobotStudio* listed in the section *References on page 11*.

3.3 Data Storage

Documentation

The complete documentation for RobotWare MachineTending is shipped with the robot controller.

The documentation consists of:

- Operating Manual - RobotWare MachineTending (3HAC044397-001)
- Application Manual - RobotWare MachineTending (3HAC044398-001)

The latest release notes can be found in the AdditionalOption folder of RWMT on the RobotWare installation data storage medium, shipped with the robot controller.

The license conditions are shown, when executing the setup program has been shipped with the robot controller.

Program examples

The RWMT additional option folder implements some programming examples in the subfolder **Program Example**, where one can see how RWMT can be implemented into a RAPID program.

This module can be used as a basis for the further programming.

ABB suggests to use the **Machine Tending PowerPac (MTPP)** (See *Operating Manual - Machine Tending PowerPac* listed in the section *References on page 11*) to generate user programs, that implement RWMT functionalities.

Stations

Normally there are peripheral machines, conveyors, slides , and so on, inside a robot cell, surrounding the robot. In RWMT language, they are called stations. Each station is represented by its own RAPID module.

The use of station modules is explained in the chapter *Setting up the graphic user interface (GUI) on page 37*

ABB suggest you to use the **Machine Tending PowerPac (MTPP)** (See *Operating Manual - Machine Tending PowerPac* listed in the section *References on page 11*) which provides some basic station templates.

Graphics for the user interface (GUI)

Some of the RAPID-data types that are used in RWMT allow the specification of a graphics file.

If such a graphics file is specified, then this will be shown on the user interface (GUI) of RWMT in the graphic element that corresponds to the corresponding data type declaration.

Such user defined graphics must be saved in the HOME or SYSTEM directory, so that they are recognized by RWMT.

Alternatively, if the RWMT projects concept is used, the graphics of a project can be saved in its specific project folder. Please refer to *Operating Manual - RobotWare Machine Tending* and *Operating Manual - Machine Tending PowerPac* listed in the section *References on page 11* to learn more about RWMT projects.

3.4 Notes on the next steps

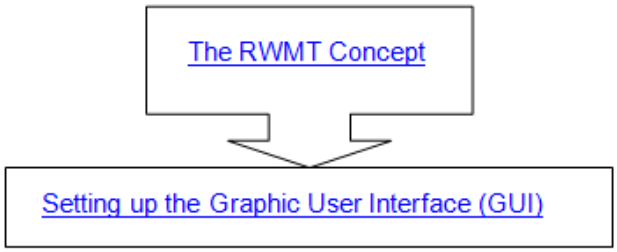
Introduction

After the installation, some steps are necessary for customizing the RWMT to suit the individual conditions, or to add data, instructions and functions to the application program for using the RWMT.

The chapters of this manual that are listed below give information about this. It is advisable to work through the respective chapters in the specified sequence, since they build successively on each other.

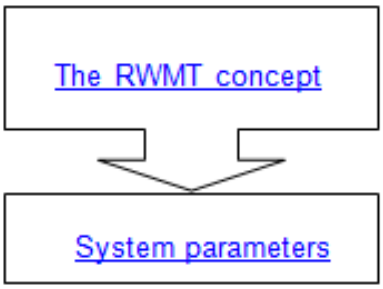
The chapters can be accessed as links if you are reading them on the computer.

Setting up the graphic user interface (GUI)



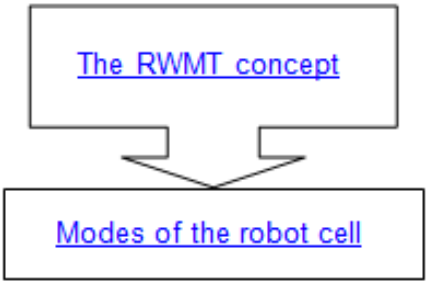
en1200000736

Parameterization of RWMT



en1200000737

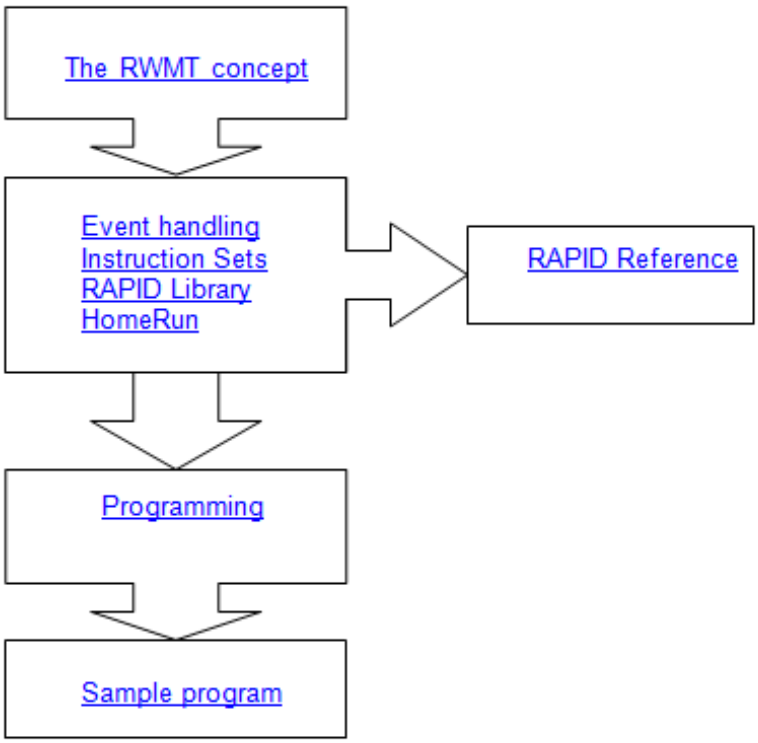
Understanding the concept of cell operation modes (modes)



en1200000738

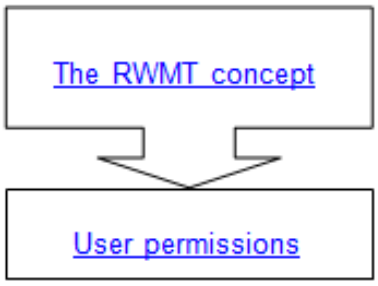
Continues on next page

Carry out RAPID-programming



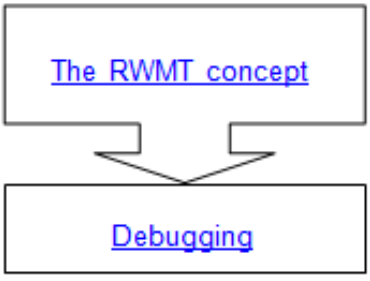
en1200000739

Assign user permissions



en1200000740

Initiate measures for detecting errors



en1200000741

This page is intentionally left blank

4 System characteristics

4.1 Introduction

The restrictions and system properties that are described below must be considered while programming and using the RWMT, to enable a fault free operation.

4.2 Restrictions

Multimove

RWMT is not suitable for use in MultiMove-Coordinated applications.

Apart from this, the RWMT is suitable for use in MultiMove-Independent applications.

Hot-Plug-Option for FlexPendant

RWMT has been conceived in such a way that the production can be ongoing without the RWMT GUI. Thus, it is theoretically possible to undock the FlexPendant during production from a robot controller that is equipped with the Hot-Plug function.

However , since RWMT can be configured in such a way that the robot of the action cell handling cell is controlled completely through the FlexPendant screen (for example, pre-selection of part type or start of production), every individual case should be examined whether the undocking of the FlexPendant is useful or not.

Thus, for instance, it would not be possible to stop the production or request the home position when the FlexPendant is undocked through the RWMT screen.

Furtheron, error messages can only be shown on a FlexPendant when being connected to the robot controller.

Safe return to starting position

While using the HomeRunning as part of the overall scope of RWMT, other restrictions may arise. These are described in the chapter *HomeRun on page 105*.

Language

Currently, the RWMT user interface is available only in *English* and *German*.

4.3 Properties

Maximum number of open windows

By using RWMT, the maximum number of open windows on the FlexPendant is reduced by one.

Background colours

The FlexPendant of the IRC5-controller is available in three different versions at the time of creating this documentation. These differ with respect to the display.

As a result, the representation of the background for graphics could differ depending on the type of the FlexPendant, for instance.

User permissions

In the delivery condition of the robot controls, the default user has all the user permissions. Thus, the default user can access all the functionalities of RWMT. If this is not desirable, then the desired users and user permissions should be created as shown in the chapter *User permissions on page 167*.

This page is intentionally left blank

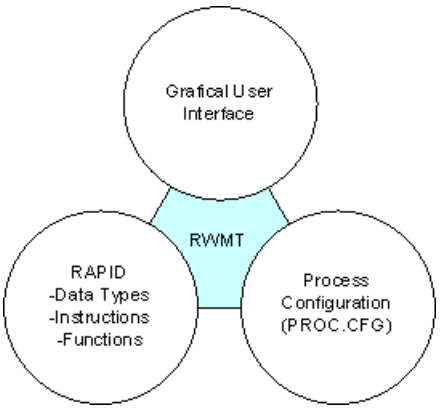
5 The RWMT concept

What is RWMT

RobotWare Machine Tending (RWMT) is a software option, which is used to make it easier to access the robot and the system peripherals in handling applications, both for the system operator as well as for the integrator.

Overview of the RWMT components

RWMT essentially consists of three components, as illustrated by the following figure:



en1200000742

The **user interface** gives the operator and programmer an overview of the handling cell with all its stations such as the processing machines, bands, slides and it also contains control functions.

A library of **RAPID data types, instructions and functions** supports the integrator while creating the robot programs and in designing the details of the user interface with graphic elements and information.

RWMT makes it possible to include external signals as well, through the **process configuration**, such as for the communicating the program numbers, the cycle pre-selection, error notifications or the safe return to starting position. If the signal interfaces are missing, however, it is also possible to use these functionalities directly at the operator screen of RWMT.

The following sections deal with these three components in greater detail and provide references to the corresponding chapters in this manual or in the *Operating Manual - RobotWare Machine Tending* listed in the section *References on page 11* for in-depth information.

The user interface

The user interface provides the following functionalities:

- Visualization of the operating states and production processes
- Gripper actuation and gripper monitoring
- General and station wise view and control of signals
- Station wise view and control of RAPID variables
- Selection of part types for the production

Continues on next page

Continued

- Definition and selection of production cycles (run-in (start up), run out, idle runs, and so on)
- Execution of setup and service routines
- Advanced Hot Edit (correction of positions during production)
- Execution of further FlexPendant applications within the RWMT
- Safe return to starting position
- Messaging
- Project interface to the associated MachineTending PowerPack

In order to be able to represent these functions, the user interface should be told which stations (that is, machines, conveyor belts, , and so on.) are present in the cell, which signal interfaces are available, which service routines are available, and much more.

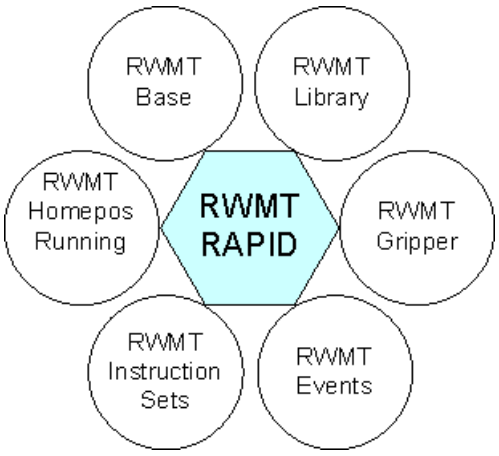
This information is made available exclusively through RAPID data declarations, instructions and functions and through process parameters. The integrator does not need any further knowledge of other programming languages apart from knowledge of the programming language RAPID and the handling of system parameters of the IRC5-robot controller.

Furthermore, this concept also makes it possible to integrate the RWMT in existing robot cells too, because, to put it simply, the only thing this requires is the inclusion of additional data, commands and functions in the robot program.

The scope of the integration here is almost unlimited. Sub-aspects of the user interface (GUI) can be used; other aspects can be left out or included at a later point in time. This often meets the requirements of narrowly measured set up and testing times in production cells.

RAPID data types, instructions, and functions

Firstly, the functional sub-division of the RAPID-libraries of RWMT can be illustrated with the help of the following image.



en1200000743

According to this diagram, RWMT consists of a total of 6 blocks that can be differentiated from each other functionally, and each of which is defined by a more or less extensive set of data types, instructions and functions.

Continues on next page

The declaration of special data types in RAPID creates icons or symbols, control elements, signal tables, and so on, on the RWMT user interface.

Changes such as the station or messaging tasks that are to be handled currently by the robot are supported through special instructions.

Pre-selections of the operator screen can be queried with the help of special functions.

The use of these individual blocks and their functionalities is optional, but this will also have an impact on the scope of functions of the user interface (GUI). Thus, for instance, the Block **RWMT Gripper** and its data, instructions and functions can be excluded from use in the application program. No gripper actuation or gripper view will be available in the user interface (GUI) in that case.

The following table gives an overview of the functions of the individual blocks and references to the respective chapters that contain more information and details about them.

Block	Explanation and references
RWMT Base	This block contains the RWMT-Engine as the core element. This is loaded in the application program in the main() routine and defines the course of the program with the help of various constraints. Furthermore, this block contains all the necessary data-declaration for displaying graphic elements on the user interface (GUI) of RWMT and for visualizing the instructions and functions and querying the system states. The data types that belong to this function block, and the graphic elements that are created with it can be obtained from the chapter *Setting up the graphic user interface (GUI) on page 37*. For a precise description of every individual data type, refer the chapter *Data types on page 245*. The available instructions and functions are described in detail in the chapters *Instructions on page 307* and *Functions on page 457*.
RWMT Library	The RWMT Library provides a library of instructions and functions for the following areas: • Waiting fordigital signals and signal groups position calculations • Tool computations • Dynamic calls of movement routines The description of the corresponding instructions and functions is available in the chapters *Instructions on page 307* and *Functions on page 457*.
RWMT Gripper	RWMT Grippers contains data types, instructions and functions for actuating and controlling the robot grippers and for requesting gripper sensors and component control sensors that are present on the gripper The available data types make it possible to represent simple to very complex grippers. If the data types that are offered are used, then the concerned grippers and their functionality are visualized automatically on the user interface of the RWMT. The chapter *Grippers on page 72* provides information about this concept of gripper administration in RAPID. The description of the corresponding data types, instructions and functions is available in the chapters *Data types on page 245* and *Functions on page 457*.

Continues on next page

Continued

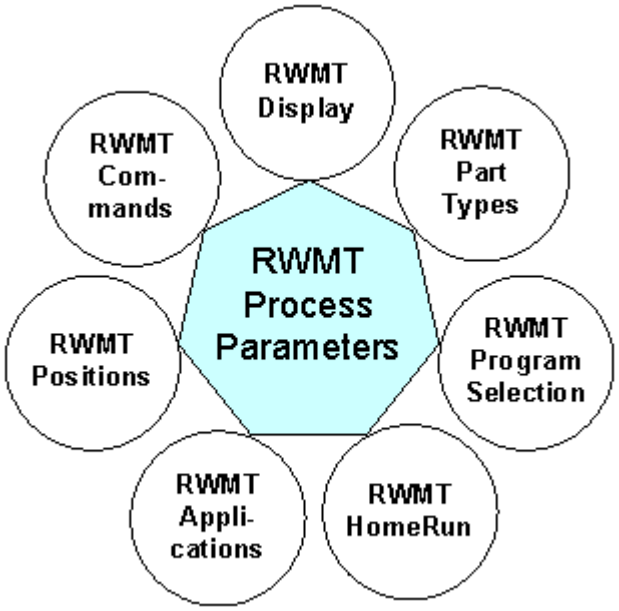
Block	Explanation and references
RWMT Events	With the functionalities of the RWMT Events, routines can be associated with the application program with pre-defined events. In this way, the routines will be executed as soon as the corresponding event occurs. The association of routines of the application program with pre-defined events is done through declarations based on the data type `eventdata`. Details of the usage are given in the chapter *Event handling on page 93*. The description of the data type eventdata that is necessary for use is given in the chapter *Data types on page 245*.
RWMT Instruction Sets	The Instruction Sets consist of declarations of the data type instset. With the help of these data type declarations, selected signals and variables can be set automatically to a pre-defined value on leaving or entering the RWMT mode **production** or when using the key switch to change the robot operation mode between **manual** and **automatic**. Details of the usage can be taken from the chapter *Instruction sets on page 99*. The description of the data type instset that is necessary for use is given in the chapter *Data types on page 245*.
RWMT HomeRun	HomeRun makes it possible to return the robot automatically from any position to the home position, by the click of a button. It supports the quickest possible restoration of the initial state of the production unit e.g. after any errors. To learn more about HomeRun, refer to the chapter *HomeRun on page 105*.

Continues on next page

Process parameters

RWMT provides an area, in which the product can be configured and parameterized as per the usage conditions.

This area consists of the so-called process parameters (PROC.CFG) of the robot controls. For RWMT, four main areas of configuration are offered, as per the following illustration.



en1200000744

The following table gives a brief explanation of these configuration areas and gives references to the corresponding chapters for details.

Domain	Explanation and references
RWMT Display	The display parameterization has an influence on the appearance of the user interface (GUI), the presence of buttons for specific user actions. Details are available in the chapter *MT Visualization settings on page 152*.
RWMT Com-mands	The command parameters are also used, among other things, for linking external signals for selecting the mode of operation or for selecting the current speed overrides. The chapter *MT API Commands on page 154* gives information about the possible settings.
RWMT Positions	The RWMT concept envisages certain default positions of the robot, such as the home position, a safeposition and up to 3 service positions. The position parameterization defines, among other things, which digital signals will be used for requesting a specific position of the robot, and which signals show that the robot in is in a particular position. Details are available in the chapter *MT API Positions on page 157*.
RWMT Program Selection	If a program pre-selection or a service request has to come from an external source, then, normally, a digital signal interface has to be specified. The parameterization of the program pre-selection also links such interface signals with the corresponding RWMT-functionalities. Details are available in the chapter *MT Program Selection on page 159*.

Continues on next page

Continued

Domain	Explanation and references
RWMT Part Types	RWMT supports different part types to be handled in one program. It provides appropriate parameters in the process configuration. Details are availble in the chapter *MT Part Settings on page 162*.
RWMT Home Run	Home Run makes it possible to return the robot automatically from any position to the home position, by the click of a button. The parametrization allows e.g. to modify the behavior of this functionality and to define, by which signal it is triggered. To learn more about HomeRun, refer to the chapter *HomeRun on page 105*.
RWMT Applications	The application parameters are used to integrate other flexpendant applications into RWMT. This can be either customer applications that have been programmed with FlexPendant SDK or ScreenMaker, or integrated basic applications like the production view The chapter *MT Applications on page 163* gives information about the possible settings.

Concept for the cell operation mode

The RWMT concept for the mode of operation takes into account various requirements for a handling application such as a mode of operation for the production , an exclusive mode of operation for service routines (gripper inspection, gripper replacement, automatic cleaning , and so on.).

Further, there are sub-modes of operation such as the production without parts (ghost mode), in which the logical production processes are examined, but without actually handling the parts or components.

The RWMT concept for the cell operation mode is explained in the chapter *Mode of operation of the robot cell on page 173*.

User permissions

RWMT contains various access options such as access to the production process, the peripherals or the service routines. It is not necessary that all users should have all these access options.

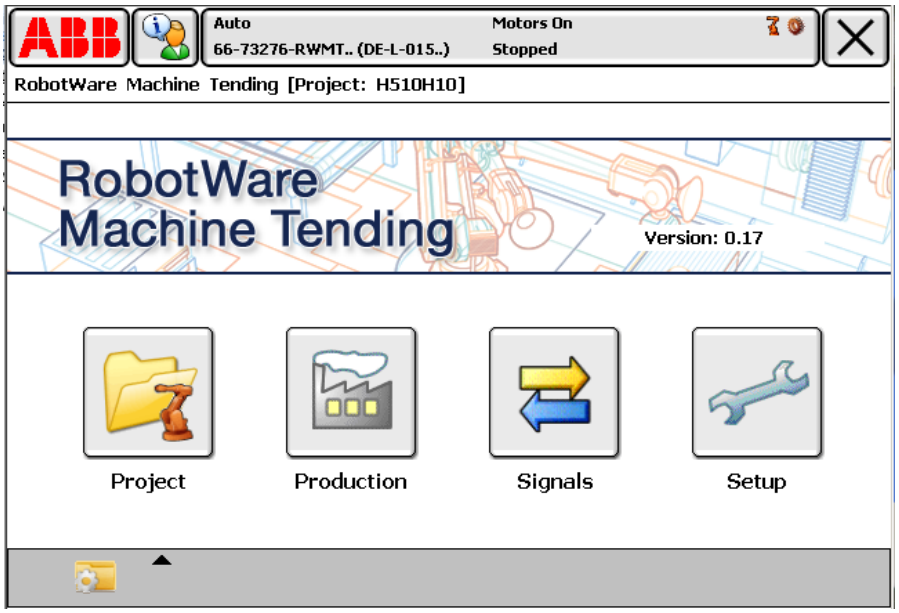
The chapter *User permissions on page 167* explains which user permissions of the RWMT are available to the system operator, the service staff, or the programmers.

6 Setting up the graphic user interface (GUI)

6.1 Startup view

Purpose

The startup view splits the GUI into 4 different main sections, that can be reached by clicking one of the buttons as shown in the following figure:



en1300000016

Main section	Explanation
Project	Project section allows the loading, unloading, import and export of projects (consisting of RAPID programs, depending parameters and other files, that belong to a certain application).
Production	Prodcution section allows you to visualize the different machines (stations) as well as the production process and allows to operate the cell.
Signals	Signals section allows you to see the general signals, which do not belong to a certain station
Setup	Setup section contains a list of setup routines, that can be executed only in manual operation mode.

The content of each of those main sections is normally created by RWMT data declarations in the RAPID program The next chapters will explain, how this is done.

6.2 Project view

6.2.1 General

The project view  and the projects by narrower sense of RWMT build the interface between the Machine Tending PowerPack (MTPP) and RWMT itself. A project is normally created by the MTPP.

- Program & system modules
- System parameters
- Part- and station-related images
- RobotStudio **Pack and Go** stations

The RWMT GUI allows the following opterations on projects

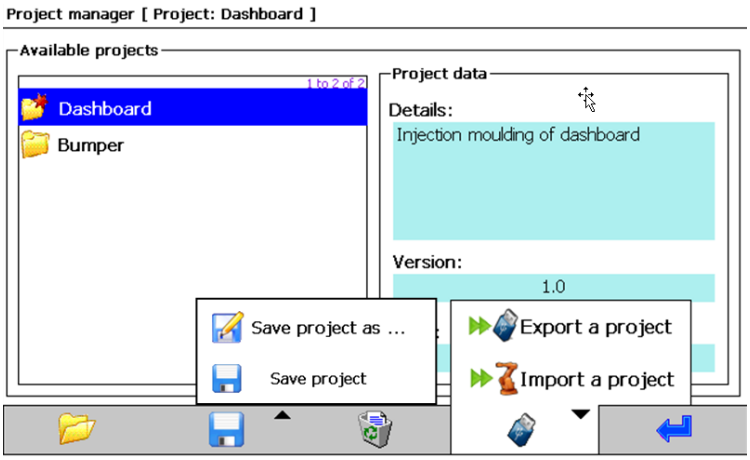
- Load a project for execution or to unload it
- Save a project with current name or with new name (copy)
- Import, export and save projects to/from USB storage device

6.2.2 Identification

To identify a project, a persistent declaration of type projectinfo must be available in any module of the first motion task (T_ROB1). An already loaded project then is recognized by the projectinfo.title entry and will be displayed in the RWMT GUI.

Example:

```
TASK PERS projectinfo piProject:=[“Bumper”,”Producing bumpers ..”,
     ”1.0”,”2012-10-22”];
```



en1300000018

The content of the project info declaration has to be equal to the project file (*.MTP), which is delivered together with the project by the MTPP.

A loaded project is marked as unknown, if the projectinfo title does not match the name of the related project. This can mostly happen, if the project title has been changed manually or the project files are not imported to the project folder.

A project marked as unknown can be saved, by using the menu entry **Save project as ....**

By default, projects are saved under HOME:RWMT. In some cases, the project folder size might become too big and this can obstruct the creation of robot backups, because the home directory content is part of the backup.

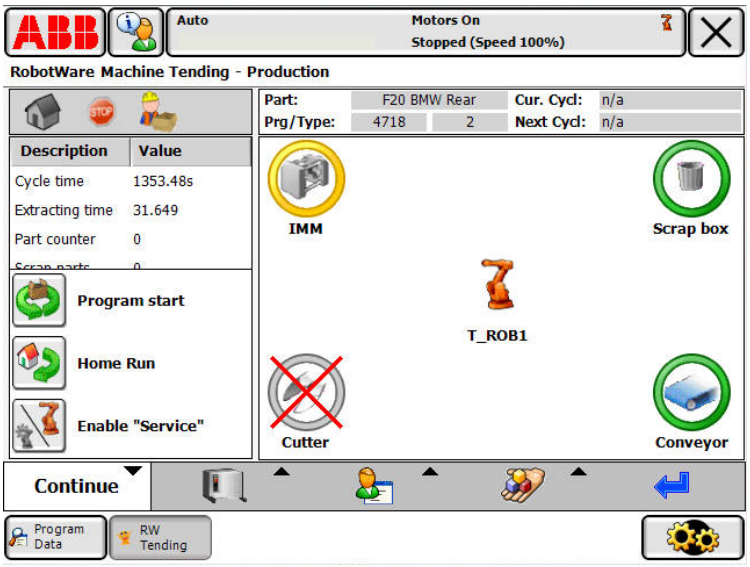
Therefore, the project path of a real controller can be modified in the system parameters (for more information, see *MT Visualization settings on page 152*). This setting will not influence the project path of a virtual controller, where the projects are always saved under HOME:RWMT.

6.3 Production view

6.3.1 General

Purpose

The production view  of the **RobotWare Machine Tending** is meant for visualizing the production processes and operating the robot cell.



en1300000020

Information in the production window

The following control elements and information are displayed in the production window:

- Status of the individual stations (ready, busy, error)
- Station selected or deselected
- Highlighting the concerned station in which the robot is currently situated
- Name of the current product
- Current program number
- Cycle time
- Display general information, based on data declarations for `bool`, `num`, `dnum` or `string` declared as persistent.
- Messages from the robot program, e.g. errors or information texts
- Buttons to operate (start, stop, and so on) the robot

Continues on next page

Information in menus

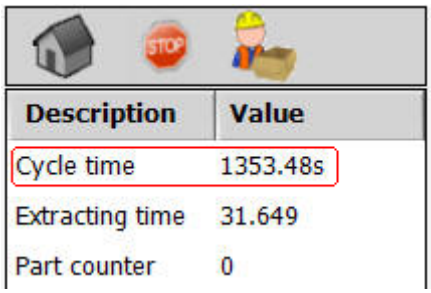
The following functions are provided through further sub-menus:

- Display of the station-specific variables and signals
- Manual operation of the gripper
- General signal page
- Cycle settings menu
- HotEdit for changing the positions
- Part data display
- Manual selection of the part that is to be finished
- Service and setup-menu for configuring the program

6.3.2 Production Information

Cycle time

The cycle time that was last required is displayed by default in the production info display and the current cycle time is displayed during a run.



en1200000746

Other Production Data

To be able to display other data of the type `bool`, `num`, `dnum` or `string`, the data field

```
CONST infodata MT_InfoView{xx}:=[..];
```

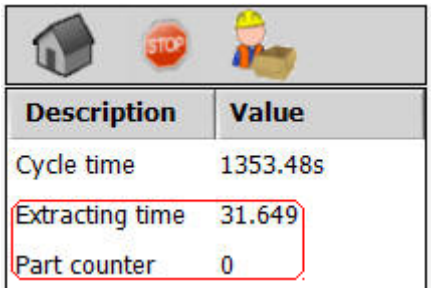
should be created and populated with data.

The data, whose values are to be displayed, must be declared as `PERS`, `LOCAL PERS` or as `TASK PERS`.

The production data display can show upto eight different data at a time, including the cycle time. If more than seven additional production data are defined, then, it is possible to scroll through the production info display using the scroll arrow that will be displayed.

Example:

```
PROC MainModule

TASK PERS num nExtractTime:=35;
TASK PERS num nPartCnt:=411;

CONST infodata MT_InfoView{2}:=[
["Extracting Time","nExtractTime","MainModule","T_ROB1"],
["Part Counter","nPartCnt","MainModule","T_ROB1"]];
ENDMODULE
```



en1200000747

6.3.3 User program messages

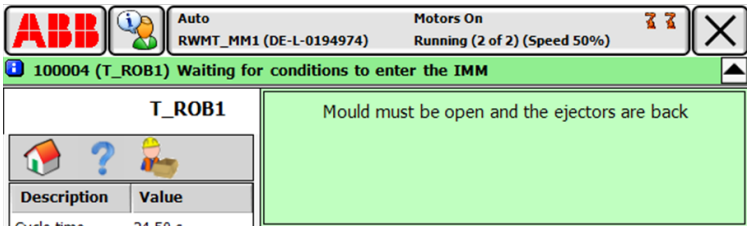
Types of message output

Basically, there are two types of message output available:

- Output takes place through the standard functions of RAPID
- Output takes place through the GUI

While outputting the messages through the graphic user interface (GUI), the advantage is that it is always in the foreground and is not overlapped by external message windows.

However, there is no acknowledgement for the messages that are output in the GUI. In this case, it is necessary to yield to the standard-RAPID-commands.



en1300000021

Message output in the GUI

Messages in the GUI can be output with the help of the data types msgdata and the instruction `MT_ShowMessage` or, more simply, through the instruction `MT_ShowText`.

With the help of the instruction `MT_ClearMessage`, these message outputs can be deleted if they are not required any more.

Example:

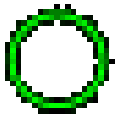
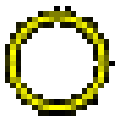
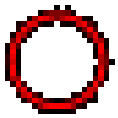
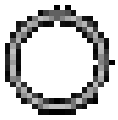
```
MODULE Messages

TASK PERS msgdata msgEnter:=[10,1,0,"Robot inside station
     xyz","","","","","",1,""];
PROC MessageTest()
!Show message
MT_ShowMessage msgEnter;
!...
!Robot does something inside the machine
!...
!Clear message
MT_ClearMessage;
ENDPROC
ENDMODULE
```

6.3.4 Stations

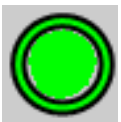
6.3.4.1 Introduction to Stations

Station states

A station  within the RWMT can be a machine, a conveyor, and so on, which the robot passes through during its program run. Within the production window, every station is represented by an image with a coloured ring, which reflects the status of the station. The various colours here have the following significance:

Station ring	Meaning
 green	The station is ready for the robot
 yellow	The station is busy or is working.
 red	The station has an error
 grey	The status of the station is undefined

Other states of the station are represented through additional symbols or by filling in the ring:

Station ring	Meaning
 red ring, crossed through	The station has been deselected
 green, filled ring	The robot is inside the station

Continues on next page

Preparing the station library

In order that stations can be reused, a separate program module, in which all the station-specific data and routines are present, should be created for every station.

To define a station, a data declaration for the data type `stationdata` is necessary in the station module as `TASK PERS` or `LOCAL PERS`.

If one does not wish to create any station library, then the station data declarations can be created in any desired module, wherein a module can also contain even several station data declarations.

```
LOCAL PERS stationdata IMM_Station:=
["IMM","IMM","Machine to build plastic parts",
"station-IMM.png","IMM_sdiEn_OPMode",
"IMM_sdiMouldClosed","","",TRUE,FALSE,1,1];
```

Station name

Through the station name (`stationdata.name`), the station prefix is defined, which is used to access the variables and signal declarations belonging to the station.

This naming convention must be adhered to, so that the RWMT-user interface (GUI) can assign the corresponding variable lists and signal lists to the corresponding stations.

Example:

```
TASK PERS stationdata IMM_Station:=["IMM1","",…];
const stationsignal IMM1_SIGNALS{2}:=[[…]];
const stationvariable IMM1_Variables{2}:=[[…]];
```

In this case, the station declaration `IMM_Station` with the name `IMM1` also includes the station signals of the declaration `IMM1_Signals` and the station variables of the declaration `IMM1_Variables`.

Station label

The station label is used to label the station symbol in the production window of the GUI. If a blank string is used for the station label (`stationdata.Label := ""`), then the station name is used for the label.

The station label can be used to display the label in the GUI in another language, while the name of the station remains the same.

Station description

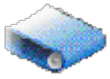
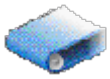
The station description is used for displaying any abbreviated or cryptic station name in plain text on the user interface (GUI).

Continues on next page

6.3.4.1 Introduction to Stations
Continued

Station images

The following station images are provided by RWMT and can be used directly by using the name of the file:

Image	File name
	station-blank.png (station icon which will be used as default, if nothing else has been defined).
	station-conveyor-in.png
	station-conveyor-out.png
	station-ctrlpanel.png
	station-cutter.png
	station-fan.png
	station-flaming.png
	station-gear.png
	station-gripper.png
	station-IMM.png

Continues on next page

Image	File name
	station-insert.png
	station-marking.png
	station-pliers.png
	station-press.png
	station-quality.png
	station-rack.png
	station-robot.png
	station-saw.png
	station-scrap.png
	station-shower.png

Continues on next page

Image	File name
	station-tape.png
	station-vision.png

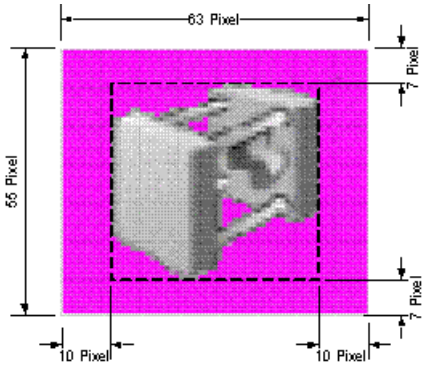
Example:

```
LOCAL PERS stationdata IMM_Station:=["IMM","IMM",
"Machine to build plastic parts",
"station-MM.png","IMM_sdiEn_OPMode",
"IMM_sdiMouldClosed","","",TRUE,FALSE,1,1];
```

If separate station images are to be used, the following points should be noted:

- The image should be 63 x 55 Pixel in size.
- The symbol should be located in the middle of the image and should leave adequate place at the edges, so that the station ring does not overlap with parts of the symbol.
- The background color of the image should be magenta (R: 255, G: 0, B:255).
- The images should be saved in the directory **HOME:**, **SYSTEM:** or **HOME:RWMT/IMAGES**.
- The image should be of the type **JPG**, **PNG** or **GIF**.

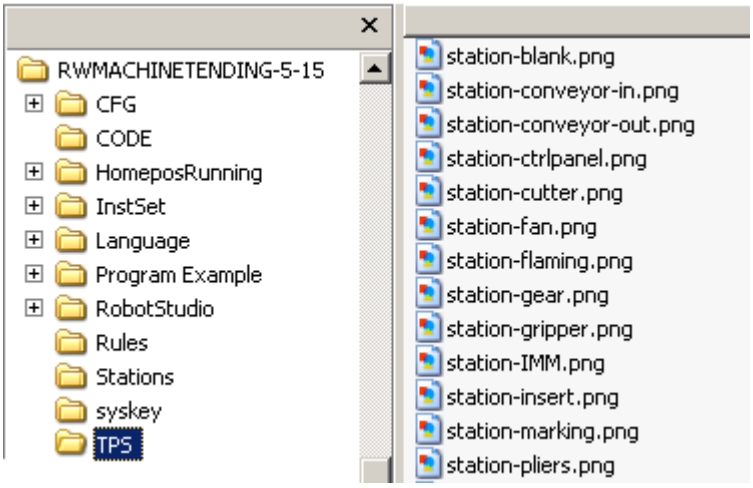
Example:



en1200000748

Continues on next page

The station images, provided by RWMT can be used as a template for customized images. After installing a robot system, which covers the RWMT option, the images shown in the table above are situated on the flashdisk in the following location.



en1200000749

Station Status

The station status shows whether a station is **ready** or **busy**, has an **error** or has been **deselected**.

In order to be able to update the status display automatically, the signals that define the respective status should be specified in the station declaration.

For the definition of a status message, digital inputs and outputs, as well as persistent Boolean declarations can be used

Note

The definition of the station status is used only within the graphic user interface (GUI) and does not have any impact whatsoever on the robot program.

Since a status can also arise as a result of **combination of several signals**, the following notation can be used to link them.

Logical symbol	Meaning
*	Inversion of signal or the persistent entity that follows the symbol.
&	Logical AND connection
!	Logical OR connection

Example:

```
TASK PERS bCNV_Ready:=FALSE;
stationdata.ReadyState:="diCNV_Automatic
& *diCNV_Running
! bCNV_Ready" ;
```

Continues on next page

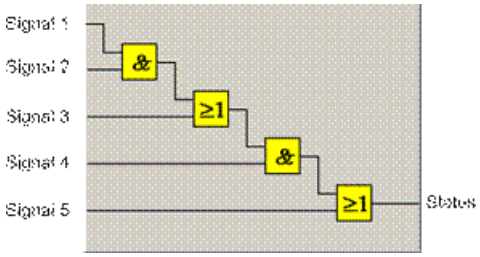
6.3.4.1 Introduction to Stations
Continued

The station (here a conveyor) is ready, if it is in the automatic mode (`diCNV_Automatik=1`) and if the conveyor is not running (`diCNV_Running=0`) or the Boolean persistent `bCNV_Bypass` is set to the value `TRUE`.

> **Note**
>
> Since a status definition cannot be longer than 80 characters (maximum length of a string in RAPID), the length of the individual signal names should be noted, that is, the longer the name of the signal, the fewer signals that can be linked

If it is not possible to accommodate all the signals in the declaration string, then a logical cross connection in the EIO.CFG can be used and specify its result as the status.

Here, the evaluation of the signal definition is done from left to right, by linking the first signal with the second signal. After this the result is linked with the third signal, and the result of this is then linked with the fourth signal, , and so on.



en1200000750

If signals are not available for every status for a station (**Ready**, **Busy** or **Error**) , then a blank string is used.

Depending on the status, this is evaluated as follows:

Status	Evaluation
Ready State	Station is always ready
Busy State	The status is not considered in the display
Error State	The status is not considered in the display

Use of ALIAS signals

To build a station library, station modules can be created in a general manner by using alias names for the station signals. The allocation of the signals of the respective station to the alias names in the program module is done through the `ALIASIO` instruction.

In order to be able to use these alias names for the station status declarations as well, the allocation of the names of the station signals and the alias names is done through the station signal declaration (See data type `stationsignal`).

Example:

```
VAR signaldi IMM_sdiEn_OPMode;
VAR signaldi IMM_sdiMouldClosed;

LOCAL PERS stationdata IMM_Station:=["IMM","IMM",
"Machine to build plastic parts",
```

Continues on next page

```
"station-MM.png","IMM_sdiEn_OPMode",
"IMM_sdiMouldClosed","","",TRUE,FALSE,1,1];

const stationsignal IMM_SIGNALS{2}:=
[[”IMM in automatic”,"diEn_OPMode","IMM_sdiEn_OPMode"],
[”IMM closed”,"diMouldClosed","IMM_sdiMouldClosed"]];
```

Using a signal for the station selection

Depending on the system, it may be necessary to remove a station from the finishing process, for instance, because it has a error or because the station should not be used right now.

To simplify the selection and deselection of a station, the current status is displayed in the production window.

Depending on the system concept, the deselection of a station can be done through the following options:

- a digital input
- a digital output
- the graphic user interface (GUI)

If the station is selected or deselected through a signal, it should be ensured that only one signal can be used in the station declaration.

A station is selected, if the signal has the status **high** and it is deselected if the signal has the state **low**. The state of the signal can be inverted if the character * is used before the name of the signal.

If it is necessary to use several signals, then, a logical cross connection should be created in the system parameters. Their result will be used in the station declaration.

Note

The deselection of the station through the GUI is not possible if a digital signal is used.

Parameterization of the station selection

The station selection is set as follows:

Function	System Parameter	
	ExtEnable	AllowDisable
Station should never be deselected	`stationdata.ExtEnable:= ""`	FALSE
Station selection should be done through the GUI	`stationdata.ExtEnable:= ""`	TRUE
Station selection is done through digital input or output.	`stationdata.ExtEnable := "diWithCNV"`	FALSE
Station selection is done through digital input or output with inverted signal function	`stationdata.ExtEnable:= "*diWithoutCNV"`	FALSE

In the robot program, the station selection should be considered accordingly in the program run.

Continues on next page

To do so, the `MT_StationIsEnabled` function can be used, which takes into account the station selection through a digital signal as well as through the GUI.

Example:

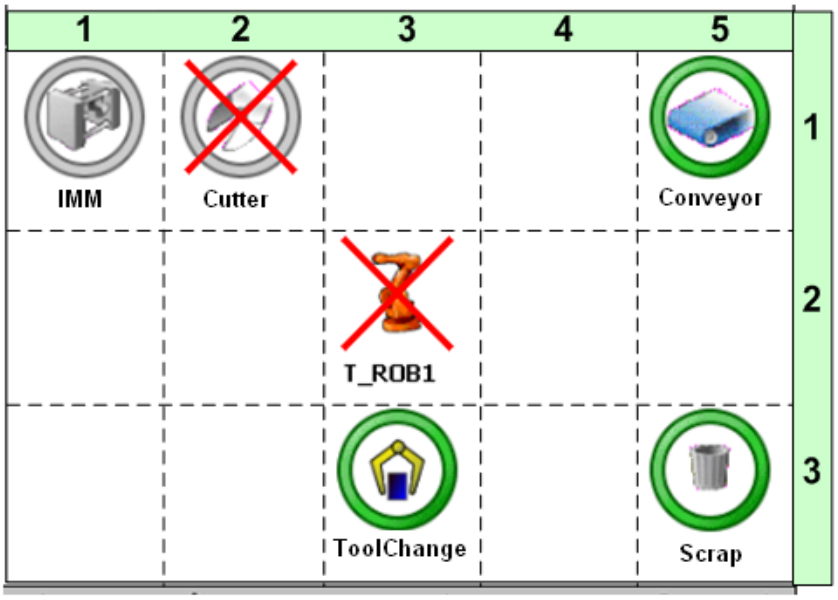
```
PROC ProgFlow()
Unload_IMM;
IF MT_StationIsEnabled(sdFLAMING) THEN
Flame;
Load_Conveyor;
ELSE
Load_Slide;
ENDIF
ENDPROC
```

If a station may not be deselected, then the parameter `AllowDisable` of the `stationdata` declaration has to be explicitly set to **NO**.

Position of the station symbols

The position of the station symbols (icons) in the production window is defined by specifying the column (`stationdata.Column`) and the row (`stationdata.Row`).

On the whole, 5 columns and 3 rows are available:



en1200000751

The robot station is always displayed in the middle of the station matrix (column 3, row 2). If necessary, the robot station can also be customized in the system parameters at the following places:

```
PROC/MT_GUI_SETTINGS/RobIconCol
PROC/MT_GUI_SETTINGS/RobIconRow")
```

Continues on next page

If the same position is specified for several stations, these will be shifted automatically to the next free place.

Note

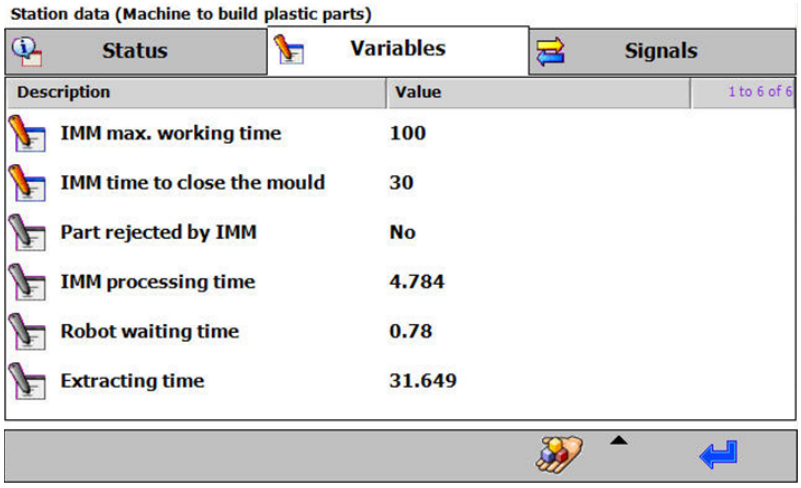
A maximum of 14 stations can be displayed in the production window.

6.3.4.2 Station variables

Display in the GUI

For every station, the values of the data declarations of the type `bool`, `num`, `dnum` or `string` can be displayed and modified.

In every station page, at the most two variable pages can be displayed, each of which can have one or two lists with data declarations.



en1200000752

The number of data declarations that is to be displayed in a list is unlimited. It is possible to scroll through the list using the scroll arrows, in case a list has more than 6 entries.

Creating lists of variables

Every list is defined through an array declaration of the data type `stationvariable`, whose name must begin with the assigned station name (See `stationdata.Name`).

The arrays are named as follows:

Page	List	Name of the array constant
1	1	`<stationdata.Name>_Variables` or `<stationdata.Name>_Variables1`
1	2	`<stationdata.Name>_Variables2`
2	3	`<stationdata.Name>_Variables3`
2	4	`<stationdata.Name>_Variables4`

If only one array is declared for a page, then the list will occupy the entire width of the FlexPendant, so that the descriptive text and the data that is to be displayed can be lengthier (See above).

Example:

```
TASK PERS stationdata IMM_Station:=["IMM",…];
const stationvariable IMM_VARIABLES{4}:=[
["Unload Time","nIMM_UnloadTime","IMM","",0,0,FALSE,
FALSE,FALSE,0.0],…];
```

Continues on next page

Changing the values of the variables in the GUI

In order to be able to change station variables in the GUI, this has to be set explicitly for every data declaration in the array.

The following options are available for doing this:

- Using data from any desired tasks and modules.
- Persistent data is updated automatically, variables must be updated by pressing a key.
- All the data except constants can be changed.
- Changes can be allowed in the automatic mode.
- The release of a data change can be done depending on the user who is logged in, so that a **Configurer**, for instance, can change a station variable while the cell operator can only view these.

Additional options for numerical data (num or dnum)

- Resetting to an adjustable value through a button.
- Limiting the input by using a lower and an upper limit.

Example 1:

Parameterization of the various station variables for an injection molding machine for display and editing:

```
MODULE IMM
LOCAL PERS num nIMM_UnloadingTime:=0;
LOCAL PERS bool bIMM_WithCorePullers:=FALSE;
LOCAL PERS num nIMM_WaitingTime:=5;
TASK PERS num nIMM_PartCounter:=0;

TASK PERS stationdata IMM_Station:=["IMM",…];

const stationvariable IMM_VARIABLES1{4}:=[
["Unload Time","nIMM_UnloadTime","IMM","",0,0,FALSE,
FALSE,FALSE,0,0],
["Core pullers selected",
"bIMM_WithCorePullers","IMM","",0,
0.TRUE,FALSE,FALSE,0,100],
["Wait Time","nIMM_WaitTime","IMM","",0,20,
TRUE,TRUE,FALSE,0,10],
["Part Counter","nIMM_PartCounter","IMM","",0,0,
FALSE,TRUE,TRUE,0,10]];

ENDMODULE
```

Continues on next page

6.3.4.2 Station variables
Continued

Example 2:

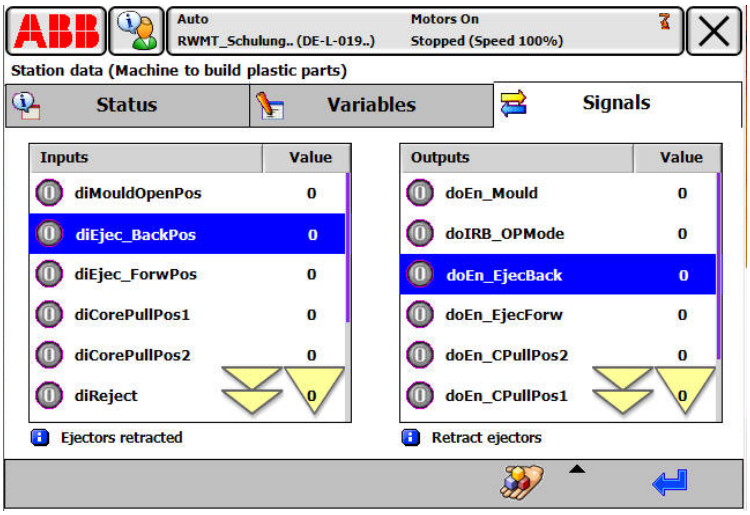
Parameterization of the station variables of a control station, in which the sensors for the control can be selected or deselected in the manual mode of the robot controls:

```
MODULE CTR

LOCAL PERS bool bSensor1:=FALSE;
…
LOCAL PERS bool bSensor12:=FALSE;

TASK PERS stationdata CTR_Station:=["KTR",…];
const stationvariable CTR_VARIABLES1{6}:=[
["Sensor1","bSensor1","CTR","",0,0,TRUE,FALSE,FALSE,0,0],
["Sensor2","bSensor2","CTR","",0,0,TRUE,FALSE,FALSE,0,0],
["Sensor3","bSensor3","CTR","",0,0,TRUE,FALSE,FALSE,0,0],
["Sensor4","bSensor4","CTR","",0,0,TRUE,FALSE,FALSE,0,0],
["Sensor5","bSensor5","CTR","",0,0,TRUE,FALSE,FALSE,0,0],
["Sensor6","bSensor6","CTR","",0,0,TRUE,FALSE,FALSE,0,0]];
const stationvariable CTR_VARIABLES2{6}:=[
["Sensor 7","bSensor7","CTR","",0,0,TRUE,FALSE,FALSE,0,0],
["Sensor 8","bSensor8","CTR","",0,0,TRUE,FALSE,FALSE,0,0],
["Sensor 9","bSensor9","CTR","",0,0,TRUE,FALSE,FALSE,0,0],
["Sensor 10","bSensor10","CTR","",0,0,TRUE,FALSE,FALSE,0,0],
["Sensor 11","bSensor11","CTR","",0,0,TRUE,FALSE,FALSE,0,0],
["Sensor 12","bSensor12","CTR","",0,0,TRUE,FALSE,FALSE,0,0]];

ENDMODULE
```

6.3.4.3 Station signals

Displaying signals in the GUI

Digital and analog signals; as well as signal groups can be displayed in the station view of the GUI for every station. All the outputs can be set in the manual or automatic mode of the robot, depending on the user settings.



en1200000753

Every list of signals is defined through an array declaration of the data type `stationsignal`, whose name should begin with the assigned station name (See `stationdata.Name`).

The array is named as follows:

`<stationdata.Name>_Signals`

Example:

Station signal array for the signals of a station:

```
TASK PERS stationdata IMM_Station:=["IMM",…];

LOCAL CONST stationsignal IMM_Signals{5}:=
[
[”IMM in automatic mode”"diAutomatic",""],
["IMM mould is open","diMouldOpen",""],
["IMM mould is closed","diMouldClosed",""],
["Start IMM cyclce","doStart",""],
["Forward IMM pusher","doFwdPusher",""]
];
```

In the signal array, all the signals should be listed in the sequence in which they are to appear in the list.

Continues on next page

6.3.4.3 Station signals
Continued

In the GUI, input signals (DI, GI or AI) are shown in the list on the left and output signals (DO, GO or AO) are shown in the list on the right; there is no subsequent sorting of the list.

Note

The alias names in the signal array are not taken into consideration while displaying the list of signals. The linking between the alias name and the signal name is done only in the case of the station status display and the status deselection.

Displaying the signal descriptions

If a descriptive text (`SignalLabel`) has been entered for a signal in the signal configuration (EIO.CFG), it will be shown in the list of signals as soon as the signal is selected.

Example:

```
EIO_SIGNAL:

-Name "diMouldClosed" -SignalType "DI" -Unit "IMM"\
-SignalLabel "Mould is closed" -UnitMap "0"
```

Use of ALIAS-signals

Several stations, which differ in their function only in terms of the signals used (e.g. loading several conveyor belts) can exist in a robot cell.

In order to be able to create a station module as a generally applicable template, access the station signals are accessed within the robot programs through alias names.

The allocation of the signals of the respective station to the alias names in the program module then takes place internally within RAPID, through the instruction `ALIASIO`.

If this allocation has to take place automatically in the robot program and if the GUI should also be able to access the link, the signal allocation is done through the station signal array that was described earlier.

Example:

Allocation of the signals of the real station to the signals that have been declared in the program module:

```
MODULE IMM
!Digital signals
LOCAL VAR signaldi adiIMM_AutoMode;
LOCAL VAR signaldi adiIMM_MoldOpen;
LOCAL VAR signaldi adiIMM_MoldClosed;
LOCAL VAR signaldo adoIMM_Start;

TASK PERS stationdata IMM_Station:=["IMM",…];

LOCAL CONST stationsignal IMM_Signals{5}:=
```

Continues on next page

```
[
["IMM in Automatic",diAutomatic","adiIMMAutoMode"],
["IMM mould is open","diMouldOpenPos","adiIMM_MouldOpen"],
["IMM mould is closed","diMouldClosed","adiIMM_MouldClsd"],
["Start IMM","doStart","adoIMM_Start"],
[["IMM pusher fwd"doPusherFwd",""]
];


PROC IMM_Unload()
Set adoIMM_Start;
…
ENDPROC
```

The allocation of the signals is done through the instruction `MT_AliasIO`, which is to be used by the integrator as follows:

```
CONST eventdata edIMM_START :=
EE_POWERON_OR_START,"IMM:IMM_EVT_START",1];

!Routine for initializing the signals
LOCAL PROC IMM_EVT_START()
!allocation of the local alias signals,
!if signal name and alias name have been used
MT_AliasIO IMM_Signals\ModuleName:="IMM";
ENDPROC
```

In RAPID, these signal declarations are read automatically when the event **Start** or **PowerOn** occurs and the corresponding signals are assigned through the instruction `ALIASIO`. If an alias signal does not have any allocation (""), then the signal will be displayed only in the GUI, and there is no allocation through `ALIASIO` within the RAPID program.

If the alias signal declaration is declared locally and not globally, the signals are to be allocated explicitly in the station module through the instruction `AliasIO`.

In the robot program, only these alias signals may be used, hence, if a station is used several times, no changes have to be made to the signals inside the program.

Example:

```
VAR signaldi sdiIMM_Unload;
Const stationsignal
IMM_SIGNALS{2}:=
[
["Release to unload IMM","diIMM_Unload","sdiIMM_Unload"],
["Start IMM","doIMM_Start","sdoIMM_Start"],...]
];
```

6.3.4.4 Station applications

General

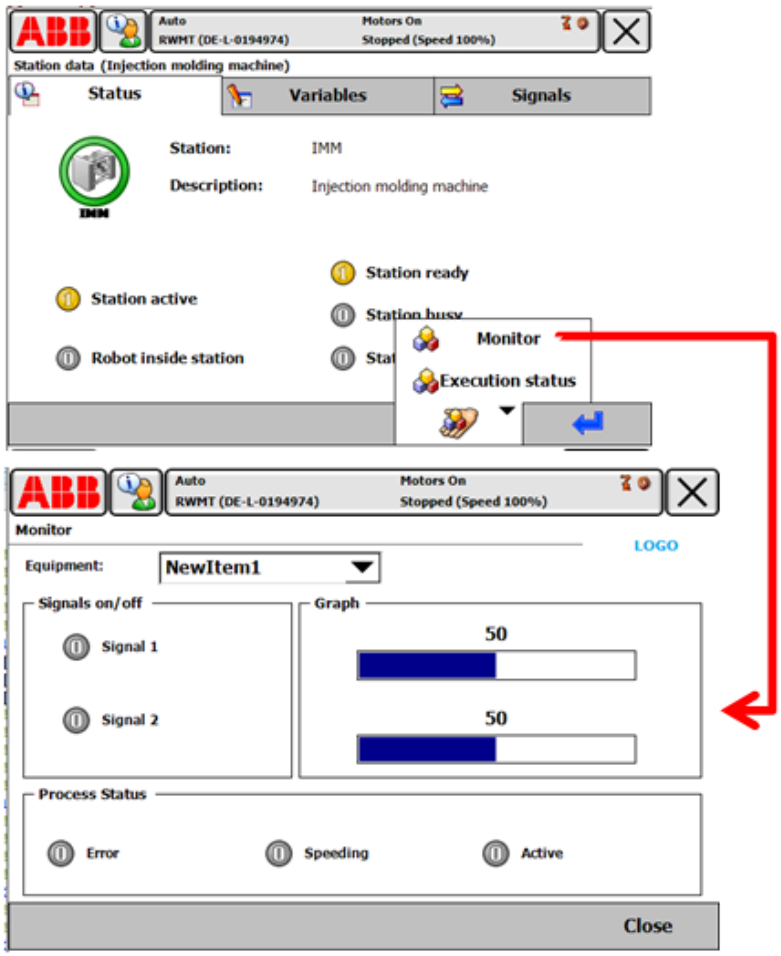
Application screens, created by means of ScreenMaker or FP-SDK can be used as so called **embedded screens** for each station.

The purpose of these station applications is to enlarge the possibilities to control or monitor the functionality of a station, if the variable and signal pages of RWMT donot fit the customer needs.

> **Note**
>
> Embedded means, that when having started such an application, the RWMT GUI view cannot be accessed until the application is closed. So each application must have a **Close** button, to be able to come back to RWMT.

For each application screen, customized images can be used for the menu representation in RWMT. Upto 8 different station applications can be applied to each station.

In the following example, 2 station applications have been assigned:



en1300000247

Continues on next page

When pressing the application button **Monitor**, the appropriate application view is started and shown as embedded, so that it overlays the RWMT. A **Close** button is available to terminate the application view.

Defining station applications

Station applications are defined as an array of data type `station app` (for more details, see *stationapp – External applications to be started in GUI on page 293*). The array is named as follows:

`<stationdata.Name>_Applications`

Array sizes from 1 up to 8 are allowed. The configuration is equal to the embedded **external applications**, described in a different chapter.

Example:

```
LOCAL CONST stationapp IMM_Applications{2}:=
[["Monitor","","TpsViewExtended.dll","Extended",
"Monitor:MainScreen"],["Execution status","",
"TpsViewExtended.dll","Extended",
"ExecutionStatus:MainScreen"]];
```

This declaration represents a station application with a the menu text **Monitor**, the application library `TpsViewExtended.dll`. Inside the application, the view **MainScreen** has been selected as the startup view.

Limitations

ScreenMaker application variables will only be initialized in the main screen of the application. If a sub window is opened directly it must be observed that application variables cannot be used.

If nevertheless application variables are used when opening a subwindow directly, an error will be generated when launching the view.

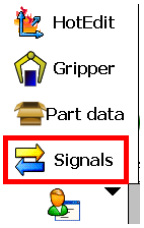
6.3.5 General Signals

Displaying signals in the GUI

General signals can be accessed in 2 different locations inside RWMT:

In the startup view

In the production view



Please refer to the chapter *General signal view on page 89*, to get a detailed description.

6.3.6 Part data

Use of part data

The part data, represented by the data type `partdata` form the basis for the program execution within RWMT. The program run and ultimately even the movement paths and positions should all refer to the part that has to be handled currently.

With the help of several declarations of the data type `partdata`, the following can be realized for different parts:

- Completely different production processes or
- a production process with different gripper and rest positions.
- A check of the program code and the gripper codes as well as up to 8 additional codes, that might be necessary to continue production

Declarations of the data type partdata create a visual respresentation in the part data submenu of the production view.

Explanation of the part data-components

A `partdata` declaration contains all the necessary information related to the part. The components that are most important for a further understanding will be explained in the following table. A complete list of all the components of `partdata` can be obtained from the chapter *Data types on page 245*.

Information in partdata	Explanation
Description	Description of the part type.
Routine that is to be called	This is the start up routine for the production process of the concerned part. Different, each of which can be described by a separate part data declaration, can thus have a separate production routine each. This is then called by selecting the corresponding part.
Motion tasks, where the part shall be executed	When having a multimove application, the robots might be executing production cycles with different part types, or they handle the same part types. So a decision must be taken in which motion tasks the part type is valid.
Selecting and deselecting cycles	Here, it is possible to decide if the concerned part has various production processes, or so-called cycles. More details of this can be obtained from the chapter *Program cycles on page 68*.
Program code and type code	One means of beginning the production for a part type is by making the higher controls send a program code to the robot controls. Another code may be necessary for the indexing within the robot programs. Hence, a part data declaration always contains a type code. This can be identical with the program code, but need not be. In this way, the indexing can be done in an independent manner for use in the higher order controls and in the robot controls.

Continues on next page

6.3.6 Part data
Continued

Information in partdata	Explanation
Tool code and the check codes	The tool code is the code of the tool (that is to be used by the robot). This is important if there could be a need to change the tool in the robot. The check codes could be form codes of pressing tools, for instance (press molding). The codes can be used to ensure that, for the part that is to be handled in each case, the correct gripper is engaged or that the peripheral machines are equipped properly If this is not the case, the production will not start.

Examples for parameterization

Example 1:

In the following sample code, there is a part type, **Type1**.

Type1 is a part type with the program number 1 and the type number 100. It can manage without cycles and it does not need any tool code or other codes as condition for the processing. If this part type is selected for the production, then, on being called, it will execute the routine **Production** in which the corresponding production process has been programmed.

The significance of the initialization values of every part type `partdata` can be obtained from the chapter *Data types on page 245*.

```
MODULE IMM
…
…
!***********************************
! Part type declarations
!***********************************
TASK PERS partdata pdPartType:=["Type1",
"Production ","",TRUE,1,100,1,
-1,[-1,-1,-1,-1,-1,-1,-1,-1], "Part1.GIF",
[1.5,[0,0,0.001],[1,0,0,0],0,0,0]
…


!***********************************
! Production routine
!***********************************
PROC Production()
…
UnloadMachine11;
LoadMachine2;
…
ENDPROC
…
…
ENDMODULE
```

Continues on next page

Example 2:

In the following sample code there are two part types **TypeA** and **TypeB**.

TypeA is a part type with the program number 2 and the type number 110. It can manage without cycles. **TypeA** requires the tool code 10 and a few more codes as condition for the processing. If this part type is selected for the production, the result is that the Routine **ProdA**, in which the corresponding production process has been programmed in detail, will be called.

TypeB is a part type with the program number 3 and the type number 200. It uses the start-up cycles, normal cycles and run-out cycles. It does require the tool code 11 and no other codes as condition for the processing. If this part type is selected for the production, the result is that the Routine **ProdB**, in which the corresponding production process has been programmed in detail, will be called.

The significance of the initialization values of every part type `partdata` can be obtained from the chapter *Data types on page 245*.

```
MODULE IMM
…
!***********************************
! Part type declarations
!***********************************
TASK PERS partdata pdPartTypeA:=["TypA",
"ProdA","",TRUE,2,110,3,
10,[8,1,9,-1,-1,-1,-1,-1],
"PartA.GIF",[1.5,[0,0,0.001],[1,0,0,0],0,0,0],""];
!
TASK PERS partdata pdPartTypeB:=["TypB",
"ProdB","",FALSE,3,200,4,
11,[-1,-1,-1,-1,-1,-1,-1,-1],
"PartB.GIF",[1.5,[0,0,0.001],[1,0,0,0],0,0,0]," "];
…
!***********************************
! Cycle declarations
!***********************************
!Definition of the cycle list
TASK PERS cycledata MT_CycleList{20}:=
[
["Start cycles","",1,1,1,0,2,0],
["Normal cycles","",2,1,10,0,3,0],
["Runout cycles","",3,1,2,0,0,0],
["","",0,0,0,0,0,0],
["","",0,0,0,0,0,0],
…
!***********************************
! Production routine
! Part type TypeA
!***********************************
PROC Production()
…
UnloadMachine1;
LoadMachine2;
```

Continues on next page

6.3.6 Part data
Continued

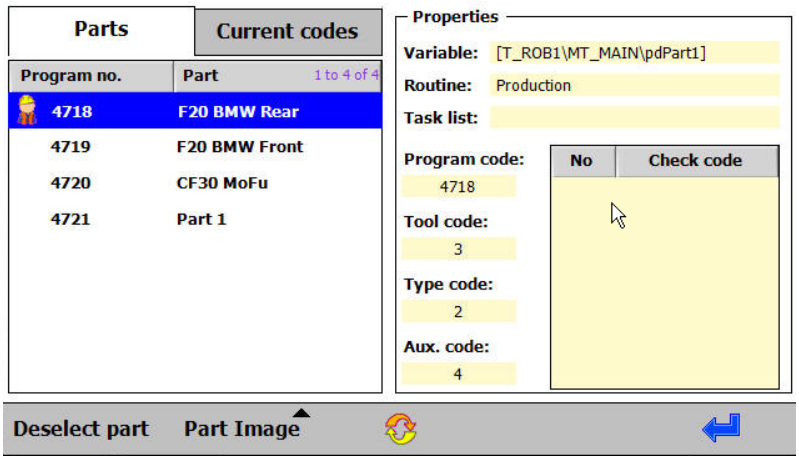
```
…
ENDPROC
…
…

!***********************************
! Production routine
! Part Type TypeB
!***********************************
PROC ProdB()
…
!If startup cycle is requested
IF MT_GetCycleIndex()=1 THEN
StartupCycle;
!If a normal cycle is requested
ELSEIF MT_GetCycleIndex()=2 THEN
NormalCycle;
!If a runout cycle is requested
ELSEIF MT_GetCycleIndex()=3 THEN
RunoutCycle;
ENDIF

…
…
ENDPROC
…
…

ENDMODULE
```

Continues on next page

Representation on the RWMT screen

The defined parts and their detailed information are displayed on the RWMT-screen for selection. The selection can be done manually or exclusively through the program number of an external higher order controls that has been transferred (see the chapter *MT Program Selection on page 159*).

Sample display:



en1200000755

The parts view that is shown, as well as the parts selection will be explained in detail in the *RWMT Operating manual* listed in the section *References on page 11*.

6.3.7 Program cycles

Use of cycle types

A program cycle describes the production process from the point of view of the robot, starting from the home position or safe position with the first handling action at a station upto its return to the home position or the safe position after completion of the last handling action.

In the following examples, there could be just one production process for a part type, or several processes could be required during the production for one and the same part type. In the latter case, RWMT supports the programmer with a concept for defining various cycles.

The cycles can be defined through a list `MT_CycleList{20}`, which is present in the delivered state of RWMT in the module `MT_Main`. This list thus supports up to 20 different cycles. It is based on the data type `Cycledata`, which is explained in the chapter *Data types on page 245*.

Explanation of the various types of cycles

To enable a handling of cycles that is as flexible as possible, different types of cycles have been implemented in RWMT:

Cycle type	Explanation
Continuous cycle	Continuous cycles are run without conditions. Normally, the execution is carried out till a *Halt after end of cycle* is requested. **Application example:** Production with recurring process
Counter cycle	Counter cycles are executed with the help of a counter, specifying the number of repetitions. **Application example:** Batch finishing of 100 parts, for instance
Action cycle	Action cycles will be executed only on request from the user interface. The number of direct repetitions is specified. **Application example:** Specific request for ejecting parts for manual quality control
Periodic cycle	Periodic cycles are called in a recurring (periodic) manner during the program execution. Here, it is necessary for this type of cycle to specify the number of cycles of another cycle type after which the periodic cycle should be executed. The number of immediate repetitions is also specified. **Application example:** Regular ejection of parts for manual quality control.

Sample applications

In the easiest case, there is only one (main) cycle, in which the process is always the same.

Example:

- Leave the home position / safe position
- Unload machine 1
- Load machine 2

Continues on next page

- Return to the home position / safe position

In many cases, however, it is necessary to envisage different cycles (that is, production processes), because different processes are required at the beginning and at the end of the production, than during a normal production cycle.

Example:

1 Filling cycles, execution 10 times in direct succession
   - Leave the home position
   - Unload machine 1
   - Load the storage 10 times
   - Return to the safe position
2 Normal cycles, execution till the halt after end of cycle is requested
   - Leave the safe position
   - Unload the storage 10 times
   - Load machine 2
   - Unload machine 1
   - Load the storage 10 times
   - Leave the safe position
3 Idle run cycles, execute till storage is empty
   - Leave the safe position
   - Unload the storage 10 times
   - Load machine 2
   - Leave the safe position

Once the storage is completely empty: Return to the home position and end of program.

Example for parameterization

In the following sample code, a start up cycle is parameterized, followed by a normal cycle and then the run out cycle. Here the start up cycle is executed once, the normal cycle is executed 10 times and the run out cycle is executed two times successively. Thus, these are exclusively counter cycles.

The meaning of the initialization values of each cycle data can be obtained from the chapter *Data types on page 245* of this manual.

```
MODULE IMM
…
…
!***********************************
! cycledata declaration
!***********************************
!Definition of the cycle list
TASK PERS cycledata MT_CycleList{20}:=
[
["Start cycles","",1,1,1,0,2,0],
["Normal cycles","",2,1,10,0,3,0],
["Runout cycles","",3,1,2,0,0,0],
```

Continues on next page

6.3.7 Program cycles
Continued

```
["","",0,0,0,0,0,0],
["","",0,0,0,0,0,0],
…
…

!***********************************
! Production routine
!***********************************
PROC Production()
…
!If startup cycle is requested
IF MT_GetCycleIndex()=1 THEN
StartupCycle;
!If a normal cycle is requested
ELSEIF MT_GetCycleIndex()=2 THEN
NormalCycle;
!If a runout cycle is requested
ELSEIF MT_GetCycleIndex()=3 THEN
RunoutCycle;
ENDIF

…
…
ENDPROC
…
…

ENDMODULE
```

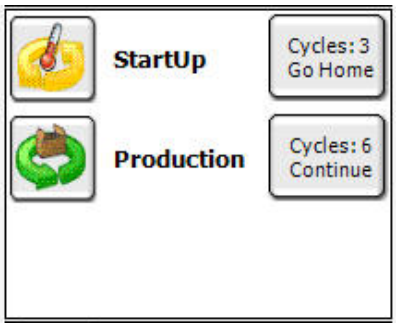
Representation on the RWMT screen

The defined cycles are displayed for selection on the RWMT screen if the selection is not done exclusively through an external signal interface (for this, see the chapter *MT Program Selection on page 159*).

If the production has not been started yet, then only counter cycles and continuous cycles will be offered for selection. Once the production has started, even action cycles can be selected. Periodic cycles, which are called automatically once the defined period has lapsed, cannot be selected.

Continues on next page

Sample display:



en1200000756

The description of the action that is triggered by the buttons is given in the *RWMT Operating manual* listed in the section *References on page 11*.

The cycles that have been created earlier can be edited in the RWMT screen. The description of the editing functions can also be obtained from the *RWMT Operating manual* listed in the section *References on page 11*.

External cycle selection

In addition to the cycle selection on the FlexPendant, this selection can be done remotely through digital IO signals.

6.3.8 Grippers

Overview

RWMT provides a RAPID-support for actuating and controlling the control elements of grippers (e.g. clamps or vacuum cups), for sequential actuation of several control elements as well as for checking part controls at the grippers. The actuators and part controls can be configured freely and can be customized for any desired gripper.

Through the declaration of gripper data (`grpdata, grpseq, grppart`), moreover, the corresponding gripper will also get a graphic representation in the GUI. Here, the status of the gripper can be shown and the gripper can also be actuated.

Control elements

For every control element, a control element configuration of the data type `grpdata` should be created.

An **Action** can be controlled on the gripper with a **control element**. Normally, a valve is switched with this. Up to 4 pairs of sensors (e.g. end position check in the case of cylinders) can be used for each valve. Through the configuration, control elements can be described with various options.

The configuration of the control elements is used with the instruction `MT_GripSet` and `MT_GripCheck`. Up to 6 actuators can be used simultaneously.

Example:

```
MODULE MT_MAIN

!Declaration for sucking gripper
TASK PERS grpdata gdSucker1:=
[
"Sucker",1,TRUE,0.2,TRUE,TRUE,
["doVacuum1On",0,"doBlow1On",0.2,
"Vacuum on","Vacuum off"],
["Vacuum 1 ok","diVacuum1OK",""],
["","",""],["","",""],["","",""]
];

PROC GripperTestVariant1
!Switch on gripper vacuum, no check of feedback
MT_GripSet gsVacuumOn,gdSucker1\NoCheck;
!Wait until gripper vacuum is switched on
MT_GripCheck gsVacuumOn,gdSucker1;
!Switch off gripper vacuum, no check of feedback
MT_GripSet gsVacuumOff,gdSucker1\NoCheck;
!Wait until gripper vacuum is switched off
MT_GripCheck gsVacuumOff,gdSucker1;
ENDPROC
PROC GripperTestVariant2
!Switch on gripper vacuum, check feedback
MT_GripSet gsVacuumOn,gdSucker1;
!Switch off gripper vacuum, check feedback
```

Continues on next page

```
MT_GripSet gsVacuumOff,gdSucker1;
ENDPROC

ENDMODULE
```

Sequences

In some areas of production, such as injection moulding, the robot grippers normally consist of several control elements, e.g. vacuum gripper, tongs, etc, which need to be connected one after the other and not simultaneously (e.g. swivel, switch on vacuum and close gripper).

For implementing this sequence of connections or switching operations, RWMT offers two options:

- Multiple use of the GripSet instruction one after the other
- Defining the control elements that are to be connected in an array of the data type grpsequence , which will be passed to an instruction for processing.

Example 1:

```
MODULE MT_Main

!gripper data of the individual control elements
const grpdata gdY1:=[…];
const grpdata gdY2:=[…];
const grpdata gdY3:=[…];
const grpdata gdY4:=[…];

!gripper sequence
const grpseq gsOpenSeq {3}:=
[[gsClose,gaSetAndCheck,"gdY1","","","","","",0],
[gsOpen, gaSetAndCheck,"gdY2","gdY3","", "","","","",0],
[gsClose, gaSetAndCheck,"gdY4”,"","", "","","",0]];

PROC SeqTest()
!Testing the gripper sequence
MT GripSequence\Sequence:=gsOpenSeq;
ENDPROC

ENDMODULE
```

Some gripper sequences for different part types might differ. In this case, the names of the gripper sequence declarations can be assigned an additional index. Those sequences can be called dynamically by means of the GripSequence instruction.

Example 2:

```
MODULE MT_Main

!Gripper data for the available actuators
const grpdata gdY1:=[…];
const grpdata gdY2:=[…];
const grpdata gdY3:=[…];
```

Continues on next page

6.3.8 Grippers
Continued

```
const grpdata gdY4:=[…];
const grpdata gdY5:=[…];
const grpdata gdY6:=[…];
const grpdata gdY7:=[…];

!Gripper sequence for part type 1
const grpseq gsOpen_T1 {3}:=
[[gsClose, gaSetAndCheck,"gdY1","","","","","",0],
[gsOpen,gaSetAndCheck,"gdY2",“gdY3“,"","","","",0],
[gsClose, gaSetAndCheck,"gdY4“,"","","","","",0]];


!Gripper sequence for part type 14
const grpseq gsOpen_T14 {3}:=
[[gsClose, gaSetAndCheck,"gdY4","","","","","",0],
[gsOpen, gaSetAndCheck,"gdY6","gdY7","","","","",0],
[gsClose, gaSetAndCheck,"gdY5","","","","","",0]];


PROC SeqTest()
!Test gripper sequence for a specific part, defined
!by its program number
MT_GripSequence \SeqName:="gsOpen_T"+ValtoStr(nProgNo);

ENDPROC

ENDMODULE
```

Part controls

With the help of the instruction MT PartCheck and the data type grppart, RWMT provides a means of testing the component control sensors on a gripper. If the sensor status does not correspond to the state which is to be checked, then, an error message is output after a waiting period.

The check can be queried, optionally, for **part controls busy** (high) or **part controls free** (low).

Example :

```
MODULE MT_Main

!gripper data of the individual control elements
const grppart gpPartControl:=
["Part control“,1,

[ "Sensor 1","diSensor1"],[ "Sensor5","diSensor2"]
[ "Sensor 2","diSensor3"] [ "Sensor6","diSensor4"]
[ "Sensor 3","diSensor5"] [ "Sensor7","diSensor6"]
[ "Sensor 4","diSensor7"] [ "Sensor8","diSensor8"]];

PROC SeqTest()
!Teilekontrolle mit Sensoren auf "high" durchführen
```

Continues on next page

```
MT PartCheck high, gpPartControl;
ENDPROC

ENDMODULE
```

A check is conducted to see if all the part sensors of the part controls of the gripper are busy.

Instead of labels like **Sensor 1**, the electrical identifier can be used which might be helpful for technical service staff.

Continuative example 1: One actuator and one part control

```
!
MoveJ *, vmax, fine, tGRP1;
!
!open actator 11 without monitoring (simultaneously to robot
     movement)
MT_GripSet gsOpen, gdGRP1_STG11\NoCheck;
!
MoveJ *, vmax, z200, tGRP1;
MoveL *, vmax, z50, tGRP1;
!
!Check if the required position has been reached
MT_GripCheck gsOpen, gdGRP1_STG11;
!
!Move to gripping position
MoveL pGrip, vmax, fine, tGRP1;
!
!Check part control 1 for condition high (part in the gripper)
MT_PartCheck high, gstGRP1_BT01;
!
!Grip part = Close actuator 11
MT_GripSet gsClose, gdGRP1_STG11;
!
MoveL *, vmax, z50,tGRP1;
MoveJ *, vmax, z200,tGRP1;
MoveL *, vmax, z50,tGRP1;
!
!Move to drop position
MoveL pDrop, vmax, fine, tGRP1;
!
!Drop part = Open actuator 11
MT_GripSet gsOpen, gdGRP1_STG11;
!
!Move away from drop position
MoveL *, vmax, z5, tGRP1;
!
!Check part control 1 for condition low (no part in the gripper)
     MT_PartCheck low, gpGRP1_BT01;
!
MoveJ *, vmax, z200, tGRP1;
MoveJ *, vmax, z200, tGRP1;
```

Continues on next page

6.3.8 Grippers
Continued

```
!
!Close actuator 11 (simultaneously to gripper movement)
MT_GripSet gsClose, gdGRP1_STG11\NoCheck;
!
MoveJ *, vmax, z5, tGRP1;
MoveJ *, vmax, z5, tGRP1;
```

Continuative example 2: Multiple actuators and part controls

The actuators 14 and 15 must be closed before using the actuators 11, 12 and 13 to grip a part since otherwise the clamps would collide with the machine in the grip position.

```
MoveJ *, vmax, fine, tGRP1;
!
!Open actuators 11, 12 and 13
!without monitoring
MT_GripSet gsOpen,gdGRP1_STG11\Grp2:=gdGRP1_STG12
\Grp3:=gdGRP1_STG13\NoCheck;
!
!Close actuators 14 and 15 without monitoring
MT_GrpSet gsClose,gdGRP1_STG14
\Grp2:=gdGRP1_STG15\NoCheck;
!
MoveJ *, vmax, z200, tGRP1;
MoveL *, vmax, z50, tGRP1;
!
!Check actuators 11, 12 and 13 for open state
MT_GripCheck gsOpen,gdGRP1_STG11
\Grp2:=gdGRP1_STG12\Grp3:=gdGRP1_STG13;
!Check actuators 14 and 15 for closed state
MT_GripCheck gsClose,gdGRP1_STG14
\Grp2:=gdGRP1_STG15;
!
!Move to gripping position (zone fine!)
MoveL pGrip, vmax, fine, tGRP1;
!
!Check part control 1,2,3,4,5 for condition “high“
!to make sure the part is located in the gripper
MT_PartCheck high,gpGRP1_BT01\Part2:=gpGRP1_BT02
\Part3:=gpGRP1_BT03
\Part4:=gpGRP1_BT04\Part5:=gpGRP1_BT05;
!
!Grip part = Close actuators 11 , 12 , 13
!and activate load for inner part
MT_GripSet gsClose,gdGRP1_STG11\Grp2:=gdGRP1_STG12
\Grp3:=gdGRP1_STG13\PartLoad:=loInnerPart;
!
MoveL *, vmax, z50, tGRP1;
MoveJ *, vmax, z200,tGRP1;
!
```

Continues on next page

```
!Move to drop position (zone fine!)
MoveL pDrop, vmax, fine, tGRP1;
!
!Drop part = Open actuators 11,12,13 and
!activate load0
MT_GripSet gsOpen, gdGRP1_STG11\Grp2:=gdGRP1_STG12
\Grp3:=gdGRP1_STG13\PartLoad:=load0;
!
!Leave drop position
MoveL *, vmax, z5, tGRP1;
!
!Check part control 1,2,2,4,5 for condition “low“
MT_PartCheck low, gpGRP1_BT01\Part2:=gpGRP1_BT02
\Part3:=gdGRP1_BT03\Part4:=gpGRP1_BT04
\Part5:=gpGRP1_BT05;
MoveJ *, vmax, z200, tGRP1;

!Open actuator 14 und 15 without monitoring
MT_GrpSet gsClose, gdGRP1_STG14\Grp2:= gdGRP1_STG15\NoCheck;
```

Continuative example 3: Gripping and dropping multiple parts

3 actuators, 3 part controls and 3 heavy parts, that are picked in a certain order and later dropped in the same order (part 1, 2, 3)

The following loaddata are to be used

Gripping part 1, load 1= loPart1

Gripping part 2, load 1 + 2 = loPart1_2

Gripping part 3, load 1,2 + 3 = loPart1_2_3

Dropping part 1 load 2+ 3 = loPart2_3

Dropping part 2 load 3 = loPart3

Dropping part 3 load 0 = load0

The `loaddata` must be identified by means of `LoadIdentify` for each combination.

```
! Grip part 1
!
!Move to gripping position of part 1
MoveL pGrip1, vmax, fine, tGRP1;
!
!Check part control 1 for condition “1“
!(part located in the gripper)
MT_PartCheck high, gpGRP1_BT01;
!
!Grip part 1 = Close actuator 11 and set
!load for part 1
MT_GripSet gsClose, gdGRP1_STG11
\PartLoad:=loPart1;
!
MoveL *, vmax, z50, tGRP1;
```

Continues on next page

6.3.8 Grippers
Continued

```
MoveJ *, vmax, z200, tGRP1;
!
! Grip part 2
!
!Move to gripping position of part 2
MoveL pGrip2, vmax, fine, tGRP1;
!
!Check part control 2 for condition “1“
!(part located in the gripper)
MT_PartCheck high, gpGRP1_BT02;
!
!Grip part 2 = Close actuator 12 and set
!load for part 1 + part 2
MT_GripSet gsClose, gdGRP1_STG12
\PartLoad:=loPart1_2;
!
MoveL *, vmax, z50, tGRP1;
MoveJ *, vmax, z200, tGRP1;
!
! Grip part 3
!
!Move to gripping position of part 3
MoveL pGrip3, vmax, fine, tGRP1;
!
!Check part control 3 for condition “1“
!(part located in the gripper)
MT_PartCheck high, gpGRP1_BT03;
!
!Grip part 3 = Close actuator 13 and set
!load for part 1, 2 and 3
MT_GripSet gsClose, gdGRP1_STG13
\PartLoad:=loPart1_2_3;
!
MoveJ *, vmax, z200, tGRP1;
MoveJ *, vmax, z200, tGRP1;
!
! Drop part 1
!
!Move to drop position 1
MoveL pDrop1, vmax, fine, tGRP1;
!
!Drop part 1 = Open actuator 11 and set
!load for part 2 and part 3
MT_GripSet gsOpen, gdGRP1_STG11
\PartLoad:=loPart2_3;
!
!Leave drop position
MoveL *, vmax, z5, tGRP1;
!
!Check part control 1 for condition “0“
MT_PartCheck low, gpGRP1_BT01;
```

Continues on next page

```
!
MoveJ *, vmax, z200, tGRP1;
MoveJ *, vmax, z200, tGRP1;
!
! Drop part 2
!
!Move to drop position 2
MoveL pDrop2, vmax, fine, tGRP1;
!
!Drop part 2 = Open actuator 12 and
!set load for part 3
MT_GripSet gsOpen, gdGRP1_STG12
\PartLoad:=loPart3;
!
!Leave drop position
MoveL *, vmax, z5, tGRP1;
!
!Check part control 2 for condition “0“
MT_PartCheck low, gpGRP1_BT02;
!
MoveJ *, vmax, z200, tGRP1;
MoveJ *, vmax, z200, tGRP1;
!
! Drop part 3
!
!Move to drop position 3
MoveL pDrop3, vmax, fine, tGRP1;
!
!Drop part 2 = Open actuator 13 and
!set load0
MT_GripSet gsOpen, gdGRP1_STG13
\PartLoad:= load0;
!
!Leave drop position
MoveL *, vmax, z5, tGRP1;
!
!Check part control 3 for condition “0“
MT_PartCheck low, gpGRP1_BT03;
!
MoveJ *, vmax, z200, tGRP1;
MoveJ *, vmax, z200, tGRP1;
```

Continues on next page

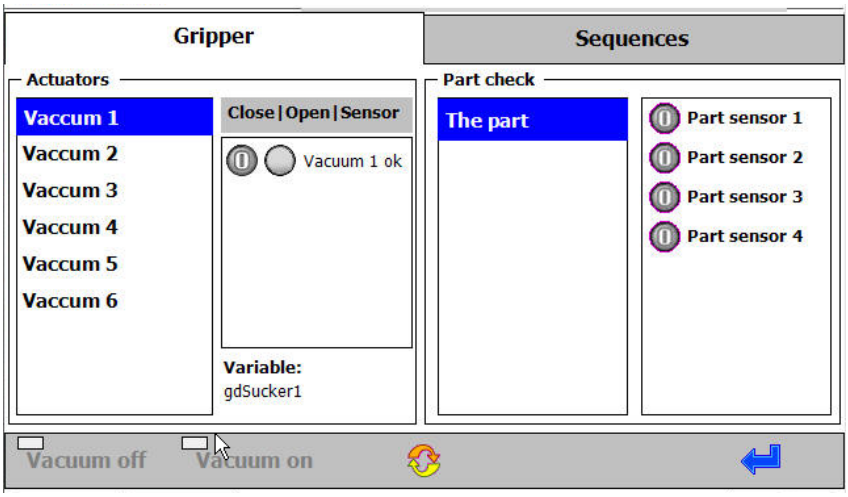
6.3.8 Grippers
Continued

Representation in the RWMT GUI

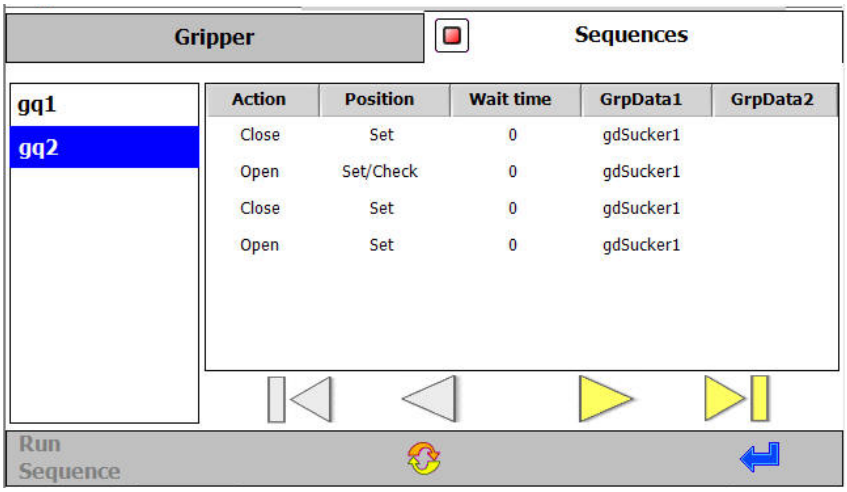
If data of the type `grpdata`, `grpsequence` or `grppart` is declared, then these declarations will be represented on the graphic user interface (GUI) in the form of list views for every data type.

The state of actuators (control elements) and of sensors can be checked through selection of a specific actuator in the list.

Besides this, it is also possible to actuate the control elements in the `Set up` mode of operation of the robot in the GUI.



en1200000758



en1300000062

6.3.9 Service routines

General

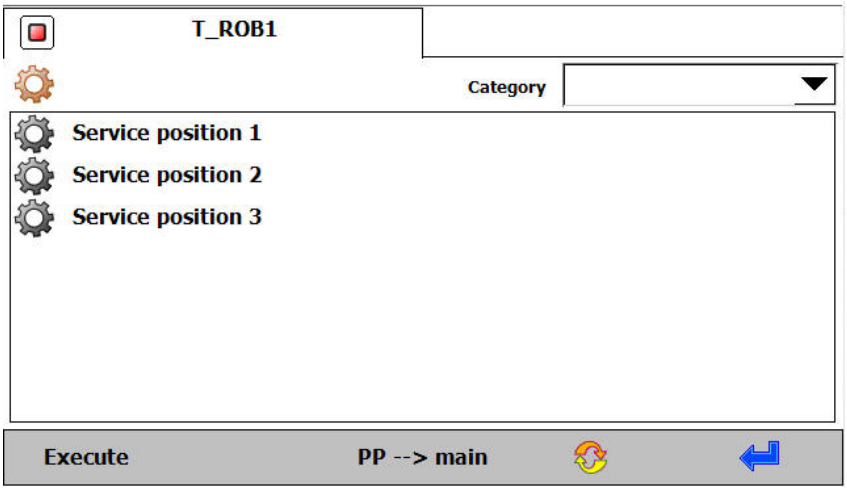
This chapter explains how set up routines and service routines can be set up or configured.

Service routines are used for service jobs and set up jobs and differ in the manner in which they are executed:

- Service routines of the Type I are executed instead of a production program.
- Service routines of the Type II are executed in parallel with the normal program run, hence the program pointer in the robot program is not changed.

In the service menu, all the menu declarations are displayed depending on the permissions of the user who has logged in.

If the robot is in the required mode of operation and the specified position, then, a service routine can be executed.



en1200000759

Differences between the service routine types

There are two types of service routines, which differ as follows:

Property	Type I	Type II
Will the service routine be executed in parallel with the stopped robot program?	No	Yes
Processing possible in the automatic mode?	Yes	No
Are interrupts processed?	Yes	No
Are event routines processed	Yes	No
Can the service routine be started through an external program number selection?	Yes	No

Based on the type of processing of the service routines. one has to decide the type that is to be used with the help of the requirement of the service routine that is to be required.

Example for the service routine type I:

Continues on next page

The robot should execute movements and should be in a position to move safely to the home position through the HomeRun function. Furthermore, if a processing program is interrupted, it should be continued afterwards.

Example for the service routine type II:

The robot should not execute any movements, or after cancellation, should be moved manually to the home position. The service routine should be executed through an interruption in the processing program. The processing program should be continued again after the service routine has ended.

Declaring a service routine

A service routine is defined through a menu declaration of the data type `menudata`.

Example:

```
CONST menudata mnuGreiferWechsel:=
["Change Gripper","Gripper","station-gripper.png",
     "GripperChange","",3.TRUE,1000,1,50];
```

Apart from the menu text, a category as well as an icon that will be displayed in front of the menu entry can be specified.

If a large number of service routines are used, then, the display of menu entries can be filtered with the help of the category, so that only the entries with identical category are displayed.

Position dependent release

If a service routine may be executed only if the robot is at a particular position, then the corresponding coding must be specified under `menudata.ValidPosition`. If several positions are allowed then the coding should be determined by addition (e.g. home position (`ValidPosition=1`) OR safe position (`ValidPosition=2`) gives the `ValidPosition=3`).

If the position of the robot cannot be checked then the value 255 should be specified.

Processing in the automatic mode

To be able to execute a service routine of the Type I in the automatic mode, `menudata.RunInAutoMode` must be set to `TRUE`, so that the menu entry is activated.

Execution by dialing the program number

Service routines of the Type I can be executed by dialing a program number (`menudata.ProgCode`). To do so, the number of the corresponding service routine is communicated instead of the program number for a processing program.

Note

To execute a service routine of the Type I, the cell mode of operation **Service** should be set.

Continues on next page

Defining the menu type

The `menudata.menu type` is set through the following numbers:

1 Service routine Type I
2 Service routine Type II

User controlled menu display

To prevent service routines from being called inadvertently, the display of menu entries can be controlled depending on the level of knowledge of the user who has logged in.

For this, the minimum user level (`menudata.MinUserLevel`) of the menu declaration is compared with the user level of the user who has logged in, and only those menu entries whose minimum user level is less than or equal to the user level will be displayed.

Additional information for service routines of type I

If the robot program was not running while launching a service routine, the robot program stops after the service routine is finished.

If the execution of a service routines is allowed only if the robot is located in home position, the robot program stops after the service routine is finished.

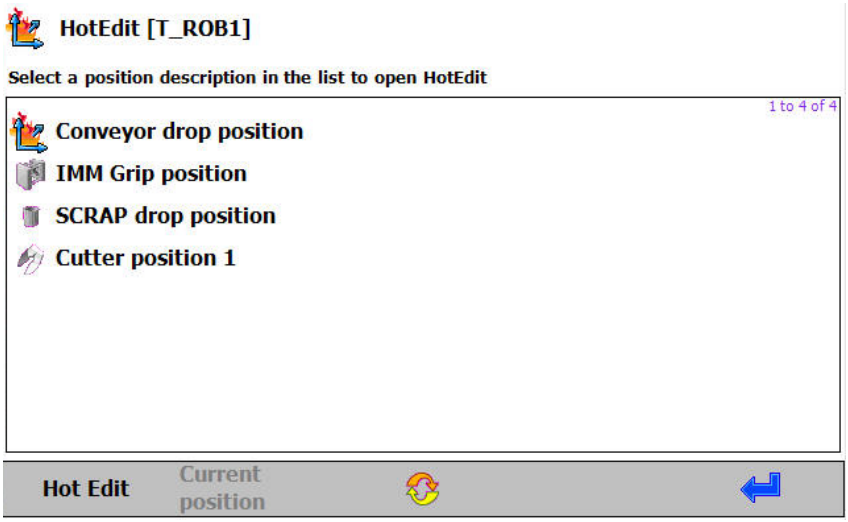
6.3.10 Advanced HotEdit

RWMT and HotEdit

In order to be able to change the robot positions during ongoing operations, the Standard HotEdit is used.

Since the difficulty here is that the desired positions cannot be selected quickly through a filter, the GUI is used as an intermediate step for selecting the positions.

With the help of array data declarations of the data type `hoteditdata`, the desired positions are saved along with their routine name and module name as well as a descriptive text and an icon (e.g. a station icon) and offered for selection in the HotEdit window of the GUI.



en1200000761

The aim of this pre-selection is to offer only those positions which belong to the part which is being processed currently, for selection. Furthermore, only a few positions will need to be modified always, as a rule (e.g. gripper positions and deposit positions).

Program concepts for pre-selection

Depending on the structure of the program, different options are available for creating a HotEdit declaration.

The following program structures are possible:

- No part type indexed movement routines and positions will be used (e.g. `PROC mv11_12`)
- For every type (part), a program module with local data and routines will be used. The program module is indexed in a type related manner, for example, `MODULE PART_4711`.
- Global, type specific movement routines and positions are used (for example, `PROC mv11_12_T4711`).

Continues on next page

Declaration of the pre-selection

In order to be able to represent these program structures in a HotEditData declaration, the following possibilities exist:

All the generally valid positions can be declared, for instance, in one or more arrays of the data type `hoteditdata`.

```
CONST hoteditdata hedGeneral{5}:=
[[„IMM Grip Position“,…],[…],[…],[…],[…]];
```

Type dependent declaration

Introduction

If the naming convention is the same for all the positions that are to be modified, then these differ only in terms of the type number or program number, hence, the wildcard **&** can be used for the type number and the wildcard **%** can be used for the program number in the declaration.

If the wildcard **&** is used and a part does not use a type code (-1), the program code will be used automatically.

Program module

A type dependent program module `MODULE PART_4711` is parameterized as follows:

`PART_4711`	Module name for type coding 4711.
`PART_&`	Module name for the current type coding, for example, `PART_14`, if the current type code is 14.
`PART_%`	Module name for the current program code, for example, `PART_5`, if the current program code is 5.

Routine name

A type dependent routine `PROC mv11_12_T4711` is parameterized as follows:

`mv11_12_T4711`	Name of the routine for type coding 4711.
`mv11_12_T&`	Name of the routine for current type coding, for example, `mv11_12_T8`, if the current type code is 8.
`mv11_12_T%`	Name of the routine for current program code, for example, `mv11_12_T3`, if the current program code is 3.

If we use only the placeholders:

- &	The module name is built by using the module name prefix of the section **RWMT part settings** of the process configuration, the type prefix of the section "RWMT part settings" of the process configuration and the current type code.
- %	The module name is built by using the module name prefix of the section **RWMT part settings** of the process configuration, the type prefix of the section "RWMT part settings" of the process configuration and the current program code.

Name of the position (robtraget)

A type dependent Position `robtarget p11_T4711` is parameterized as follows:

`p11_T4711`	Position name for type coding 4711.
`p11_T&`	Position name for current type coding, for example, `p11_T9`, if the current type code is 9.

Continues on next page

6.3.10 Advanced HotEdit
Continued

p11_T%	Position name for current program code, for example, p11_T5, if the current program code is 5.

Type dependent module, local data, and routines

```
CONST hoteditdata hedExample{1}:=
[[“IMM pick position“,“”,“PART_&“,“mv10_11“,“p11“,“”,“”]];
```

Type dependent data and routines

```
CONST hoteditdata hedExample{1}:=
[[“IMM pick position“,“”,“”,“mv10_11_T&“,“p11_T&“,“”,“”]];
```

Type dependent HotEdit array declaration

If different positions are changed in different types, and hence these cannot be represented by wildcards, then, the entire data declaration can be done in a type dependent manner.

This means that the names of the constants of the HotEditData array will receive at the end the tag or suffix **_T** with the corresponding type number or program number:

```
CONST hoteditdata hedIMM_T4711{1}:=
[[“IMM Greifposition“,“”,“”,“mv10_11_T&“,“p11_T&“,“”,“”]];
```

Note

The menu entries of the type dependent data declarations will be displayed only if the corresponding type has been selected.

Example

For the operations of an injection casting machine, the program is structured in such a way that all the movements of the robot are in type dependent program modules. The data declarations and routines are declared LOCALLY.

The naming of the unload position of the injection casting machine (PROC mv11_12), as well as the position of depositing on the conveyor belt (PROC mv30_32) is the same for all types, but differs only in terms of the module name.

The parts for the types 4711 and 815 must also be cut, whereby the number and the names of the positions is differentiated by the line (PROC mv20_21) (?).

=> data declaration for the generally valid position:

```
CONST hoteditdata hedGeneral{2}:=[
[“IMM grip position“,“station-IMM.png“,“PART_&“,
“mv10_11“,“p11“,“”,“”]
[“Drop position conveyor“,“station-conveyor.png“,“PART_&“,
“mv30_31“,“p31“,“”,“”]
];
```

=> type dependent declaration

```
CONST hoteditdata hedCutting_T4711{2}:=[
[“Cutting positions group 1“,“station-cutter.png“,“PART_&“,
“mv20_25“,“p21_P1,p21_P2,p21_P3“,“”,“”]
[“Cutting positions group 2“,“station-cutter.png“,“PART_&“,
“mv20_25“,“p22_P1,p22_P2,p22_P3“,“”,“”]
];
```

Continues on next page

```
CONST hoteditdata hedCutting_T815{2}:=[
[“Cutting positions group 1“,“station-cutter.png“,“PART_&“,
“mv20_25“,“p21_P1,p21_P2“,““,““],
[“Cutting positions group 2“,“station-cutter.png“,“PART_&“,
"mv20_25","p22_P1,p22_P2","",""],
[“Cutting positions group 3“,“station-cutter.png“,“PART_&“,
"mv20_25","p23_P1,p23_P2","",""]
];
```

6.3.11 External applications

General

Besides the *station-dependent application screens* (see *Station applications on page 60*), the *external application screens* can be launched in RWMT, that do not depend on a station, but have a more general purpose.

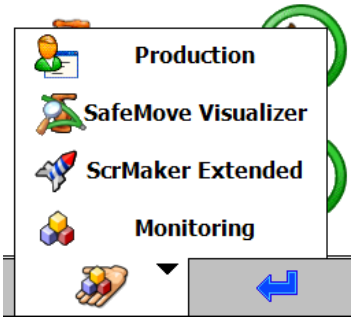
The purpose of these applications is to enlarge the possibilities to control or monitor general functionalities of the application, if the possibilities of RWMT do not totally fit the individual customer expectations.

As a difference to the station application screens, the external applications can be launched as embedded or non-embedded applications.

> **Note**
>
> Embedded means, that when having started such an application, the RWMT GUI view cannot be accessed until the application is closed. So each application must have a **Close** button, to be able to come back to RWMT.

For each application screen, customized images can be used for the menu representation in RWMT. Up to 8 different external applications can be applied to the production view.

In the following example, 4 external applications have been assigned. One of those applications is the standard production window of the FlexPendant, the other applications are customer screens.



en1300000063

The following standard views of the FlexPendant can be applied to the menu:

- Production
- Rapid Editor
- Rapid Data
- Backup & Restore

Defining external applications

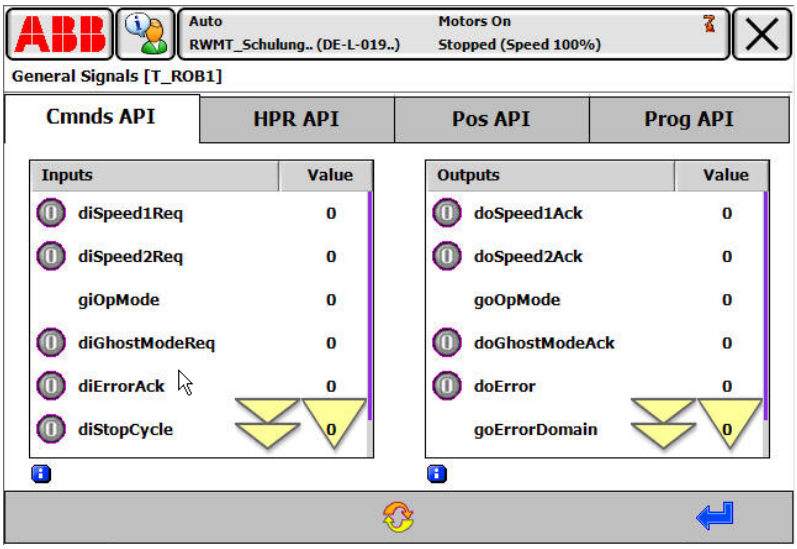
External applications have to be applied in the process configuration of RWMT in the section **MT Applications**. Please refer to chapter *MT Applications on page 163* to get detailed information, since there are various ways depending on the type of application.

6.4 General signal view

Usage

In the general signal view  , upto eight tab panes can be displayed, with digital inputs and outputs, group inputs and group outputs or analog inputs and outputs.

The outputs can be set or reset by the operator in the manual mode of operation. In the automatic mode, this is allowed only if the access level (EIO.CFG) of the respective signal has the corresponding permissions.



en1200000757

The configuration of a signal page is done with the help of the data types `signalpage` and could be declared as `CONST` or `LOCAL CONST`.

The name of the constant of the signal page will not be evaluated, it should be unique in the `TASK`.

The name of the tab pane is defined through `signalpage.PageName`. If several signal page declarations contain the same page name, then all the signals in a tab pane will be displayed, that is, more than 20 signals can be displayed per page in this way.

The position of the respective tab pane is defined by the `signalpage.PageIndex`.

If the number of pages present is unknown and if one wishes to display a page on the extreme right, then the `PageIndex` should be selected as greater than or equal to 8. If a page should always appear on the extreme left, then the `PageIndex` should be selected as less than or equal to 0.

If several signal page declarations contain the same `PageIndex`, then the sequence with the help of the page names.

Continues on next page

Example:

```
CONST signalpage spEUROMAP1:=["Euromap 67",1,
"diIMM_MouldOpen","diIMM_PusherRetracted", "diIMM_PusherForwarded",
"diIMM_Core1Retracted","diIMM_Core1Forwareded","diIMM_Scrap",
"diIMM_Automatic","diIMM_MouldClosed","diIMM_MouldIntermedPos",
"diIMM_Core2Retracted","diIMM_Core1Forwareded ","diIMM_Manufact1",
"","","","","","","","","];
```

Declaration of a signal page with the inputs from the EUROMAP 67-interface of an injection moulding machine.

6.5 Setup view

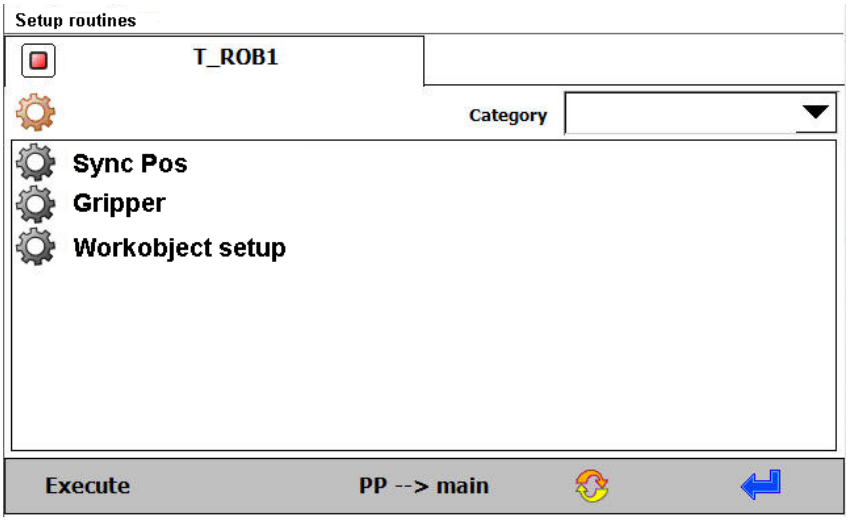
6.5.1 General

The **setup routines** in the setup view  are auxiliary routines that are used while setting up the robot at the time it is commissioned for operations (for example, measuring a work object, and so on).

This chapter explains how the setup routines can be configured.

In the setup menu all the menu declarations are displayed depending on the permissions of the user who has logged in.

If the robot is in the required mode of operation and the specified position, then, a setup routine can be executed by accessing the following view.



en1300000066

6.5.2 Declaring a setup routine

Introduction

A setup routine is defined through a menu declaration of the data type `menudata`.

Example:

```
CONST menudata mnuGripperChange:=
["Change Gripper","Gripper","station-gripper.png",
     "GripperChange","",3,TRUE,1000,3,50];
```

Apart from the menu text, a category as well as an icon that will be displayed in front of the menu entry can be specified.

If a large number of sertup routines are used, then, the display of menu entries can be filtered with the help of the category, so that only the entries with identical category are displayed.

Position dependent release

If a service routine may be executed only if the robot is at a particular position, then the corresponding coding must be specified under `menudata.ValidPosition`. If several positions are allowed then the coding should be determined by addition (e.g. home position (`ValidPosition=1`) OR safe position (`ValidPosition=2`) gives the `ValidPosition=3`).

If the position of the robot cannot be checked then the value 255 should be specified.

Processing in the automatic mode

To be able to execute a setup routine in the automatic mode, `menudata.RunInAutoMode` must be set to `TRUE`, so that the menu entry is activated.

Defining the menu type

The `menudata.menutype` menutype entry has to be set to the following number:

3: Setup routine

User controlled menu display

To prevent sertup routines from being called inadvertently, the display of menu entries can be controlled depending on the level of knowledge of the user who has logged in.

For this, the minimum user level (`menudata.MinUserLevel`) of the menu declaration is compared with the user level of the user who has logged in, and only those menu entries whose minimum user level is less than or equal to the user level will be displayed.

7 Event handling

Usage

Depending on the task, it may be necessary to execute routines in the case of certain program events or system events, which combine signals through the AliasIO instruction at the time of program start, for instance .

An example might be the initialization of program data or signals when starting the program.

Configuration

For simplification, all the program- and system events are made available through a RAPID declaration.

For this, in every module in which a routine is to be executed through a program- or system event, a declaration of the type `eventdata` has to be introduced.

An `eventdata` declaration consists of the following components:

Component	Explanation
Event	The event specifies the system event or program event for which the specified routine is to be executed.
Routine	The name of the routine contains a RAPID procedure without any transfer of parameters (for example, a procedure with the name `MY_PROCEDURE`), where even routines that are local to the module can be used. Here, the module name is written separated by means of a colon before the name of the routine (for example, `MY_MODULE:MY_PROCEDURE`).
Sortorder	The priority indicates the sequence in which these routines are called, for the same system event, that is, a routine with a `Priority` = 1 will be called before a routine with the `Priority` = 10.

Possible events

The system events and program events are defined through the following event numbers of the type `eventnum`:

Constant	Event No.	Event
`EE_POWERON`	1	Restarting the robot.
`EE_START`	2	Processing is started from the beginning of the program.
`EE_POWERON_OR_START`	3	Robot has been restarted or the processing starts from the beginning of the program.
`EE_RESTART`	4	Processing starts from the current position of the program pointer.
`EE_START_OR_RESTART`	5	Processing will be started from the beginning of the program or from the current position of the cursor.
`EE_STOP`	6	The program was stopped: • with the help of the stop key, • with the help of a stop instruction, • or with a stop after the current instruction.
`EE_QSTOP`	7	A quick stop of the robot has been executed (emergency stop).

Continues on next page

Continued

Constant	Event No.	Event
EE_STOP_OR_QSTOP	8	Program was paused by stop or emergency stop.
EE_RESET	9	A program has been closed or loaded. The event will not be triggered if a system module or a program module has been loaded.
EE_STEP	10	Step-wise execution of the program forwards or backwards.
EE_STEP_FWD	11	Step-wise execution of the program forwards.
EE_STEP_BCK	12	Step-wise execution of the program backwards.
EE_BEFORE_INIT	20	Program was started **from main** and the initialization is done in the next step.
EE_AFTER_INIT	21	One time initialization of the application program after start from main() and before a production routine is called. The robot is located in the home positionProgram was started **from main** and the initialization is done in the next step.
EE_WAIT_ORDER	22	Execution is done always if the robot is waiting for program number.
EE_BEFORE_PROD	23	Execution will be done before the start of the production cycle.
EE_AFTER_PROD	27	Execution will be done after the production cycle has ended and after the event EE_AFTER_PART.
EE_BEFORE_MENU	28	Execution will be done before the processing of a service routine.
EE_AFTER_MENU	30	Execution will be done after the processing of a service routine.
EE_ERROR	31	The event will be triggered if one of the error numbers ERR_MT_HOME (termination of current production cycle with immediate stop after cycle execution) or ERR_MT_ABORT (termination of current production cycle) are raised.
EE_HOMERUN	32	This event is triggered when starting the program from main and robot is not located in home position.
EE_PROG_END	33	Triggered, if the MT engine is left and the robot program is finished.
EE_AFTER_PROG_NUMBER	34	Triggered, if a program number could be successfully read.
EE_PROGNO_UNKNOWN	35	Triggered, if a program number has been read by RWMT but this program number neither matches a partdata nor a menudata declaration. The event can be used, for example, to load a module which contains the missing partdata or menudata declaration. After the execution of the event, RWMT verifies again, if the program number matches a partdata or menudata. If this is still not the case, the program execution is aborted with an appropriate message.

Continues on next page

Continued

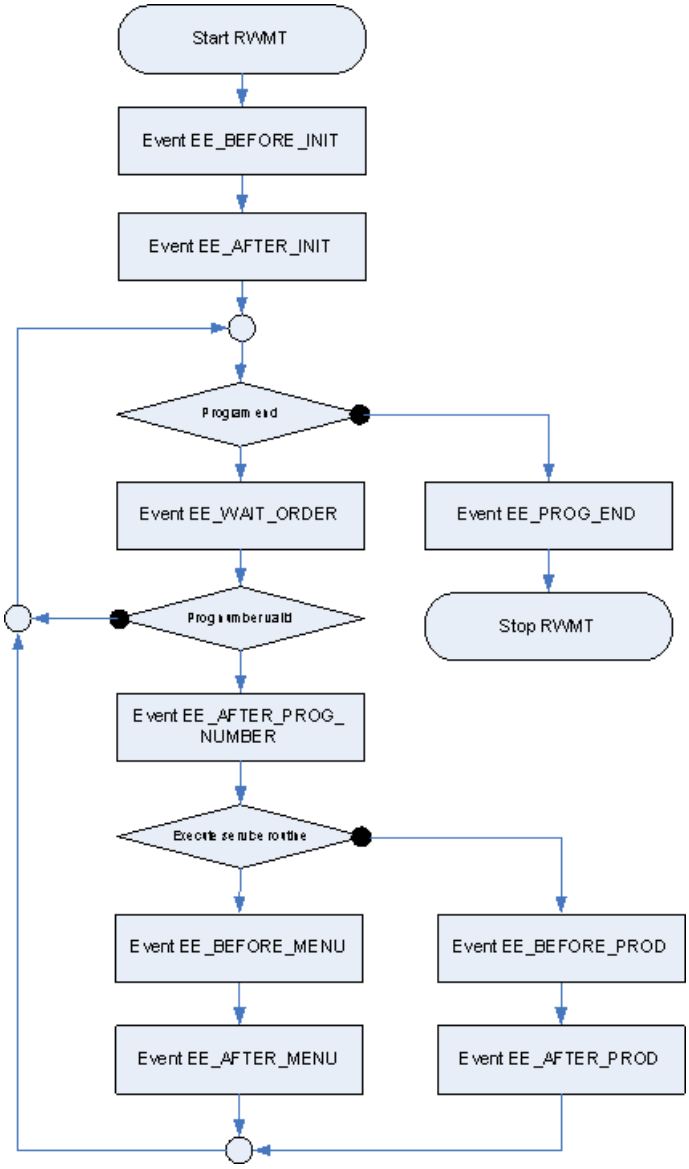
Constant	Event No.	Event
EE_PROD_UNKNOWN	36	Triggered, if a routine name has been specified in the <routine> item of a partdata declaration, which has been selected for production, but the specified routine does not exist in the program. The event can be used, for example, to load a module which contains the missing routine. After the execution of the event, RWMT verifies again, if the routine is available in the program. If the routine is still not available, the program execution is aborted with an appropriate message.
EE_BLOCKED	100	Triggered after having executed a setup routine in a task.

Continues on next page

Continued

Events in the context of program execution

The following flowchart shows some of the above listed events in the context of the program execution:



en1200000762

Continues on next page

Example

```
MODULE MT_Main()
…
!Routine call for the event "program start"
CONST eventdata
IMM_evStart:=[EE_START,"IMM:IMM_Initialize",1];
!Routine PowerOn to be called, when
!controller is restarted
CONST eventdata
IMM_evPowerOn:=[EE_POWERON,"PowerOn",1];
…
PROC Power On()
…
ENDPROC
…
ENDMODULE


MODULE IMM()
…
PROC IMM_Initialize()
…
ENDPROC
…
ENDMODULE
```

In this example the event declaration `IMM_evStart` is books an event of type `EE_START` and causes an event to be triggered, if the program is started from the beginning. It will call a routine `IMM_Initialize`, located in the module IMM.

This page is intentionally left blank

8 Instruction sets

Usage

Instruction Sets provide a means of setting specific variables or the digital output signals for a machine to a pre-defined value when entering or leaving the RWMT mode **Production** or when using the key switch to change the robot operation mode between **Manual** and **Automatic**.

A typical use case is the digital signal interface in injection moulding machines (EUROMAP). This interface expects the removal device to set its output signals to for the injection moulding machine to **high** as soon as the machine quits the **Production** mode of operation. It is only when these signals have been set that the injection moulding machine will be released for manual operations.

On the other hand, if the injection moulding machine is set back to the **Production** mode of operation, then the corresponding signals from the removal device must be reset again.

Since application program of the robot need not necessarily be running during such a change of mode, and hence cannot react on its own, Instruction Sets are a suitable means of implementing this requirement.

The change of RWMT operation mode can be alternatively requested through a group input or the graphical user interface.

In contrast, the change of the robot operation mode can only be done by using the key switch and confirming the respective dialogs on the FlexPendant.

Instruction Sets are available only if the robot system is equipped with the software option **Multitasking**.

Configuration of general instruction sets

Instruction Sets are data declarations of the type `instset` and are processed automatically by RWMT. If the robot system has been equipped with the software option **Multitasking**, then Instruction Sets will be available, otherwise not.

Instruction Sets that contain the string `RunWithRobot` in the name will be executed as soon as there is a change to the **Production** mode of operation of the cell.

The Instruction Sets that contain the `RunWithOutRobot` in the name will be executed as soon as the cell mode of operation **Production** changes to the cell mode of operation **Service** or **Without robot**.

An `instset` declaration for a station is created as a single dimensional array having the size 10. The name of the declaration must fit the following naming convention has to be considered.

Name	Explanation
<Stationdata.name>_RunWithRobot	`Station-dependent instset-` declaration name for changing to the **RWMT production** mode of operation.
<Stationdata.name>_RunWithoutRobot	`Station-dependent RWMT` declaration name for the change from the RWMT production mode of operation to the operation mode **Service** or **Without robot**.
<Stationdata.name>_OpModeManual	`Station-dependent instset-` declaration name for changing to the manual mode of the robot.

Continues on next page

Continued

Name	Explanation
<Stationdata.name>_OpModeAuto	Station-dependent instset- declaration name for changing to the automatic mode of the robot.

In addition to this, two general, station independent instset declarations can be created as single dimensional array having size 15. Here, the following declaration names have to be used:

Name	Explanation
MT_RunWithRobot	General instset- declaration name for changing to the RWMT production mode of operation
MT_RunWithoutRobot	General instset- declaration name for the change from the RWMT production mode of operation
MT_OpModeManual	General instset- declaration name for changing to the manual mode of the robot.
MT_OpModeAuto	General instset- declaration name for changing to the automatic mode of the robot.

Each declaration of type InstSet covers the following information:

Component	Explanation
Type	Here, the data type that is to be modified is specified. Digital and analog outputs, group outputs, data of the type num, dnum, string, bool are possible.
Data name	Declaration name of the data that is to be changed.
Value	Value to which the data should be set.
Task name	Name of the task that the data declaration contains.

Examples

Example 1

```
CONST InstSet IMM_RunWithRobot{10}:=[
["GO","goTest1","5",""],
["BOOL","bWithRobot",”TRUE”,""],
["NUM","reg1","1",""],
["STRING","string1","Start",""],
["","","",""],
["","","",""],
["","","",""],
["","","",""],
["","","",""],
["","","",""] ];
```

Example 2

```
CONST InstSet IMM_RunWithoutRobot{10}:=[
["GO","goTest1","0",""],
["BOOL","bWithRobot",”TRUE”,""],
["NUM","reg1","1",""],
["STRING","string1","Start",""],
["","","",""],
["","","",""],
["","","",""],
```

Continues on next page

```
["","","",""],
["","","",""],
["","","",""] ];
```

In the examples 1 and 2, a group output, a Boolean variable, a numerical variable, and a string-variable will be set to a specific value if the RWMT operation mode changes to production or leaves production.

Example 3

```
CONST InstSet MT_OpModeManual{15}:=[
["GO","goTest3","5",""],
["BOOL","bAutomatic",”FALSE”,""],
["NUM","reg2","7",""],
["STRING","string2","TestA",""],
["","","",""],["","","",""],["","","",""],["","","",""],
["","","",""],["","","",""],["","","",""],["","","",""],
["","","",""],["","","",""],["","","",""] ];
```

Example 4

```
CONST InstSet MT_OpModeAuto{15}:=[
["GO","goTest3","7",""],
["BOOL","bAutomatic",”TRUE”,""],
["NUM","reg2","9",""],
["STRING","string2","TestB",""],
["","","",""],["","","",""],["","","",""],["","","",""],
["","","",""],["","","",""],["","","",""],["","","",""],
["","","",""],["","","",""],["","","",""] ];
```

In the examples 3 and 4, a group output, a Boolean variable, a numerical variable, and a string-variable will be set to a specific value if the operation mode of the robot changes (key switch).

This page is intentionally left blank

9 RAPID Library

What is the RAPID-Library

The RAPID-Library is collection of instructions and functions, which extend the scope of the programming language RAPID.

This is related to the following areas:

- Message output
- Waiting for signals
- Movement functionalities

Contents

The following list is a compilation of all the instructions and functions of the RAPID-Library in RWMT. Details can be obtained from the chapter *RAPID references on page 245*.

Routine	Explanation
MT_UIMessage	instruction for outputting messages on the FlexPendant.
MT_WaitMsgGI	Instruction to wait for a specific state of a group input with message output. A message is shown, if the required state is not present after a while.
MT_WaitMsgGI32	Instruction to wait for a specific state of a 32-bit group input with message output. A message is shown, if the required state is not present after a while.
MT_WaitMsgDI	Instruction to wait for the high- or low- state of a digital input with message output. A message is shown, if the required state is not present after a while.
MT_WaitMsgGO	Instruction to wait for a specific state of a group output with message output. A message is shown, if the required state is not present after a while.
MT_WaitMsgGO32	Instruction to wait for a specific state of a 32-bit group output with message output. A message is shown, if the required state is not present after a while.
MT_WaitMsgDO	Instruction to wait for the high or low state of a digital output with message output. A message is shown, if the required state is not present after a while.
MT_ToolCheckL	Movement instruction with tool check.
MT_JointCompare	Function for comparing the angles of the robot axis of a given position with those of the current position.
MT_RecalcPoint	Function for converting a position on a new work object and a new tool.
MT_RelTCP	Function for moving or reorienting a TCP with respect to the current tool.
MT_PosCompare	Function for comparison of positions.

Continues on next page

Continued

Routine	Explanation
`MT MoveTo`	Instruction for the dynamic call (loading) of a movement routine. Requires compliance with the naming conventions.
`MT SetActual Position`	Instruction for setting the start position for the `MT MoveTo` instruction.
`MT GetActual Position`	Function for querying the start position for the `MT MoveTo` instruction.

10 HomeRun

10.1 Introduction

10.1.1 Overview

During the program execution of the robot, situations in which the robot program can not or may not be continued can arise.

For example:

```
A peripheral station has failed (fault)
Release signals are missing
Emergency off (program continuation is not possible)
The robot has collided with peripheral parts (crash)
```

If the occurring error cannot be rectified immediately, then the robot program must be cancelled and the robot moved into its home position. The error must then be rectified and the program restarted from **MAIN**.

If the robot program offers no possibility of moving the robot automatically into the home position, then the plant operator must move the robot with the aid of the joystick. If the plant operator is not familiar with handling the robot, then the standstill time of the overall plant is increased considerably by this. The occurring fault itself is rectified quickly as a rule.

Many faults occurring during the course of a program can be rectified in the program with the aid of error handling. For example, if the robot is waiting for a release signal from the periphery, then it can be moved selectively into its home position by an operating request with the aid of error handling.

When error handling is selected the momentary position of the robot is known, so that the robot can be moved into the home position without any collisions.

However, error handling offers no solution if an **emergency stop** or a **collision** which stops the robot during a movement has occurred or the robot program was restarted. In these situations the robot is not in a defined position as a rule, but is on a movement path from which the program must be restarted by **Start from Main**.

So that the robot can also move automatically into the home position in these cases, the `HomeRun` functionality is used.

This offers the possibility of moving the robot after a restart of the robot program from any automatically moved to position into the home position using existing movement routines. If a part is located in the gripper, then it can be set down in a defined manner at an arbitrary position.

10.1.2 HomeRun functionalities

HomeRun provides the following functionality:

- Simple programming of the movement sequence as in the standard baseware.
- Moving into the home position from every automatically moved to position.
- Position-related HomeRun strategy.
- Use of the movement routines available in the robot program.
- Backwards execution of movement routines (moving from the destination point to the start point).
- Usable in **MultiMove-Independent** robot systems

10.1.3 Method of operation

The following concept is used so that the robot can be moved to the home position from every position that it has automatically moved to.

- When a position is reached, a numeric value is stored in the robot program that clearly identifies this position.
- When moving into the home position, the required movement routines which move the robot to the home position in a defined manner are called on the basis of the last position value that was reached. When this occurs, no robot movements are made until the position value is found at which the robot program was interrupted.

Note

For this purpose all robot movements must be executed with special movement instructions which also store the robot position as a numeric value (that is, `MT_MoveL, MT_MoveJ, MT_MoveLDO, MT_TriggJ`, and so on).

In order to simplify the programming effort for the automatic movement into the home position, the robot movements that have already been created are used.

Note

However this is possible only if the movement and administration routines are strictly separated, that is, the robot movement instructions are in their own routines and are called up by the administration routines.

The strategy to be used with which the robot is moved into the home position depending upon the stored position number must be produced depending on the application.

10.2 First steps

10.2.1 Example program

Home Run programming is explained on the basis of the following example program.

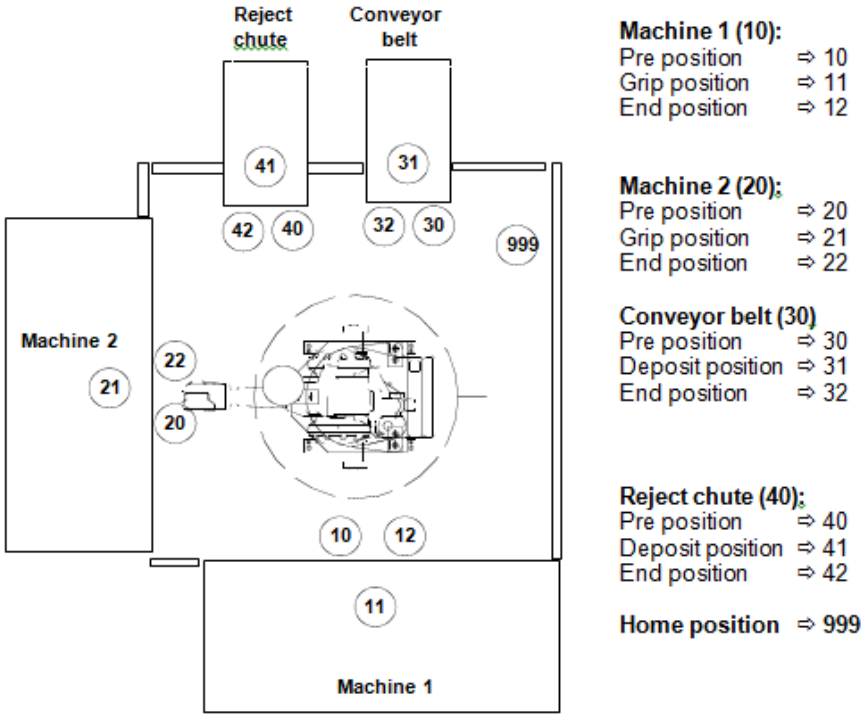
In a plant a robot must in each case take a part from machine 1 or machine 2 and place it on a delivery conveyor. Reject parts are placed on the reject chute.

Each station that the robot has to move to is given a two-digit or three-digit station number, e.g. machine 1 ->no. 10, machine 2 ->no. 20, deposit belt ->no. 30, reject chute ->no. 40.

The home position always allocated the number **999.**

The robot movements are divided up into different segments, i.e. the start and end points of a movement are given a position number, which is derived from the station number and can be found again later in the designation of the individual movement routines.

Position layout:



en1200000763

10.2.2 What is the Home Position?

The Home Position is the position in which the robot has to be at the start of program execution (**start from main**) or the position that the robot moves to at the end of the program.

This ensures that the robot always starts from a defined position and collisions with the system peripherals are therefore avoided.

The Home Position is selected taking the following conditions into consideration:

- The robot is outside the interference range of all machinery.
- The robot does not obstruct the machine operator when the system is being operated in manual.
- The robot does not obstruct the service personnel if maintenance or repairs are being carried out on the machines.

10.2.3 Movement routines

All robot movement instructions are stored in movement routines. The name of the movement routine always contains the start and the destination point of a movement and can also contain a type number for differentiating several types.

```
mv<Startpoint>_<Destinationpoint> [_<TypeNo>];
```

Examples:

`mv10_11`	Moving from the machine 1 pre position to the grip position Machine 1.
`mv20_99`	Moving from the machine 2 pre position to the home position.
`mv12_30_T3`	Moving from the machine 1 destination position to the pre position of the conveyor for type number 3.

The complete movement between the start and destination point is programmed in the movement routines, whereby the first point (start point) must always be moved to with the argument `\NoMove`. The use of instructions other than `MT_MoveX`, `MT_MoveXDO`, `MT_MoveXSync`, `STriggX`, and so on, is not allowed.

```
PROC mv10_11()
!From: Pre position Machine 1
!To : Grip position Machine 1
!IM-Pos: 1011xx
MT_MoveJ 10,p10,v200,z10,tGripper\NoMove;
MT_MoveJ 101101,*,v800,z10,tGripper;
MT_MoveJ 101102,*,v800,z10,tGripper;
MT_MoveL 11,p11,v500,fine,tGripper;

ENDPROC
```

10.2.4 Administration routines

For each station of the system an administration routine is created which contains the program sequence of the required function, including the movement routine calls (for example, unload machine 1).

Example:

```
PROC Maschine1_Unload()
OpenGripper;
WaitDI diUnloadMachine1,high;
Reset doIRBausM1;
mv10_11;
CloseGripper;
mv11_12;
Set doStartM1;
ENDPROC
```

10.2.5 Calling up the HomeRun movement

In order to ensure that the robot always starts from the home position after a program restart, this is automatically checked by RWMT using the **HomeRun**.

10.2.6 Routines in the MT_MAIN module

The following routines in the MT_MAIN module require application-related adaptation for the move to the home position:

Routine	Description
MT_HomeDirect	Direct movement to the home position.
MT_SpeedUpdate	Adapt speed before carrying out a movement instruction.
MT_HomeRun	Position related definition of how the robot should move to the home position.

10.2.7 Creating the HomeRun strategy

The strategy for moving the robot into the home position is stored in the `MT_HomeRun` routine

For each start or end position of the individual stations it is defined how the robot has to behave, i.e. of the gripper has to be actuated, signal communication is required with the machine as to which the robot should move to next.

Only the robot movement to the next position is programmed, since the `MT_HomeRun` routine is called with the current position number until the robot has reached the home position.

Example:

```
PROC MT_HomeRun(num Position)
!Call up movement routines of the robot
!depending on the position number
TEST Position
CASE 10:
MT_MoveRoutine "mv999_10";
CASE 11:
CloseGripper;
mv11_12;
CASE 12:
mv12_40;
CASE 20:
MT_MoveRoutine "mv99_20";
CASE 21:
CloseGripper;
mv21_22;
CASE 22:
mv22_40;
CASE 30:
mv30_40;
CASE 31:
OpenGripper;
mv31_32;
CASE 32:
mv32_999;
CASE 40:
!Wait until reject chute is ready
WaitDI diNOK_Free,high;
mv40_41;
OpenGripper;
mv41_42;
SET doNOK_Load;
CASE 41:
OpenGripper;
mv41_42;
SET doNOK_Load;
CASE 42:
mv42_99;
```

Continues on next page

```
DEFAULT:
!Automatic continuation of intermediate positions
MT_ContHomeRun Position;
ENDTEST
ENDPROC
```

10.2.8 Creating the HomeRun description

HomeRun provides an *operator dialog* (see *Operator dialogue for the HomeRun on page 124*) to show where the robot is currently located. The location is specified by the position number, used by the `MT_Move` instructions and by a descriptive text.

This text has to be provided by the integrator as shown in the chapter *posname – Assigning position description for HomeRun on page 287*.

10.2.9 Checking the Home position

During the execution of **HomeRun**, it is checked whether the robot is in the home position. If not, the robot is moved to its home position using the HomeRun strategy.

Home position checking can be carried out either by the robot system or by means of an external sensor. The most suitable check must be used depending on the purpose of use.

Type of monitoring	Purpose of use
External sensor (Digital input)	A sensor (for example, limit switch) is used if the position signal from the robot is also needed in a higher-order controller when the robot controller is switched off.
World zones	The robot controller provides a facility for monitoring robot positions with the aid of stationary or temporary world zones. If the robot is inside a world zone, a digital output can be set (see *RAPID Reference Manual – RAPID Overview* listed in the section *References on page 11*). A world zone can be used if the home position is also needed in an external controller or in the robot system.
-> Stationary world zone (Digital output)	The home position must be monitored via a stationary world zone if the home position is signalled to a higher-order controller and the robot program is not executed to do this (for example, operation without robot) or the output is used within a cross-connection or in a background task. To setup a stationary world zone, see *RAPID Reference Manual – Instructions, Functions, Datatypes* listed in the section *References on page 11*.
-> Temporary world zone (Digital output)	The use of a temporary world zone is already integrated in `HomeRun`, the use of which has to be activated in the system parameters (see parameter `CreateWZone` in the chapter *MT HomeRun on page 165*).
Internal check	If the position signal for the home position is only needed by **HomeRun**, the check can also be carried out by means of an internal position comparison between the home position and the current robot position. In order to be able to use the internal check, no signal name must be assigned to parameter `DIO_At_Home` in the system parameters.

WARNING

If an external sensor (switch) is being used, the robot position is not checked - it is checked whether the switch has been operated.

In this case there is a risk of a collision when the robot program starts, if:

- the robot is not in the home position and the switch is operated anyway.
- the robot has been moved with erroneous orientation in relation to the switch using the joystick (for example, 6th axis rotated by 360°).

10.2.10 Setting up the system parameters

The settings regarding the behaviour of HomeRun and the signals for monitoring and requesting the move to the home position are set in the HomeRun process parameters (see the chapter *MT_HomeRun – HomeRun Strategy on page 357*).

10.2.11 Signal combinations for HomeRun with stopped program

In order to be able to move a robot, whose program has been stopped, automatically into the home position with the **HomeRun** request, a program start from **Main** must be triggered.

This can be achieved by means of logical signal combinations within the robot controller at system level.

Example:

Type	Signal name	Description
DI	diIRBgoHome	HomeRun request
DI	diHR_MotorOn	**Switch on motors** request (System input: **MotorOn**)
DI	diHR_StartMain	**Start program from main** request (System input: **StartMain**)
DI	diHR_RobotsInHome	All robots are in their home positions (Cross-connection of all existing doRobXinHome outputs)
DO	doRob1inHome	Robot 1 is in the home position (for example, world zone)
DO	doRob2inHome	Robot 2 is in the home position
DO	doRob3inHome	Robot 3 is in the home position
DO	doRob4inHome	Robot 4 is in the home position
DO	doHR_CycleOn	Program is executed (System output: **CycleOn**)
DO	doHR_CycleOn	Motors are switched on (System output: **MotOnState**)

Tip

The signals described above are added to the robot system during first installation, and can be adapted as required!

Switch on motors of the robot

```
-Act1 "diIRBgoHome" -Oper1 "AND" -Act2 "doHR_MotorOn" -Act2_invert\
-Res "diHR_MotorOn"
```

Start robot programs from Main

```
-Act1 "doIRB1inHome" -Res "diHR_RobotsInHome"
-Act1 "diIRBgoHome" -Oper1 "AND"
-Act2 "diHR_RobotsInHome" -Act2_invert -Oper2 "AND"\
-Act3 "doHR_CycleOn" -Act3_invert -Oper3 "AND"\
-Act4 "doHR_MotorOn" -Res "diHR_StartMain"
```

10.2.12 Checking the position numbers

The use of the correct position numbers for the start, end and intermediate positions in the movement routines is decisive for moving the robot to the home position without collisions.

For example, when programming robot positions it may be that one or more movement instructions are copied from another movement routine. If the intermediate position numbers are not adapted to the name of the new movement routine, the wrong movement routine is called up during HomeRun, which may result in the robot colliding with the peripherals.

The following errors can occur during position number allocation:

The length of the position number if wrong.

```
PROC mv100_110()
MT_MoveJ 1000,p10,v100,z10,tGripper\NoMove;
MT_MoveL 100110010,*,v1000,z10,tGripper;
MT_MoveL 10011002,*,v1000,z10,tGripper;
MT_MoveL 1,p11,v500,fine,tGripper;
ENDPROC
```

The start and end points of the movements are not constituents of the intermediate position:

```
PROC mv100_110()
MT_MoveJ 100,p10,v100,z10,tGripper\NoMove;
MT_MoveL 11012001,*,v1000,z10,tGripper;
MT_MoveL 12010002,*,v1000,z10,tGripper;
MT_MoveL 110,p11,v500,fine,tGripper;
ENDPROC
```

The first position of a movement routine contains an intermediate position number.

```
PROC mv100_110()
MT_MoveL 10011001,*,v1000,z10,tGripper;
MT_MoveL 10011002,*,v1000,z10,tGripper;
MT_MoveL 110,p11,v500,fine,tGripper;
ENDPROC
```

The last position of a movement routine contains an intermediate position number.

```
PROC mv100_110()
MT_MoveJ 100,p10,v100,z10,tGripper\NoMove;
MT_MoveL 10011001,*,v1000,z10,tGripper;
MT_MoveL 10011002,*,v1000,z10,tGripper;
ENDPROC
```

The first position of a movement routine does not correspond to the last position number of the previously executed movement routine.

a) Wrong movement routine calling order:

```
mv100_110;
mv120_125;
```

b) First movement routine position number wrong:

```
mv100_110;
mv110_120;
PROC mv100_110()
```

Continues on next page

```
MT_MoveJ 100,p10,v100,z10,tGripper\NoMove;
MT_MoveL 110,p11,v500,fine,tGripper;
ENDPROC

PROC mv110_120()
MT_MoveJ 100,p10,v100,z10,tGripper\NoMove;
MT_MoveL 120,p11,v500,fine,tGripper;
ENDPROC
```

10.2.13 Checking the HomeRun strategy

As soon as the robot program has been started up, the HomeRun strategy should be tested for freedom from errors.

This should be done by running the production program in manual mode, and triggering the HomeRun at the required location.

It should normally only be necessary to test the start and end positions of a movement, since the movement is usually automatically continued to the next end position in the event of intermediate positions. The intermediate position number must consist of the start and end position of the movement routine when this occurs.

Check the strategy at the individual positions as to whether all signal handshakes and the required functions are executed correctly.

10.3 Use of HomeRun in RobotStudio

10.3.1 Behaviour of HomeRun in a virtual controller

In order to able to use HomeRun in the virtual controller of RobotStudio, attention must be paid to the characteristics described in the following.

If HomeRun is used in a virtual controller (RobotStudio), the following differences exist compared to an actual controller:

- Home position checking takes place on the basis of a position comparison, bypassing the defined signals (world zone and input).
- The search instruction `MT_SearchL` always moves until the end of the programmed position, without taking the search signal into consideration.
- If the home position has to be moved to via a direct route, the dialogue for switching the operating mode is not displayed, and the robot starts the movement to the home position immediately.
- If `HomeRun` is executed and the robot is not in the home position, the dialogue for starting the home run appears for 10 seconds. If none of the buttons are pressed during this time, the home run is started automatically.

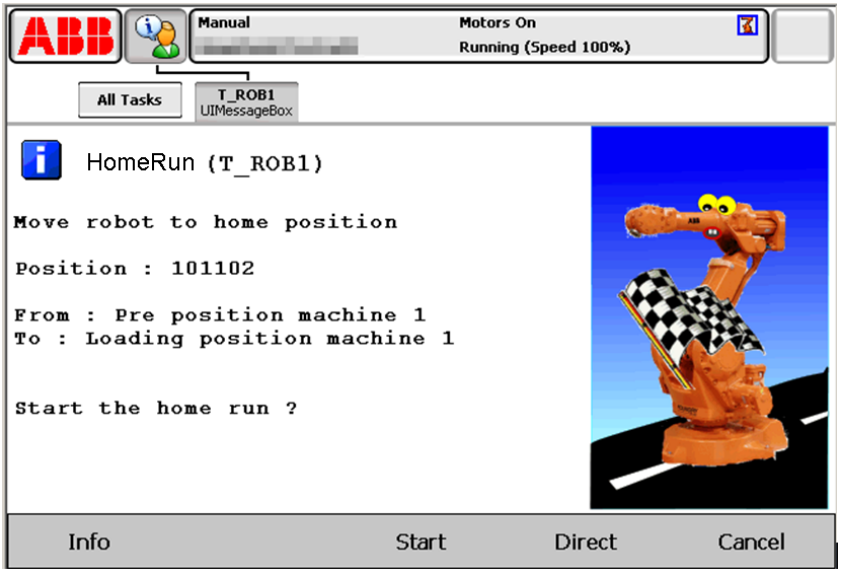
10.4 Operator dialogue for the HomeRun

10.4.1 Moving the robot automatically into the home position

If the robot lacks peripheral signals in order to continue the program, it can be forced to abort the currently active processing and move to the home position through the **HomeRun** request in combination with a restart of the robot program (start from main), error handling or an interrupt (ExitCycle).

So that the robot can move automatically from its last position to the home position, it may not have been moved with the joystick or only moved within the programmed start area (default = 150 mm) from the last position that was moved to. If it was moved out from the start area, then the **HomeRun** functionality is deactivated and only the direct movement to the home position is possible (see the following chapters).

If the robot can be moved automatically into the home position, then the motion is executed in the home position if the **HomeRun** request has been set. If this is not the case, the following menu is displayed on the programming unit.



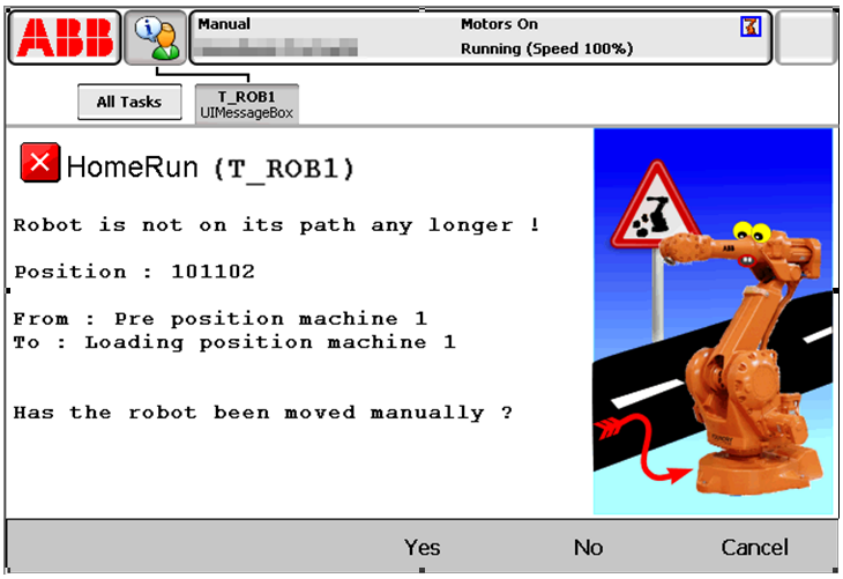
en1200000767

The current robot position and the motion direction (from start position to target position) are specified in this menu. The system operator can choose between the following possibilities:

Info	Displays theversion of **HomeRun**.
Start	The robot moves automatically on its programmed paths from the displayed start position to the home position. If there are parts in the gripper, they are deposited if necessary.
Direct	Menu for moving the robot directly to the home position (in **Manual** mode only).
Cancel	Program execution is cancelled.

10.4.2 Moving the robot semi-automatically into the Home position

The following menu appears if the robot has been moved by joystick out of the start area or a system status exists that prevents the robot from moving automatically to the home position.



en1200000768

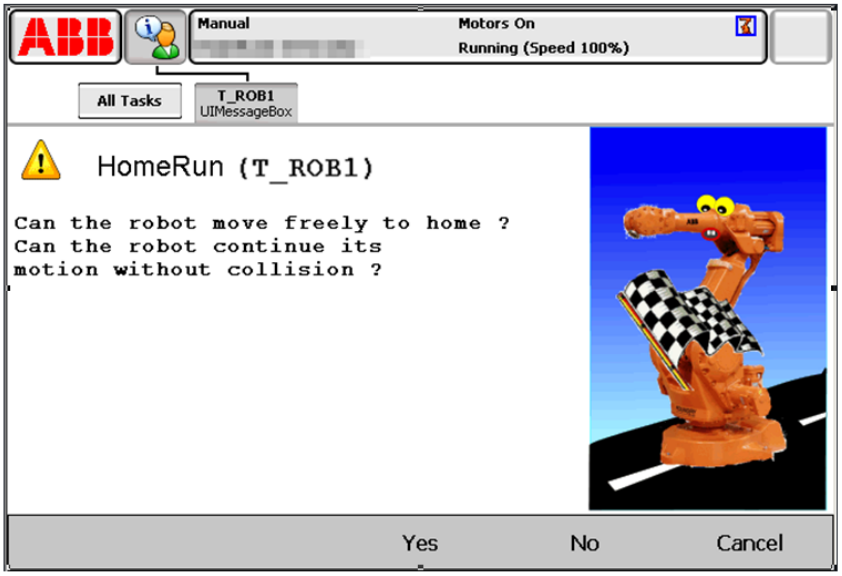
The last valid position is shown together with notification that the robot is not on its path where it should be. The system operator must specify in the dialogue whether the robot is till in the displayed position.

If the robot has been moved manually and is no longer in the displayed position, the **Yes** button must be pressed. The dialogue is continued in accordance with chapter *Moving the robot manually into the home position on page 127*.

Continues on next page

10.4.2 Moving the robot semi-automatically into the Home position
Continued

If the robot is in the displayed position, the **No** button must be pressed. In the dialogue that now appears, the system operator must specify whether the automatic home run can be carried out without a collision.



en1200000769

The robot starts the **HomeRun** as soon as the **Yes** button has been pressed. If the **No** button is pressed, the dialogue is continued as described in the chapter *Moving the robot manually into the home position on page 127*.

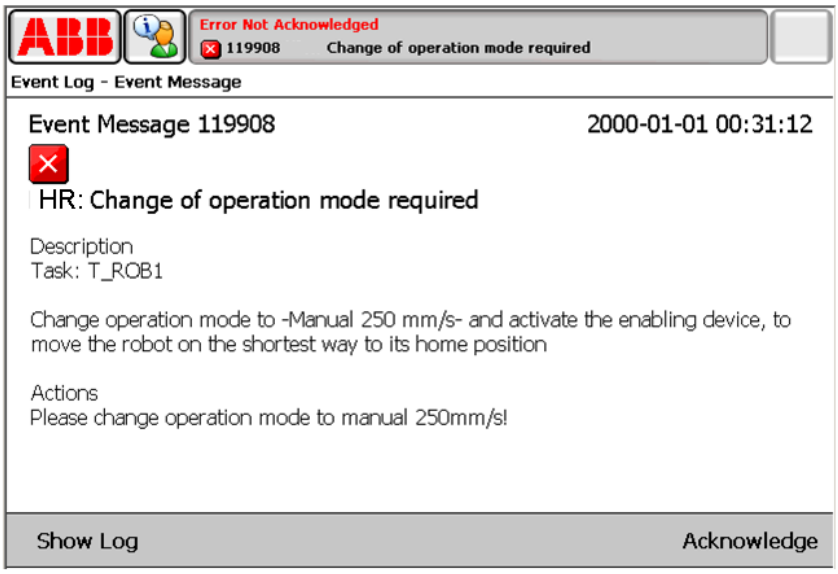
10.4.3 Moving the robot manually into the home position

If the robot has been moved using the joystick or it cannot be moved automatically from the last position that was moved to into the home position, the following user dialogue is displayed.

Note

In order to prevent damage to the robot or parts of the system during the movement to the home position, the robot controller must be switched into manual mode so that the system operator has full control of the robot movement.

If the robot is in automatic mode, the system operator is requested to switch to manual mode.

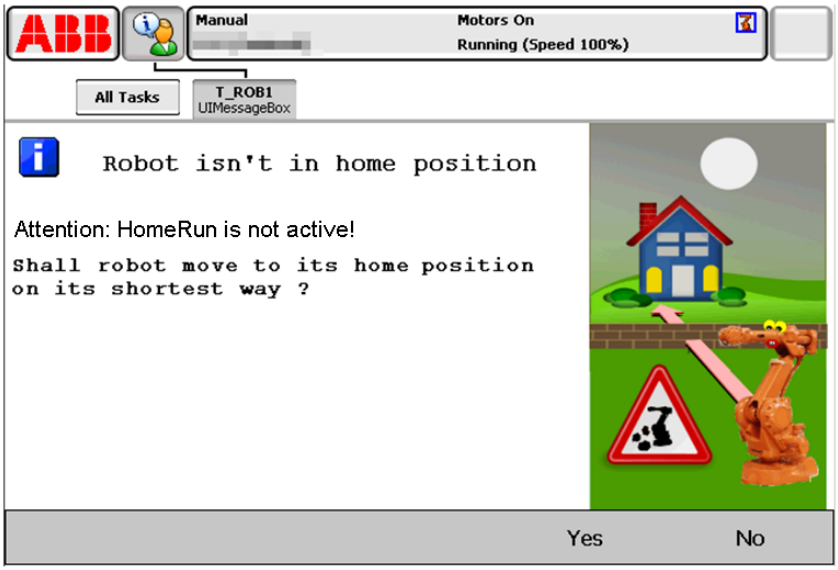


en1200000770

Continues on next page

10.4.3 Moving the robot manually into the home position
Continued

After switching to manual mode and restarting the program, the system operator is notified that the automatic home run is no longer active and the robot can be moved to the home position directly via the shortest route.



en1200000771

If the **YES** button is pressed, the robot moves to the home position directly.

WARNING

If the robot is moved to its home position using the shortest way, it must not be in the vicinity of disturbing contours since it moves directly to the home position and collisions cannot be ruled out.

The responsibility for collision-free movement lies in this case with the operator!

If the robot is in the vicinity of interfering contours, it must first be moved into a free area of the system using the joystick. In order to do this, the **NO** button must be pressed in the menu shown above.

After moving the robot into a free area of the system using the joystick, the program must be restarted in manual mode and the menu described previously is displayed again. The robot must now be moved directly to the home position without collisions.

Tip

Generally, the manual movement of the robot should be avoided as long as the automatic or semi-automatic movement can be used.

10.5 Programming the HomeRun

10.5.1 General

In order to allow the robot to be moved into the home position from any position that has been moved to automatically, all robot movements must be programmed with the instructions `MT_MoveX`, `MT_MoveXDO`, `MT_MoveXGO`, `MT_MoveXSync`, `MT_TriggX`, `MT_SearchL` , and so on. The **X** stands for the linear (**L**) or axis-related (**J**) movement type. Please refer to the chapter *Instructions on page 307* to find all available motion instructions.

With these instructions, a position number is transferred in addition to the movement parameters, reproducing the exact position of the robot in the plant.

The movement into the home position is executed by RWMT, if the application program is started from **Main**. Since the robot orients itself to the position number at which the program was stopped, the robot may not have been moved manually or only a small distance (risk of collision). If the robot was moved a long distance (>150 mm per default, start area), then the `HomeRun` functionality is disabled and the home position can only be moved to directly.

The strategy to be used to move the robot from a start/destination position to the home position must be programmed by the programmer application-related in the `MT_HomeRun` routine.

If the robot is at an intermediate position, then the robot movement is continued by renewed selection of the movement routine at the place at which it was interrupted. The movement can be continued at an intermediate position automatically, without explicitly calling up the movement routine, by using the `MT_ContHomeRun` instruction.

10.5.2 Allocation of the position designations

The position designations are responsible for determining the robot in the working area and may be allocated only once for one position. To simplify the teaching of the movement paths, notes about the different position numbers can be filed in the comment lines of the movement routines.

The position designations for the instructions `MT_MoveX`, `MT_MoveXDO`, `MT_MoveXGO`, `MT_MoveXSync`, `MT_TriggX`, `MT_SearchL`, and so on must be formed with the aid of the rules described below.

Rules for allocating position numbers

- The first and the last position in a movement routine are represented as a two-digit or three-digit number that corresponds to the position number. (for example, 10, 20, 30, 100, 210, , and so on.).
- The intermediate positions are identified as a six, seven or eight-digit number, i.e. they consist of the start position and the destination position (path) and a consecutive number between 1 and 99.

  Example:

  The first intermediate position number in the movement routine from 10 to position 20 is 10|20|01, and the third intermediate position between position 300 and position 50 is 300|050|03.
- If the start position has three digits, the target position must also be specified as a three-digit number. If the target position has only two digits, it has to be enlarged to a three-digit number by adding a leading zero (for example, 300|050|01). If the start position is allocated with two digits and the target position with three digits, no leading zero may be inserted (for example, 50|300|01).
- The direction of movement for automatic continuation of the movement can be reversed by swapping the starting position and the destination position in the intermediate positions.

  Example:

  If the robot moves into a machine (from 10 to 11) and the robot should always move out of the machine automatically in the event of an error, the intermediate positions must be 11|10|xx. If the robot is to continue to the destination point, the intermediate positions must be 10|11|xx .
- The home position gets the number 999.

Examples for intermediate positions

Start position	Destination position 20 [2-digit]	Destination position 200 [3-digit]
10 [2-digit]	10| 20|01 [6-digit]	10|200|01 [7-digit]
100 [3-digit]	100|020|01 [8-digit]	100|200|01 [8-digit]

10.5.3 Structure of the movement routines

Introduction

The names of the movement routines consist of the start position, the destination position and the type number (if necessary).

The possible **HomeRun** strategies are explained in the following on the example of movement routines from position 10 to position 11 (`mv10_11`) and from position 11 to position 10 (`mv11_10`):

```
PROC mv10_11()
!From: Pre position Machine 1
!To : Grip position Machine 1
!IM-Pos: 1110xx
MT_MoveJ 10,p10,v100,z10,tGripper\NoMove; No movement takes place
MT_MoveL 111001,*,v1000,z10,tGripper;
MT_MoveL 111002,*,v1000,z10,tGripper;
MT_MoveL 11,p11,v500,fine,tGripper;
ENDPROC
```

Move back routine with identical movement sequence

In the move back routine with identical movement sequence, the movement commands (positions and position numbers) must be programmed in the reverse order. In this case the same number of movement commands or intermediate positions between start and destination point must be used. This is necessary since every position number is checked when moving into the home position. If a different number of movement instructions or different position numbers has been used in the return movement, automatic movement to the home position is not possible.

If intermediate position 1110xx is used, routine mv11_10 is called up automatically when automatic movement from an intermediate position to the home position takes place.

```
PROC mv11_10()
!From: Grip position Machine 1
!To : Pre position Machine 1
!IM-Pos: 1110xx
MT_MoveJ 11,p11,v100,z10,tGripper\NoMove; No movement here
MT_MoveL 111002,*,v1000,z10,tGripper;
MT_MoveL 111001,*,v1000,z10,tGripper;
MT_MoveL 10,p10,v1000,z10,tGripper;
ENDPROC
```

Continues on next page

10.5.3 Structure of the movement routines
Continued

Backwards movement with other movement sequence

If one moves with the robot into a machine to fetch or set down a part, then when moving out a movement sequence other than for moving in may have to be used.

The following sequence can be programmed when using a machine destination position (for example, 12):

If intermediate position 1110xx is used, routine mv11_10 is called up automatically when automatic movement from an intermediate position to the home position takes place.

```
mv10_11; Movement from pre position to grip position
OpenGripper;
mv11_12; Movement from grip pos. to destination pos.
```

The movement routines `mv10_11` and `mv11_10` correspond to the previous example. If the move back routine (for example, `mv11_10`) is not required in the normal program run, routine `mv10_11` can be run backwards in the movement into the home position by using the `MT_MoveRoutine mv10_11` instruction.

Different movement sequences in the same positions

If only one pre position is going to be used, different position numbers must be used in the two movement routines.

```
mv10_11; Movement from pre pos. to grip pos.
OpenGripper;
mv11_10; Movement from grip pos. to pre pos.
```

Movement to the grip position:

```
PROC mv10_11()
!From: Pre position Machine 1
!To : Grip position Machine 1
!IM-Pos: 1011xx
MT_MoveJ 10,p10,v100,z10,tGripper\NoMove;
MT_MoveL 101101,*,v1000,z10,tGripper;
MT_MoveL 101102,*,v1000,z10,tGripper;
MT_MoveL 11,p11,v500,fine,tGripper;
ENDPROC
```

Backwards movement with other movement sequence:

```
PROC mv11_10()
!From: Grip position Machine 1
!To : Pre position Machine 1
!IM-Pos: 1110xx
MT_MoveJ 11,p11,v100,z10,tGripper\NoMove;
MT_MoveL 111001,p101101,v1000,z10,tGripper;
MT_MoveL 111002,*,v1000,z10,tGripper;
MT_MoveL 111003,*,v1000,z10,tGripper;
MT_MoveL 10,p10,v1100,z10,tGripper;
ENDPROC
```

Continues on next page

Different intermediate position numbers were used in both routines, so that on automatic continuation of the movement, either routine `mv10_11` or routine `mv11_10` is selected.

If in the case of a program cancellation during movement into the machine (`mv10_11`) the movement is not continued up to grip point 11, but is moved back to pre position 10, this must be explicitly called up in the home run strategy for position number 1011 by means of backwards execution of routine `mv10_11`.

Example:

```
PROC MT_HomeRun(num Position)
TEST Position
CASE 1011:
MT_MoveRoutine "mv10_11";
CASE ...
```

10.5.4 Strategy for automatic movement into the home position

Inrroduction

An application-dependent HomeRun strategy must be stored in the MT_HomeRun routine, for a robot to move automatically into the home position. In this strategy the sequence of the required movement routines or a certain program sequence depending upon the position number must be programmed. In each case it is only run up to the end of a movement routine and then the HomeRun strategy is called up again with the new position. This is continued until the home position is reached.

In accordance with the rules for allocating position numbers, the are two-digit and three-digit position numbers for the start and destination positions of a movement routine (for example, 10, 20, 300, and so on) as well six and eight digit intermediate positions (for example, 10|20|01, 20|305|03, 305|020|01 , and so on).

To select the movement routines with intermediate positions, only the path is taken into account, since this specifies the start and the destination of the movement (for example, 1020). The consecutive numbers (last two digits) are used for searching for the start position for a movement. The HomeRun strategy must be generated depending upon the normal program sequence, i.e. the order of the routines to be called up or the conditions that are needed (for example, signals, gripper opening and so on) must be incorporated in suitable locations. The required strategy is inserted in the TEST instruction of the MT_HomeRun routine by checking the position number and selecting the required instructions.

Example:

```
PROC MT_HomeRun(num Position)
TEST Position
CASE 10:
mv10_999;
CASE 11:
IF diOpenGripper=high then !Gripper is already opened
OpenGripper; !Move without part back to pre position
MT_MoveRoutine "mv10_11";
ELSE
CloseGripper; !Gripper is not open,
mv11_12; !move out of machine with part
ENDIF
CASE 30.3031:
IF diPartinGr=high THEN
!Position no. is not checked after the movement
bMT_HomeRunCheckPos:=False
Load_Belt; !Repetition of the complete process
ELSE
mv30_999;
endif
DEFAULT:
MT_ContHomeRun Position;
ENDTEST
ENDPROC
```

Continues on next page

Tip

To simplify the production of the HomeRun strategy, movement routines that are described by a path statement (intermediate position) can be selected automatically with the `MT_ContHomeRun` instruction

In this way the direction in which the robot continues moving automatically during the HomeRun can already be determined during movement programming by means of the path specification

Example:

The robot moves for unloading into a machine. The movement routine for entering the machine is **mv10_11**, and the routing for exiting is **mv11_10**. If intermediate positions **1011xx** are specified, the robot continues to move to position 11 during the HomeRun. However, if you use intermediate positions **1110xx**, the robot automatically moves back to position 10.

Structure of the HomeRun strategy

The routine calls for the home run strategy can be structured as follows:

- Because the intermediate positions are evaluated automatically (see `MT_ContHomeRun`), they only have to be taken into consideration in the strategy if other conditions or movement reversal are required.
- It must be decided at a two-digit or three-digit pre position whether the robot can move directly into the home position or whether it has to move to the next station, since for instance the part must be placed in the gripper.
- If no direct movement into the home position is possible from a pre position, then the next station must be moved to until it is possible to move into the home position.
- If the robot is at a grip position, then it is decided by reference to the gripper's status whether the robot moves out from the station with or without a part, i.e. if the gripper is closed the part is taken along, if the gripper is open the robot moves out from the station without a part.
- If there are interlocking signals for the machine (for example, open/close clamp and so on), then these must be taken into account.
- If the robot is in a deposit position, an attempt should be made to put the part down.
- To avoid collisions, the gripper should be opened only after a movement.

Continues on next page

10.5.4 Strategy for automatic movement into the home position
Continued

Behaviour at a start position

If the program is stopped, the robot is always located as a rule on the way between two positions. Since the new position number is saved only on reaching the destination position, `HomeRun` assumes that the robot is still located at the previous position.

To prevent collisions during the movement to the home position, the saved two-digit or three-digit start position is moved to once again for this reason.

Depending on the situation, the moving back to the start position can be deactivated by using variable `bMT_ContHomeRun` in the home run strategy.

Example:

```
PROC MT_HomeRun(num Position)
TEST Position
CASE 10:
!deactivate moving back to position 10
bMT_ContHomeRun:=TRUE;
mv10_99;
CASE 11:
…
```

10.5.5 Use of type-related movement routines

10.5.5.1 General

If several component types have to be managed using different movement routines in a robot program, they can be marked using a type-dependent index in the routine or position name.

In order to simplify the copying of type-dependent program modules, a type prefix (for example, **T**) should be used, which makes it easier to make a distinction between a position number and the type index during searching and replacement.

The type prefix to be used can be set using system parameter `PROC/MT_HOMRUN/TypePrefix`.

Example:

Movement from position 10 to 20

Type index 3 and type prefix **T**: `mv10_20_T3`

Type index 5 and type prefix **WN**: `mv10_20_WN5`

Since as a rule only the movement routines but not the general sequence changes, these movement routines can be called up using late binding (`%RoutineName%`) within the **`MT_HomeRun`** routine.

According to program there can be a mixture between type-dependent and type-independent movement routines. To avoid a difference between routines with and without type index inside the strategy, the instruction `MT_ContHomeRun` has been designed in such a way that a movement routine will be called automatically without an index, if the movement routine with index does not exist. In order to ensure that this works properly, the movement routine should exist only with or only once in the program: with or without an index.

Example:

```
PROC MT_HomeRun(num Position)
TEST Position
CASE 10.10999:
mv10_999;
CASE 11:
IF diPartinGr=high then
%"mv11_12_T"+ValToStr(nTypeNo)%;
ELSE
OpenGripper;
MT_MoveRoutine "mv10_11"\Index:= nTypeNo;
ENDIF
CASE 12:
%"mv12_30_T"+ValToStr(nTypeNo)%;
CASE 30.30999:
mv30_999;
DEFAULT:
MT_ContHomeRun Position\Index:=nTypeNo;
ENDTEST
ENDPROC
```

10.5.5.2 Use of module-localised movement routines

If a program has to handle different types using different movement routines, these and the associated robtarget can be declared locally within a type module. The module can then be copied and renamed without the need to rework the individual routines and declarations.

Example:

```
MODUL PROG_1

LOCAL CONST robtarget p10:=*
LOCAL CONST robtarget p20:=*

LOCAL PROC mv10_11()
MT_MoveJ 10,p10,v200,z10,tGripper\NoMove;
MT_MoveL 101101,*,v1500,fine,tGripper;
MT_MoveL 11,p11,v1500,fine,tGripper;
ENDPROC

ENDMODULE
```

Since as a rule only the movement routines but not the general sequence changes, these movement routines can be called up using late binding (`%"Module:Routine"%`) within the **MT_HomeRun** routine or during the normal program sequence, (for example, `%"PROG_1:mv10_11"%`).

To permit movements to be continued or reversed, the instructions `MT_ContHomeRun` and `MT_MoveRoutine` can optionally be used to pass the name of the program module in which the movement routines are located.

Example:

```
PROC MT_HomeRun(num Position)
TEST Position
CASE 10,10999:
%"PROG_"+ValToStr(nTypeNo)+":mv10_999"%;
CASE 11:
IF diPartinGr=high then
%"PROG_"+ValToStr(nTypeNo)+":mv11_12"%;
ELSE
OpenGripper;
MT_MoveRoutine "mv10_11"\ModName:="PROG"\Index:=nTypeNo;
ENDIF
CASE 12:
%"PROG_"+ValToStr(nTypeNo)+":mv12_30"%;
CASE 30.30999:
!Central movement
mv30_999;
DEFAULT:
MT_ContHomeRun
Position\Module:="PROG_"+ValToStr(nTypeNo);
ENDTEST
ENDPROC
```

10.5.5.3 Use of type modules with different strategies

If the program sequences of the individual component types differ in terms of functionality and position, a separate position-related HomeRun strategy must be created for each component type.

Example:

Different component types are created in a robot cell, which the robot takes out of a machine and performs different processing steps depending on the component. The robot cell contains a gripper station to which the robot must be able to go to in order to perform an automatic gripper change.

Possible solution:

The sequences and movement routines are stored locally in type-dependent modules. The gripper change movement routines are stored globally in the robot system.

For each component type a locally declared HomeRun strategy is created in the component module for each component type, which is called up by the global HomeRun strategy depending on the type.

All type-dependent movements to the home position are stored in this type-dependent HomeRun strategy. All global movements (gripper changes) and the continuation of an intermediate position are only dealt with in the global strategy routine.

Global HomeRun strategy:

```
MODULE MT_Main()
PROC MT_HomeRun(num Position)
VAR string stModule;
IF nTypeCode>0 THEN
! Determine type-dependent module name
stModule:="PROG"+ValToStr(nTypeCode);
!Call type-dependent strategy
%stModule+":MT_HomeRun"% Position;
ENDIF
!Processing of type-dependent positions
TEST Position
CASE 700:
MT_MOVEROUTINE "mv999_700";
CASE 711:
IF diGrDocked=high THEN
WZW_Lock;
mv711_710;
ELSE
WZW_Unlock;
MT_MoveRoutine "mv710_711";
ENDIF
DEFAULT:
MT_ContHomeRun Position\Module:=stModule;
ENDTEST
ERROR
```

Continues on next page

10.5.5.3 Use of type modules with different strategies
Continued

```
IF ERRNO=ERR_CALLPROC OR ERRNO=ERR_REFUNKPRC TYNEXT;
ENDPROC
ENDMODULE
```

Type-dependent local HomeRun strategy:

```
MODULE PROG1

LOCAL PROC MT_HomeRun(num Position)
!
TEST Position
CASE 100:
mv100_999;
CASE 110:
mv110_100;
CASE 130:
OpenGripper;
MT_MoveRoutine "mv110_130"\ModName:="PROG1";
ENDTEST
!
ENDPROC
LOCAL PROC mv100_999()
…
…
ENDPROC
ENDMODULE
```

Program sequence

If a valid component type (for example, **1**) is selected, the type-dependent **HomeRun** strategy in module `PROG1` is called up.

If the current robot position corresponds with one of the positions in the local strategy (for example, **130**), the relevant sequence is performed and the global HomeRun strategy is returned to.

WARNING

The same position number is then evaluated, meaning that this can no longer be contained within the `TEST-CASE` control structure, since double execution of the robots could cause a collision.

In this case, the program pointer runs through the `DEFAULT` part of the `TEST-CASE` control structure, and executes the `MT_ContHomeRun` instruction.

If the current position is a pre-position or an end position (2-3 digit number), the instruction is exited again immediately and the HomeRun strategy is called with the next position number.

If "`MT_ContHomeRun`" has been called with an intermediate position number, it is first attempted to call the module-local routine or, if this does not exist, the global movement routine.

10.5.6 MultiMove Support

HomeRun can be used in robot systems with a maximum of 4 robot manipulators, whereby only **MultiMove-Independent** is supported.

The movements of the robot manipulators into the home position are not synchronized by HomeRun which means that the strategy of each robot is independent from the strategies of the other robots in the MultiMove system.

If there is a need to lock positions between the robot manipulators, this has to be done in the individual strategies of the robots.

If the robots are to be moved into the home position in a certain order, event routines for the events `EE_BEFORE_INIT` and `EE_AFTER_INIT` can be set up by the integrator.

Example:

In this example, the following event routines have been set up:

Routine	Assigned event
`BeforeHomeRun`	`EE_BEFORE_INIT`
`AfterHomeRun`	`EE_AFTER_INIT`

To send the robot to the home position in a sequential order, those routines can be filled as follows:

Robot 1 Starts moving immediately	Robot 2 Is waiting for robot 1	Robot 3 Is waiting for robot 1 and 2

`PROC BeforeHomeRun()`
`ENDPROC`
 	
`PROC AfterHomeRun()`
`WaitDI diMT_RobotsInHome,high;`
`ENDPROC`
	`PROC BeforeHomeRun()`
	`WaitDO doRob1inHome,high;`
	`ENDPROC`
	
	`PROC AfterHomeRun()`
	`WaitDI diMT_RobotsInHome,high;`
	`ENDPROC`
		`PROC BeforeHomeRun()`
		`WaitDO doRob1inHome,high;`
		`WaitDO doRob2inHome,high;`
		`ENDPROC`
		
		`PROC AfterHomeRun()`
		`WaitDI diMT_RobotsInHome,high;`
		`ENDPROC`

If a robot has reached its home position, it waits until all other robots have reached their home positions too before continuing the program execution.

This is achieved by waiting in the `AfterHomeRun` routine until the digital output `doMT_RobotsinHome` changes to 1. This output is built by a cross connection, where the home position signals of all robots are combined logically´.

If the predefined outputs `doRob1inHome – doRob4inHome` are not to be used, this cross connection in the system parameters must be adapted accordingly or the individual signals must be queried.

10.5.7 Movement continuation in intermediate positions

If the robot program was stopped in an intermediate position of a movement routine and the robot should then move into the home position, then the movement must be restarted exactly from this intermediate position. Since processing of the instructions starts at the beginning of a routine, a facility for preventing execution of the robot movement until the previously moved to intermediate position has been found has been implemented in the instructions `MT_MoveX`, `MT_MoveXDO`, `MT_TriggX`, and so on.

For this purpose a check is made in the movement into the home position whether the last moved to position number agrees with the transferred position number. The robot movement is not performed if this is not the case. If the two position numbers agree, then this and all following robot movements are performed again.

Example:

The robot has been stopped in position 123002. After a robot program restart, the HomeRun is executed. The movement routine to be called up is determined from the saved position number. In intermediate positions the consecutive number is separated, i.e. current position 123002 becomes path number 1230, and two-digit start or end positions are evaluated directly (for example, 12 or 30). On the basis of path number 1230, the robot continues to end position 30 by calling up routine mv12_30. In relation to the respective current position, movement routines are called up until the robot is in the home position (999).

```
PROC mv12_30()
MT_MoveL 12,p12,… !No movement, pos.no.   12   123002
MT_MoveL 123001,… !No movement, pos.no.   123001   123002
MT_MoveL 123002,… !No movement, pos. no.   123002 = StartPos
MT_MoveL 123003,… !Movement is performed since start position
!has been found
MT_MoveL 30,p30,… !Movement is performed
ENDPROC
```

If the robot is between two positions, then on continuing the movement in the same direction the saved position is no longer moved to, but the movement starts with the following position.

The saved position is only moved to if the direction of movement reverses (for example, 3012xx used as intermediate position number in routine `mv12_30`) or if the saved position is a start or destination point (two-digit or three-digit position).

10.6 System characteristics

10.6.1 Position number assignment

The assignment of position numbers takes place by means of an interrupt in the middle of a zone path of the destination point of a movement instruction.

If the destination position is moved to as a stop point (**fine** zone) the position number is assigned as soon as the manipulator has come to a standstill.

The following restrictions apply:

- During execution forwards instruction-by-instruction the I/O activities are performed but the interrupt routines do not run.
- During the execution backwards instruction-by-instruction, no trigger activities at all are performed.

10.6.2 Intermediate position in movement from the home position

If the robot is stopped during the movement from the home position to a destination position before it has reached an intermediate position or the destination position, then only direct movement into the home position is possible (`MT_HomeDirect`), since the position number 999 is still active, but the robot is not in the home position.

An intermediate position that the robot reaches immediately after setting off should be inserted after the home position in order to keep `HomeRun` active.

Example:

```
PROC 999_10()
MT_MoveJ 999,p999,v200,z10,tGripper\NoMove;
MT_MoveJ 99901001,p999,v1000,z10,tGripper;
MT_MoveJ 99901002,*,v2000,z10,tGripper;
MT_MoveL 10,p10, v2000,z10,tGripper;
ENDPROC
```

Position number 99901001 is assigned immediately when the robot starts moving.

10.7 Programming and configuration data

10.7.1 Introduction

All signals, data and instructions that HomeRun uses or makes available are described in the following.

10.7.2 Modules

The project-specific adaptations must be inserted in the routines.

- MT_HomeRun
- MT_SpeedUpdate
- MT_HomeDirect

and in the declaration `pnPositions` of type `posname`.

10.7.3 Signals

Internal signals

Signals on the system-internal I/O unit HOME.

Type	Signal name	Adr.	Description
DI	diHR_TaskStopped	0	All movements tasks are stopped
DO	doHR_Task1Run	1	Task T_ROB1 is executed
DO	doHR_Task2Run	2	Task T_ROB2 is executed
DO	doHR_Task3Run	3	Task T_ROB3 is executed
DO	doHR_Task4Run	4	Task T_ROB4 is executed
DO	doHR_Trigg10 - 49		Position trigger signals

Remote control signals

Signals on the I/O unit HOMESIM

Type	Signal name	Adr.	Description
DI	diIRBgoHome	0	HomeRun request
DI	diHR_MotorOn	1	*Switch on motors* request
DI	diHR_StartMain	2	*Start program from main* request
DI	diHR_RobotsInHome	3	All robots are in their home positions
DO	doRob1inHome	1	Robot 1 is in the home position
DO	doRob2inHome	2	Robot 2 is in the home position
DO	doRob3inHome	3	Robot 3 is in the home position
DO	doRob4inHome	4	Robot 4 is in the home position
DO	doHR_CycleOn	5	Program is executed
DO	doHR_MotorOn	6	Motors are switched on

Tip

These signals are pre-installed and can be adapted in EIO.CFG if necessary.

10.7.4 Data

Name	Type	Description
bMT_ContHomeRun	bool	Move to start position (two or three-digit) when searching the current position (**FALSE**) or not (**TRUE**). These variables can be assigned in routine MT_HomeRun depending on the situation.
bMT_HomeRunCheckPos	bool	Perform position check of last position moved to in MT_HomeRun.
bMT_HomeRunActive (readonly)	bool	HomeRun is executed.
stMT_HomeRunActPos (readonly)	string	Last position number that robot moved to.
nMT_HomeRunTarget (readonly)	num	Destination position of last movement routine to be executed.

10.7.5 Instructions

Name	Description
MT_ContHomeRun	Continue a movement routing.
MT_Exit	Program processing complete.
MT_ExitCycle	Abort current cycle and start next cycle.
MT_MoveRoutine	Execute movement routine (backwards)
MT_HomeRunSavePos	Save stop position
MT_MoveJ	Axis-wise movement.
MT_MoveL	Linear robot movement.
MT_GripJ	Axis-wise movement, including a gripper action
MT_GripL	Linear movement, including a gripper action
MT_GripSeqJ	Axis-wise movement, including a gripper sequence
MT_GripSeqL	Linear movement, including a gripper sequence
MT_CSSDeactMoveL	Linear movement, which deactivates a cartesian soft servo.
MT_MoveJDO	Axis-wise movement and set digital output in corner path.
MT_MoveJGO	Axis-wise movement and set group output in zone.
MT_MoveJSync	Axis-wise movement and processing a RAPID procedure.
MT_MoveL	Linear movement.
MT_MoveLDO	Linear movement and setting a digital output in the zone.
MT_MoveLGO	Linear movement and set group output in zone.
MT_MoveLSync	Linear movement and processing of a RAPID procedure.
MT_SearchL	Linear search movement of robot.
MT_TriggJ	Axis-wise movement with events.
MT_TriggL	Linear movements with events.
MT_TorqueL	Linear movement for option TorqueSearch

Note

All of the above-mentioned instructions are available in the HomeRun instruction list of the FlexPendant.

10.7.6 HomeRun related routines in the MT_MAIN module

Name	Description
MT_HomeDirect	Movement directly into the home position
MT_SpeedUpdate	Adapt speed
MT_HomeRun	HomeRun strategy

11 System parameters

11.1 Introduction

During the installation of the robotic system with the **RobotWare Machine Tending**, the parameter group **Process** in the system parameters is extended to include the following types:

- MT Visualization settings
- MT API commands
- MT API Positions
- MT Program Selection
- MT Part Settings
- MT Applications
- MT HomeRun

Through these parameter types, the working of the *RobotWare Machine Tending* is set task related. A parameter instance exists for every motion task of the robot for this, with the name of the respective task (for example, T_ROB1, T_ROB2, T_ROB3 or T_ROB4).

Note

These instances can neither be deleted nor added.

11.2 MT Visualization settings

Overview

This section describes the parameter type `MT Visualization Settings`.

Name of the configuration

`MT_GUI_SETTINGS`

Type description

The type *MT Visualization Settings* contains parameters with which the display behavior of the GUI can be influenced.

Usage

For each motion task, an instance with the respective task name (For example, `T_ROB1`) should be present.

Activating the parameter changes

Changes in the configuration of `MT Visualization Settings` are activated by restarting the GUI and a `Start from main` of the robot program.

Parameter

The parameter type `MT Visualization Settings` contains the following parameters:

Parameter	Description	Permitted values

`name`	The configuration is loaded at the time of installation, depending on the robot existing in the system, and can neither be deleted nor renamed.
	The `name` indicates the task of the robot for which this configuration is applicable.
		`T_ROB1`
		`T_ROB2`
		`T_ROB3`
		`T_ROB4`

`common_name`	In the case of MultiMove systems, the display is done on several tab panes. The labeling of the tab panes or the name of the robot station is defined through `common_name`.	Default:""
`tab_index`	Defines the sequence of the tab panes in the production view, the service routine and set up routine view in the case of MultiMove systems. If no entry is made, the sequence is determined with the help of the task name.	1-4 Default:1
`inhib_part_sel_in_auto`	Disable manual program selection in automatic mode through the graphical user interface. TRUE: Program selection is possible only in the manual mode FALSE: Program selection is possible even in the automatic mode.	TRUE FALSE Default:TRUE
`UseOpModeSelBtn`	Use the **Operation mode selection** button in the GUI. TRUE: button displayed in the GUI. FALSE: button not displayed in the GUI.	TRUE FALSE Default:TRUE

Continues on next page

Parameter	Description	Permitted values
UseStopCycleBtn	Use the **Stop after cycle** button in the GUI. TRUE: button displayed in the GUI. FALSE: button not displayed in the GUI.	TRUE FALSE Default:TRUE
UseHomeRunBtn	Use the HomeRun button in the GUI. TRUE: **HomeRun** button is displayed in the GUI. FALSE:**HomeRun** button is not displayed in the GUI.	TRUE FALSE Default:TRUE
inhib_ghost_mode_sel	Disables the ghost mode (partless production) selection in the GUI.	TRUE FALSE Default:TRUE
wait_time_before_msg	Waiting time until an error message is displayed in the RAPID program, for example, while waiting for a signal.	0,0110[s] Default:5[s]
RobIconCol	Column number at which the robot station is to be displayed in station view of the production window.	1-5 Default:3
RobIconRow	Row number at which the robot station is to be displayed in station view of the production window.	1-3 Default:2
conf_dialog_progstart*	Disabling of extra confirmation dialogs in the GUI production view, when the start of production, respectively a cycle is selected.	TRUE FALSE Default:TRUE
conf_dialog_homrun*	Disabling of extra confirmation dialogs in the GUI production view, when the homerun is selected.	TRUE FALSE Default:TRUE
conf_dialog_gripper*	Disabling of extra confirmation dialogs in the GUI gripper view, when for example, a gripper shall be opened or closed manually..	TRUE FALSE Default:TRUE
cycle_select_all_robot*	Some cycle types should be selectable by one click for all available motion tasks, so that all tasks start the cycle execution at the same time (cycle synchronization). These cycle types are: • Count cycles • Continuous cycles These are the cycle types where the cycle execution buttons are shown only, if the motion tasks are stopped.	TRUE FALSE Default:TRUE
ProjectFolder*	Defines the location, where RWMT projects are saved to. This setting will not influence the project path of a virtual controller, where the projects are always saved under HOME:RWMT.	Path of the project folder. Default: HOME:RWMT

*) This parameter is generally valid for all motion tasks, so RWMT uses only the value entry from the first motion task (T_ROB1). All values for this parameter in other motion tasks will be disregarded.

11.3 MT API Commands

Overview

The parameter type *MT API Commands* is described in this section.

Name of the Configuration

MT_API_COMMANDS

Type description

The type *MT API Commands* contains parameters with which the signals can be set for the remote operation of the MT-functions as well other functional values.

Usage

For each motion task, an instance with the respective task name (for example, T_ROB1) should be present.

Activating the parameter changes

Changes to the configuration of the MT API Commands are activated by a restart of the GUI and Start from mainof the robot program.

Parameter

The parameter type MT API Commands contains the following parameters:

Parameter	Description	Permitted values
name	The configuration is loaded at the time of installation, depending on the robot existing in the system, and can neither be deleted nor renamed. The name indicates the task of the robot for which this configuration is applicable.	T_ROB1, T_ROB2, T_ROB3, T_ROB4
GI_OpMode	Group input for setting the cell mode (**Without robot**, **Service**, or **Production**).	
GO_OpMode	Group output for reporting the current cell mode **Without robot**, **Service**, **Production**)	
OpMode_NoRobot	Group input value for the mode of operation **Operation without robot**	0-255 **Default:** 0
OpMode_Service	Group input value for the mode of operation **Service**.	0-255 **Default:** 1
OpMode_Production	Group input value for the mode of operation **Production**.	0-255 **Default:** 2
no_cell_opmodes	Combines the production and service operation mode so that no selection between production and service mode has to be done in the GUI	TRUE, FALSE **Default:**TRUE
DI_GhostMode_REQ	Digital input for the selection **Ghost mode**.	
DO_GhostMode_ACK	Digital output for the feedback if **Ghost mode** is active.	

Continues on next page

Parameter	Description	Permitted values
DI_Stop_Cycle	Digital input for the request **Halt after end of cycle**.	
direct_stop_after_cycle	Flag to force a direct halt after cycle instead of just requesting it when pressing the halt after cycle button	TRUE, FALSE **Default:**TRUE
stop_after_cycle_timeout	Timer which is started if a halt after cycle request is set and after unsuccessfully trying to get a valid program number or request from the GUI to either execute a production or service routine. If a production or service routine can be run before timeout, the timer is stopped and reset. If the timeout is triggered before a production or service routine can be executed, then the **Stop after cycle request** changes automatically to **Stop after cycle reached** and the engine loop is cancelled (program stop). The value **0** [s] disables the timer.	0-600 [s] **Default:** 0 [s]
DO_Error	Digital output for the error notification.	
GO_Error_Domain	Group output for transmitting the error domain.	
GO_Error_No	Group output for transmitting the error number.	
DI_Error_ACK	Digital input for acknowledging an error.	
DI_Speed1_REQ	Activate speed override 1. Signal is high: The speed of movement is set with the instruction SpeedRefresh to Speed1_Value. Signal is low: The speed of movement will be set to 100%.	
DO_Speed1_ACK	Speed override 1 is active (high).	
Speed1_Value	Speed override 1 in percentage.	10-100 [%] **Default:** 30 [%]
DI_Speed2_REQ	Activate speed override 2. Signal is high: The speed of movement will be set to Speed2_Value. Signal is low: The speed of movement will be set to 100%.	
DO_Speed2_ACK	Speed override 2 is active (high)	
Speed2_Value	Speed override 2 in percentage	10-100 [%] Default:50[%]
DI_AirPressure_OK	Digital input for checking the pressurized air. Signal = 1: Pressurized air OK Signal = 0: Pressurized air error	

Continues on next page

Note

If no signals have been assigned for individual functions, then this function will not be used in the robot program.

11.4 MT API Positions

Overview

This section describes the parameter type `MT API positions`.

Name of the configuration

`MT_API_POSITION`

Type description

The type `MT API Positions` contains the parameters through which the signals for the position messages (for example, world zone signals for the home position, safe position or service positions) as well as the requests for moving to one of these position can be set.

Usage

For every task of the robot, an instance with the respective task name (for example, `T_ROB1`) should be present.

Activating the parameter changes

Changes in the configuration of the `MT API positions` a by a restart of the GUI and through a `Start from main` of the robot program.

Note

If static world zones are used for the position monitoring, a warm start of the controller may be necessary.

Parameter

The parameter type `MT API Positions` contains the following parameters:

Parameter	Description	Permitted values
`name`	The configuration is loaded at the time of installation, depending on the robot existing in the system, and can neither be deleted nor renamed. The `name` indicates the task of the robot for which this configuration is applicable.	`T_ROB1`, `T_ROB2`, `T_ROB3`, `T_ROB4`
`DIO_At_Home`	Digital input or output for checking whether the robot is present in the **Home position**. High: robot is located in the **Home** position	
`DIO_At_Safe`	Digital input or output for checking if the robot is present in the Safe position.	
`DIO_At_Service1`	Digital input or output for checking if the robot is present in the **Service position 1**.	
`DIO_At_Service2`	Digital input or output for checking if the robot is present in the **Service position 2**.	
`DIO_At_Service3`	Digital input or output for checking if the robot is present in the **Service position 3**.	

Continues on next page

Parameter	Description	Permitted values
DI_Go_Home	Digital input for requesting a trip of the robot to the **Home position**.	
DI_Go_Service1	Digital Input for requesting a trip of the robot to the **Service position 1**. Moving to service position 1 is allowed in operation mode **Production** as well as in operation mode **Service**. The onliest precondition is that the robot is located in home position before.	
DI_Go_Service2	Digital Input for requesting a trip of the robot to the **Service position 2**. Moving to service position 1 is allowed in operation mode **Pproduction** as well as in operation mode **Service**. The onliest precondition is that the robot is located in home position before.	
DI_Go_Service3	Digital Input for requesting a trip of the robot to the **Service position 3**. Moving to service position 1 is allowed in operation mode **Production** as well as in operation mode **Service**. The onliest precondition is that the robot is located in home position before.	
Pos_no_Safe	Position number of the safe position. If, during Start from main, the robot is in the safe position (DIO_At_Safe), then the start position for the instruction MoveTo will be set to this position number, so that error free processing is possible. **Note** The assignment of a position number <>0 may only be done, if there is only one safe-position. Otherwise a collision could happen because when having more than one safe position, the assigned number would be ambiguous.	0-998 Default: 0

Note

If no signals have been assigned for individual functions, then this function will not be used in the robot program.

11.5 MT Program Selection

Overview

This section describes the parameter type `MT Program selection`.

Name of the configuration

`MT_PRG_SELECT`

Type description

The type `MT Program Selection` contains the definition of the interface, which is necessary to transfer a program number.

Usage

For every task of the robot, an instance with the respective task name (for example, `T_ROB1`) should be present.

Activating the parameter changes

Changes to the configuration of the `MT Program Selection` are activated by a restart of the GUI and through a `Start from main` of the robot program. Alternatively, a warm start of the robot controller can be executed.

Parameter

The parameter type `MT Program Selection` contains the following parameters:

Parameter	Description	Permitted values
`name`	The configuration is loaded at the time of installation, depending on the robot existing in the system, and can neither be deleted nor renamed. The `name` indicates the task of the robot for which this configuration is applicable.	`T_ROB1`, `T_ROB2`, `T_ROB3`, `T_ROB4`
`cyclic_prg_read`	Cyclic reading of the program number. TRUE: To execute a processing program or service routine, the program number is read in every cycle, using the selected handshake signals. FALSE: The program number is read after the `Start from main`, once, using the selected handshake signals. After this, the same processing program will be executed repeatedly.	TRUE, FALSE Default: False
`GI_Prg_Number`	Group input. Program number (See part data)	
`GO_Prg_Number`	Group output. Returns the program number.	
`GI_Check_Code1`	Group input. Test code 1, for example, form code (See part data).	
`GI_Check_Code2`	Group input. Test code 2 (See part data)	

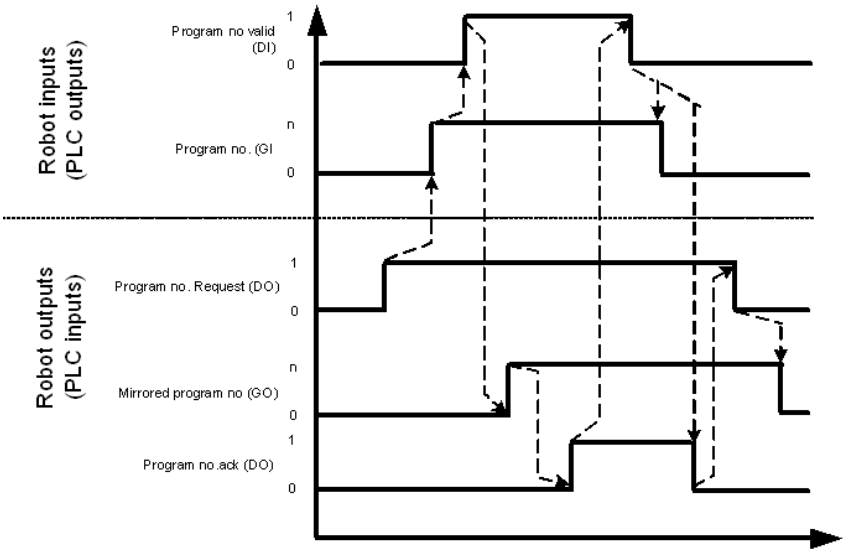
Continues on next page

Parameter	Description	Permitted values
GI_Check_Code3	Group input. Test code 3 (See part data)	
GI_Check_Code4	Group input. Test code 4 (See part data)	
GI_Check_Code5	Group input. Test code 5 (See part data)	
GI_Check_Code6	Group input. Test code 6 (See part data)	
GI_Check_Code7	Group input. Test code 7 (See part data)	
GI_Check_Code8	Group input. Test code 8 (See part data)	
GI_Tool_Code	Group input. Gripper coding (See partdata and gripdata)	
DO_Prg_Request	Digital output. Request for program number. (Ready for data transmission	
DI_Prg_Valid	Digital input. Program number is valid	
DO_Prg_RunACK	Digital output. Acknowledgement of program number transfer from robot	
DO_Prg_Running	Digital output. Selected program will be executed	
GI_Cycle_Index	Group input. Selection of production cycle	
DI_Reset_CycCnt	Digital input. Request for resetting the cycle counter after the production has been cancelled.	
DO_ResCycCnt_ACK	Digital output. Acknowledgement cycle counter was reset	
EnableUserProg	Transfer of program number will be done with user controls. TRUE: The program number will be transferred through the user defined routine MT_GetUserProgNo. FALSE: The program number will be transferred internally through the defined handshake signals.	TRUE, FALSE Default: False
wait_time_prog_number	Maximum time in seconds while waiting for a valid program number. After time has exceeded, the event EE_TIMEOUT_PROG_NUMBER is triggered where the user can force stop after cycle.	5 – 600 [s] Default: 10 [s]

Continues on next page

Signal flow chart

The following figure shows a signal flow chart with the signals that are necessary for communicating the program numbers. In case that not all signals are used, the signal flow is modified analogously.



en1200000772

Additional information are given in the chapter *Parameterization of the MT Program Selection on page 191*.

11.6 MT Part Settings

Overview

In this section, the parameter type `MT Part Settings` will be described.

Name of the configuration

`MT_PART_SETTINGS`

Type description

The type `MT Part Settings` contains parameters to define how part type related information is to be handled.

Usage

An instance with the respective task name (for example, `T_ROB1`) must be present for each robot task.

Activating the parameter changes

Changes to the configuration of the `MT Part Settings` are activated through a **Start from main** of the robot program.

Parameter

The parameter type `MT Applications` contains the following parameters:

Parameter	Description	Permitted values
`name`	The configuration is loaded at the time of installation, depending on the robot existing in the system, and can neither be deleted nor renamed. The `name` indicates the task of the robot for which this configuration is applicable.	`T_ROB1`, `T_ROB2`, `T_ROB3`, `T_ROB4`
`partmod_name`	General module name of the part type modules without type prefix and without part type number. For example,: if a specific part type module for part type 137 shall be `PART_T137`", then the value for this parameter must be: **PART** (without type prefix **T** and without part type number 137).	
`TypePrefix`	Type prefix to be used for type-dependent movement routines.	Default:**T**

11.7 MT Applications

Overview

In this section, the parameter type `MT Applications` will be described.

Name of the configuration

`MT_APPLICATIONS`

Type description

The type `MT Applications` contains parameters to declare external, station independant FlexPendant applications, that can be called from the production view of the GUI.

Usage

For each application, an instance with the respective name APP1…APP8 should be present.

Activating the parameter changes

Changes to the configuration of the `MT Applications` are activated by a restart of the GUI.

Parameter

The parameter type `MT Appplications` contains the following parameters:

Parameter	Description	Permitted values
`name`	Unique name of the application settings, must be selected from list.	`APP1, APP2,, APP3, APP4, , APP5, APP6,, APP7, APP8,`
`MenuName`	Name of the application, appearing in the application pull-down menu in the production view of the RWMT GUI	
`Image`	The file name of the image which will be shown on the left side of a menu entry (max. 32x32 pixel). Menu height depends on the largest image.	
`DLLname`	Name of the application DLL, including the file extender	
`Embedded`	Can be either embedded (application is started as child of the RWMT GUI) or as a separate application.	TRUE, FALSE **Default:**FALSE
`Namespace`	Namespace of application. (Standard namespace. `ABB.Robotics.SDK.Views` is used, if field is empty.	**Default:** `ABB.Robotics.SDK.Views`
`Class`	Class (view) of the application that shall be started, not needed for external (not embedded) applications.	

Continues on next page

Limitations and characteristics

- If there is just one station application for a specific station, its name, respectively its icon will be shown directely in the menu bar in the GUI station view.
- To be able to run an embedded applications on a virtual controller, the application files (DLL's) must be located in the **Home** or **System** directory. At the real controller there a no limitations.
- The embedded applictions should have a button which closes the window, otherwise, there is no way to close the application.
- (FP-SDK): The app should use error handling, so that each error will be handled by the application itself.
- (FP-SDK): The constructor of the application should be the standard constructor which has no parameters. Each application must create the required data by itself.
- (FP-SDK): When closing the application, all used resources must be released by means of the **Dispose** method, so that no memory leaks appear.
- (FP-SDK): The Interface **ITpsViewActivation**" should be implemented, so that the methods **Activate** and **Deactivate** will be called if the control goes from the passive state to the active state, that is, becomes visible in the client view. Normally, this is where subscriptions to controller events are set up.
- (FP-SDK): Only **TpsForms** can be used.

11.8 MT HomeRun

Overview

This section describes which system parameters of HomeRun are used, and how they have to be set.

Name of the Configuration

MT_HOMERUN

Type description

The HomeRun type contains the parameters with which the behaviour of the automatic move to the home position can be influenced.

Usage

An instance with the respective task name (for example, T_ROB1) must be present for each robot task.

Activating the parameter changes

Changes to the configuration of HomeRun are activated by a robot program Start from main.

Parameter

The HomeRun parameter type contains the following parameters:

Parameter	Description	Permitted values
name	The configuration is loaded during installation depending on the robots that exist in the system, and can be neither deleted nor renamed. The name specifies the robot task to which this configuration applies.	T_ROB1, T_ROB2, T_ROB3, T_ROB4
UseHomRun	Selection/deselection of **HomeRun**. Allows to decide wether **HomeRun** shall be used in the application program or not.	TRUE, FALSE **Default**:TRUE
DO_HomeRunActive	Digital output that is set when the HomeRun is being executed.	
CreateWZone	A temporary world zone is set up for monitoring the home position. In order to do this, an output must have been set up for monitoring the home position. The position of the robot after executing the MT_HomeDirect routine is used as the home position. Only available if the **World Zones** option is used, which is part of the RWMT option key.	TRUE, FALSE **Default**:FALSE
MaxSpeed	Maximum speed (mm/s) that is set when the move to the home position takes place (VelSet).	10-1000 Default:500
Override	Maximum speed override (%) that is set when the move to the home position takes place (VelSet).	10-80 Default:50

Continues on next page

Parameter	Description	Permitted values
StartArea	Maximum permitted distance (mm) that the robot may be moved away from the last moved to position to enable automatic movement into the home position. A value of 0 disables the check. If the robot has been moved manually using the joystick after deactivation of the check and is then to move automatically to the home position, it will move into the home position via the last stored position number, which may cause a collision. In order to avoid this, the robot should be moved to the home position in **Manual** mode after moving the robot manually. After starting the program from **Main** without any external request for the home position, the robot can be moved directly to the home position in manual mode by pressing the **Direct** menu button.	0-150 Default: 150
NotUsedExtAxes	External axes number 7-12 which shall not be used for home position verification. This is necessary if an external axis is controlled by an external device, for example, when using one of the options **MaschineSync** or **Conveyor tracking**. The axis which is not required will be deactivated by representation of the axis number (7 -12). If several axes shall be deselected for home pos verification, the axis numbers have to be separated by a blank or a comma character, like **7 8 12** or like **7,8,12**.	
StopInHome	TRUE: Stop program execution after movement into the home position. FALSE: Program is continued after reaching the home position.	TRUE, FALSE Default: TRUE
AbortProgram	TRUE: Terminate program execution and start move to the home position when digital input DI_Go_Home is activated.	TRUE, FALSE Default:TRUE
HoldToRun	TRUE: Abort movement of robot into home position when input DI_Go_Home is reset.	TRUE, FALSE Default:FALSE
ExecTriggEvt	TRUE: The programmed trigger events of the movement instructions are executed if **HomeRun** is active. FALSE: No trigger events executed with HomeRun active.	TRUE, FALSE Default:TRUE
WaitGoHomeLow	TRUE: Wait until DI_Go_Home input switches to 0 as soon as the home position has been reached. FALSE: The status of DI_Go_Home is ignored after the home position has been reached.	TRUE, FALSE Default:TRUE
SpeedUpdate	TRUE: The MT_SpeedUpdate user routine is called up before executing a movement instruction so that the speed data can be adapted. FALSE: The **MT_SpeedUpdate** user routine is not called up.	TRUE, FALSE Default:FALSE

12 User permissions

12.1 Application permissions

`RobotWare Machine Tending` provides the following application permissions, which can be used to release the access permissions for the individual functions of the GUI in a user specific manner.

Application 'Grant'	Description
Select Parts `RWMT_SEL_PARTS`	If this is activated, the user can manually select a part in the component view.
Station selection `RWMT_SEL_STATION`	If this has been activated, the user is not allowed to select or deselect stations.
GhostMode access `MT_GHOSTMODE`	If this has been activated, the ghost mode may be requested through the GUI.
Gripper control `MT_GRIPPER`	If this has been activated, the user can operate the gripper manually through the GUI, that is, the control elements as well as the gripper sequences can be executed.
Signal write access `MT_WRITE_SIGNALS`	If this has been activated, the signals of the stations or the general signal page can be set or reset.
Variable Write access `MT_VAR_WRITE` Contains the access to the reset button	This **Grant** corresponds to the `minUserLevel` field of the data type `stationvariable`. The user who has logged in may change the station variable, if he activates the **Grant** and the `minUserLevel` of the station variable is less than or equal to this user level. Min. Value: 0. Max. Value: 255. **Note** If a user has the permission **Full Access** then the user level is set to **255**.
Variable Reset access `MT_VAR_RESET`	If this has been activated, the user may reset the station variables through the reset button.
Run Menu User Level `MT_MENU_USERLEV`	This **Grant** corresponds to the `minUserLevel` field of the data type `menudata`. The user who has logged in gets the respective set up or service menu entry, if this **Grant** is activated and the `minUserLevel` of the menu entry is less than or equal to this user level. Min. Value: 0, Max. Value: 255. **Note** If a user has the permission **Full Access** , then the user level is set to **255**.
Change cycle settings `MT_CYCLESETTING`	If this has been activated, the user can change the cycle settings.
Testmode access `MT_TESTMODE`	If this has been activated, the test mode may be requested or switched off through the GUI.
Project Manager access `MT_PROJECT_MANAGER`	This will allow the user to access projects by means of the RWMT project view.

12.2 User groups

During the installation of the robotic system, the standard user groups **Operator**, **Service**, and **Programmer** are provided with the following permissions:

Application right	Operator	Service	Programmer
Select Parts MT_SEL_PARTS	X	X	X
Station selection MT_SEL_STATION	X	X	X
GhostMode access MT_GHOSTMODE			X
Gripper control MT_GRIPPER		X	X
Signal write access MT_WRITE_SIGNALS		X	X
Variable Write access MT_VAR_WRITE		100	200
Variable Reset access MT_VAR_RESET	X	X	X
Run Menu User Level MT_MENU_USERLEV	20	100	200
Change cycle settings MT_CYCLESETTING		X	X
Testmode access MT_TESTMODE			X
Project Manager access MT_PROJECT_MANAGER		X	X

Note

If a user has logged in with administrator permissions (**FullAccess**), then he will also have all the permissions of RWMT. Furthermore, the user levels for MT_VAR_WRITE and MT_MENU_USERLEV are set to **255**.

12.3 User level for service menus and change of variable

The service menu and setup menu as well as the station variables make use of the user administration of the robot, to display the service routines or to release the change of variable, for which the user who has logged in has the required permissions.

Through the grants `MT_MENU_USERLEV` and `MT_VAR_WRITE` a user level between 0 and 255 is set, which will be used for releasing the `menudata` or `stationdata`.

Here, the value **255** is equivalent to the grant `Full Access`, that is, all the service routines will be displayed or all the variables will be released for change.

For a menudata declaration to be displayed, the grant `MT_MENU_USERLEV` value of the user should be greater than or equal to the `Min-User-Level` value of the respective `menudata` declaration.

Example:

The following two service routines are declared as service menus:

```
CONST menudata mnuGripperChange:=
["Change gripper","Gripper","station-gripper.png",
     "GripperChange","",3,TRUE,0,1,10];
CONST menudata mnuTC_Unlock:=
["Unlock change
     system","gripper","station-gripper.png","TC_Unlock","",255,FALSE,0,2,100];
```

A user has logged in with the grant `MT_MENU_USERLEV` **20**.

In the service menu, the entry for the **Change gripper** is displayed, since the user level **10** at least is required for the display.

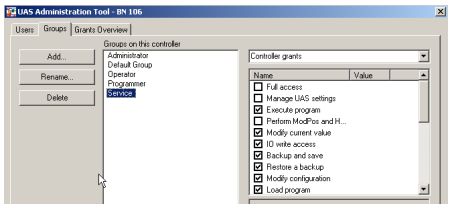
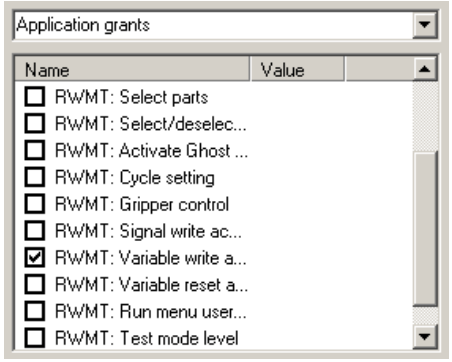
The service routine **Unlock change system** is not displayed, since the required minimum user level is **100**.

A user has logged in with the user level **100**, then all the service routines whose min user level is less than or equal to 100 will be displayed.

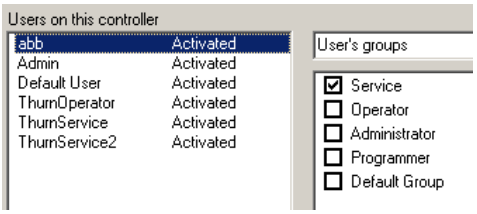
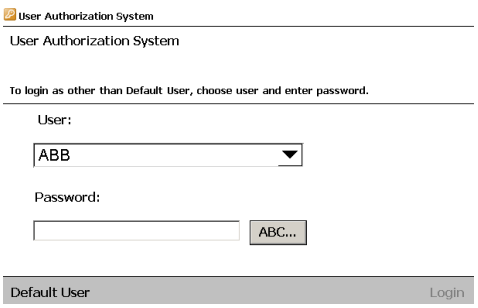
12.4 Setting up the user permissions

The setting up of user permissions is the responsibility of the system operator, who has to assign the passwords as well as the corresponding permissions to the individual users.

To setup the user permissions:

No.	Action	Information
1	Start RobotStudio	 en1300000067
2	Setup connection to robot	 en1200000774
3	Request write access to the robot	 en1200000775
4	Call user administration	 en1200000776  en1200000777
5	Switch to the Groups page and select or create the required group	 en1200000780
6	On the right side, the control permissions will be displayed. Through the combination list field, switch to the **Application permissions**, so that the permissions **RWMT** are displayed.	 en1200000781

Continues on next page

No.	Action	Information
7	If necessary, new groups can be created or the previously described permissions for the groups **Operator**, **Service** or **Programmer** can be changed.	
8	Switch to the **Users** page and add the user with the required permissions.	 en1200000782
9	Log in to the robot under the corresponding user name.	 en1200000783

This page is intentionally left blank

13 Mode of operation of the robot cell

13.1 General

The mode of operation of the robot is not the same as the position of the mode selection switch of the robot controls, but refers to the behavior of the robot program.

Basically, there are three modes of operation in RWMT:

- Operation without robot
- Service
- Production

Depending on the requirement, the mode of operation of the robot cell can be selected in two different ways:

- from outside, through a group input, defined in the **PROC.CFG** (See the chapter *Parameterization of the MT API Commands on page 183*)
- using the button in the graphical user interface (refer to *RWMT- Operating Manual* listed in the section *References on page 11*).

Both, the change from or to the RWMT operation mode **Production** as well as the change from the manual mode of the robot to the automatic mode and viceversa through the key switch plays a role in the execution of the Instruction Sets. These are used for setting output signals or RAPID data to specific values in an automated manner. For more details, see *Instruction sets on page 99*.

If you do not need to differ between the RWMT operation modes, this can be disabled by means of the process configuration section MT API Commands (parameter `no cell operation mode`).

13.2 Operation without robot

Definition

The **Operation without robot** is a mode of operation in which the machines that form the periphery of the robot are used for production, without involving the robot in the production.

This could be the case, for instance, if parts are ejected from the robot cell during the production through a device, and it is then necessary to feed these to the processing line subsequently with the help of the machines surrounding the robot.

During operations without the robot, the robot program can run in theory, but the RWMT Engine cannot load any production routines or execute any other action.

This operation mode can be selected only through the external interface, but not through the graphical user interface.

13.3 Service mode

Definition

The service mode makes it possible to execute setup routines and service routines based on the menudata declarations that are mentioned in the chapters *Service routines on page 81* and *Setup view on page 91*.

Normal production routines cannot be executed if this mode of operation is set.

13.4 Production mode

Definition

In this mode of operation, production routines can be called for the part types that are to be handled and the defined production processes can be loaded. This is usually done on the basis of the partdata declarations that are mentioned in the chapter *Part data on page 63*.

Setup routines or service routines cannot be executed in this mode of operation.

Normal production mode

The normal production mode represents a complete production, including the handling of parts, and possibly with the involvement of different production cycles such as start up cycles and idle run cycles (see the chapter *Program cycles on page 68* .

Production without parts (Ghost mode)

The ghost mode stands for a production process with the usual process logic of the normal production mode, but without any handling of parts.

The ghost mode can be activated only before and after a production cycle. It cannot be activated while processing.

The ghost mode request can be set either through the graphical user interface or by means of the external command interface (see chapter *Parameterization of the MT API Commands on page 183*).

This **Sub-mode** of production is meant for testing the program logic at the time of commissioning, when no parts are available yet for the handling, but processes have to be checked. It is regularly required in automobile assembly lines, for instance.

Whether the gripper is nevertheless made to perform open and close operations or whether the gripper messages can be queried, can be programmed in a flexible manner by using the functionalities provided by RWMT (for this, see *grpdata – Configuration of a control element of the gripper on page 257*).

The instructions and functions *MT_GripSet – Controlling the gripper on page 349*, *MT_GripCheck – Check position of the control element of the gripper on page 326*, *MT_PartCheck – Part controls on the gripper on page 395* and *MT_GhostModeActive – Ask if the ghost mode is active on page 470* support the ghost mode. For more information, see *References on page 11*.

14 Programming

14.1 Introduction

This chapter is meant as a guiding line for the program preparations in RAPID using the MT. The most important steps in the program preparations will be explained here.

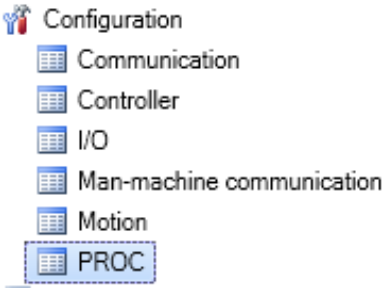
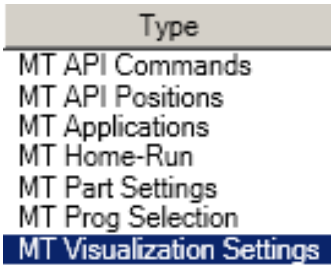
The steps should be executed in the specified sequence.

14.2 Parameterization of the MT Visualization settings

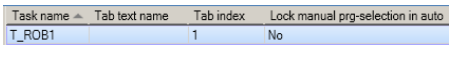
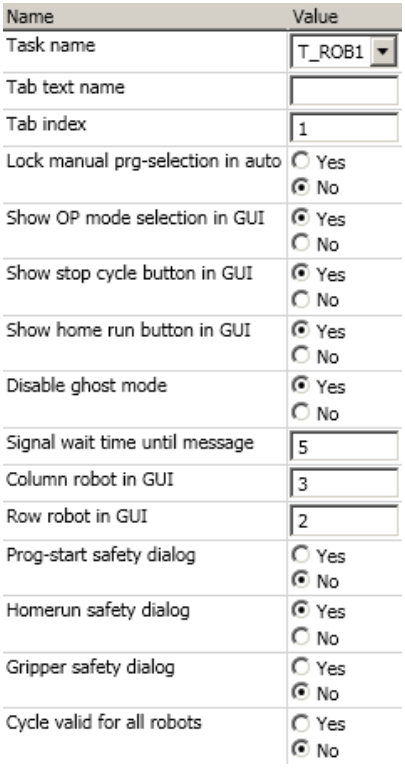
Opening the parameter window

The display parameters influence the appearance and the behavior of the MT user interface.

To enter or modify the individual display parameters, the corresponding parameter window must be opened first in RobotStudio, as shown in the following table.

Procedure in RobotStudio	Explanation
In the Explorer of the robot controls, under **Configuration**, select the **process parameters**.	 en1200000807
In the process parameters window, select the entry **MT Visualization Settings**.	 en1200000808

Continues on next page

Procedure in RobotStudio	Explanation
By double clicking on the parameter row that is displayed, open the parameter window for the **MT Visualization settings** .	 en1200000809  en1200000825  en1200000810

Descriptive text robot

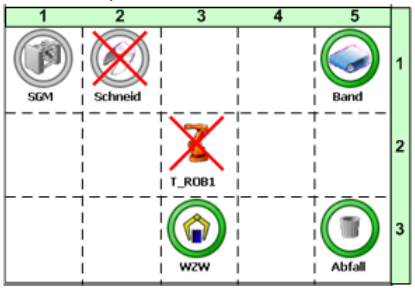
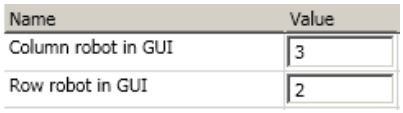
This parameter represents the text under the robot icon in RWMT. In multimove applications it represents the name on the robot tabs.

Explanation	Parameter window
Text that appears under the robot icon in the station view of RWMT. Example:  Robot	 en1200000814

Continues on next page

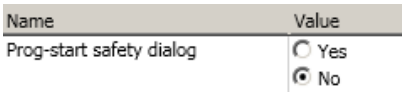
Position of the robot in the GUI

This parameter defines the column or row of the station view of RWMT in which the robot is displayed.

Explanation	Parameter window
Text that appears under the robot icon in the station view of the RWMT. **Example:** Column 3, Row 2  en1200000812	 en1200000813

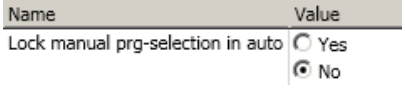
Extra Dialogs

This parameter allows to enable or disable extra dialogs.

Explanation	Parameter window
Disabling of extra confirmation dialogs in the GUI production view, when the start of production, respectively a cycle is selected. The deselection of the extra dialog should only be used for experienced operators.	 en1300000068
Disabling of extra confirmation dialogs in the GUI production view, when the homerun is selected. The deselection of the extra dialog should only be used for experienced operators.	 en1300000069
Disabling of extra confirmation dialogs in the GUI gripper view, when for example, a gripper shall be opened or closed manually. The deselection of the extra dialog should only be used for experienced operators.	 en1300000070

Blocking the manual program selection

If the program selection, that is, the selection of the part type that is to be handled for the production, is done externally through a program number, it may be useful to block the manual selection at the RWMT user interface.

Explanation	Parameter window
Blocking the manual program selection in the user interface (GUI)	 en1200000815

Continues on next page

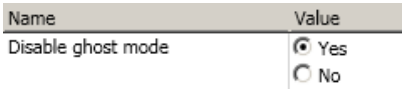
Blocking the manual mode selection

In case the mode of operation (production, service, without robot, see the chapter *Mode of operation of the robot cell on page 173*) is selected externally through the signal interface, it may be useful to block the manual selection at the RWMT user interface.

Explanation	Parameter window
Blocking the manual mode selection in the user interface (GUI)	 en1200000816

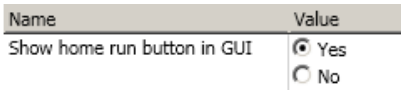
Disable ghost mode

In case the customer does not need a part-less production (ghost mode) for commissioning/test purpose, the selection of this functionality in the RWMT user interface can be disabled.

Explanation	Parameter window
Disabling of the ghost mode (partless production) selection in the GUI. Please note that the ghost mode selection by digital remote signals is possible anyway.	 en1300000071

Display Home Run button

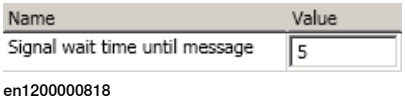
If the request for run-in (start up) of the home position is done externally through the signal interface, it may be useful to the block the manual request at the RWMT user interface (GUI).

Explanation	Parameter window
Display of the Home Run button on the RWMT user interface (GUI)	 en1200000817

Setting the waiting period for message outputs

The waiting period for the state of a signal before a message is displayed by RWMT is specified here.

This refers in particular to the RWMT Wait instructions in accordance with the chapter *Instructions on page 307*.

Explanation	Parameter window
Waiting period while waiting for the state of a signal before a message is displayed.	 en1200000818

Continues on next page

Start all robots in multimove system at the same time

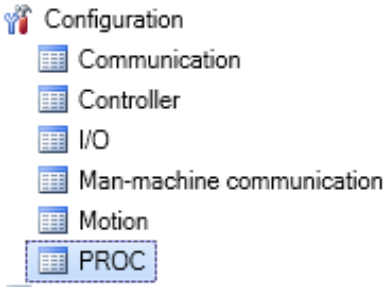
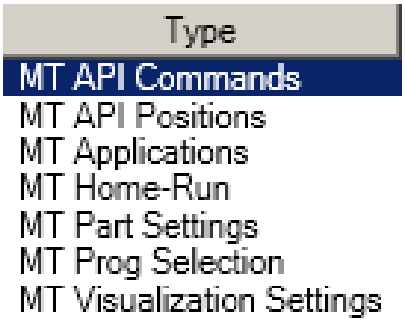
The waiting period while waiting for the state of a signal before a message is displayed by RWMT is specified here.

Explanation	Parameter window
Some cycle types should be selectable by one click for all available motion tasks, so that all tasks start the cycle execution at the same time (cycle synchronization). These cycle types are: • Count cycles • Continuous cycles These are the cycle types where the cycle execution buttons are shown only, if the motion tasks are stopped.	 en1300000072

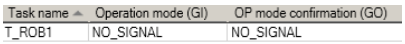
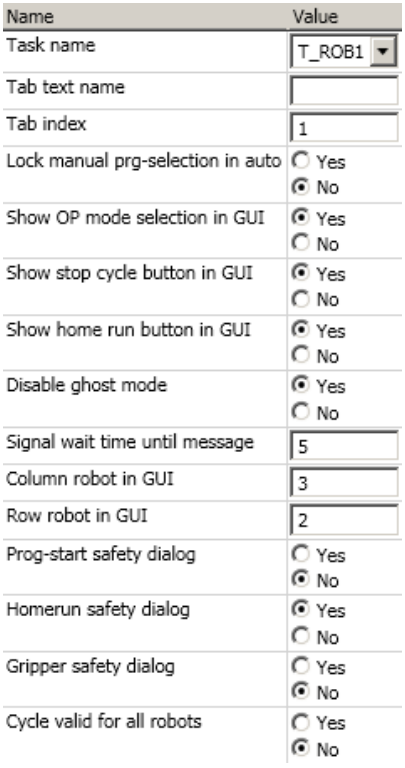
14.3 Parameterization of the MT API Commands

Opening the parameter window

To enter the individual command parameters or to modify them, first the corresponding parameter window should be opened in RobotStudio, as shown in the following table.

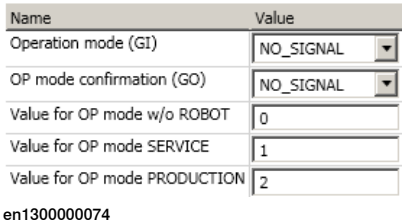
Procedure (in Robot Studio)	Explanation
In the Explorer of the robot controls under **Configuration**, select the **process parameters**.	 en1200000807
In the process parameters window, now select the entry **MT API Commands**.	 en1300000079

Continues on next page

Procedure (in Robot Studio)	Explanation
By double clicking on the parameter row that is displayed, open the parameter window for the **MT API Commands**.	 en1300000073  en1200000825  en1200000810

External pre-selection of mode

With the external mode selection, it is possible to set one of the modes of operation in accordance with the chapter *Mode of operation of the robot cell on page 173* through remote control.

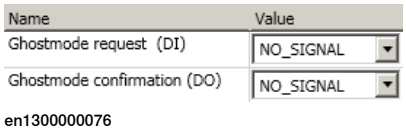
Procedure in Robot Studio	Explanation
The digital group input is mandatory for the remote operated selection of the mode of operation. Optionally, an acknowledgement can be output by the robot controls as digital group output. Further, the values that are expected by the robot controls for the group input modes of operation can be modified now. If the operation mode is changed in the graphical user interface, the group output might be different to the group input.	 en1300000074

Continues on next page

Procedure in Robot Studio	Explanation
Combining the production and service operation mode is possible so that no selection between production and service mode has to be done in the GUI	 en1300000075

Ghost mode

If it should be possible to request a ghost mode of operation in accordance with the chapter *Production mode on page 176* through remote control, then, the corresponding signals must be assigned here as well.

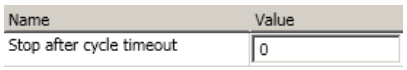
Explanation	Parameter Window
The ghost mode-signals are optional. The digital input represents the request for the ghost mode. The digital output is the acknowledgement from the robot controls, stating that the ghost mode request has been recognized and that the ghost mode has been activated.	 en1300000076

Halt after end of cycle

The **Halt after end of cycle** is normally requested by the operator if the production is to be ended. The robot will complete the handling tasks that are still pending, and then move to the home position.

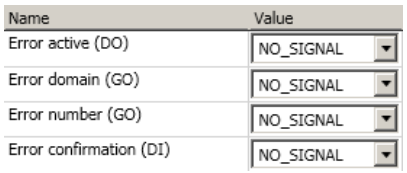
Explanation	Parameter Window
The request **Halt after end of cycle** is generated through a digital input signal.	 en1200000821
Flag to force a direct halt after cycle instead of just requesting it when pressing the halt after cycle button. This is used to simplify the halt after cycle process in the user program.	 en1300000077

Continues on next page

Explanation	Parameter Window
The parameter `stop_after_cycle_timeout` makes mostly sense in applications where startup cycles or emptying cycles are needed and where no direct halt after cycle (see *Preparation of the robot program on page 207*) should be possible. Imagine that stop after cycle request is triggered, but this request cannot be handled by the user program because the engine loops internally (because for example, no valid program number is set). This is solved by a timer which is started if a halt after cycle request is set and after unsuccessfully trying to get a valid program number or request from the GUI to either execute a production or service routine. If a production or service routine can be run before timeout, the timer is stopped and reset. If the timeout is triggered before a production or service routine can be executed, then the **Stop after cycle request** changes automatically to **Stop after cycle reached** and the engine loop is cancelled (program stop). The value **0** [s] disables the timer.	 en1300000078

Communicating the error number

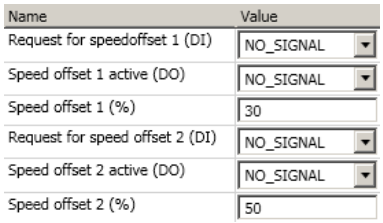
If errors occur in the program run, it may be necessary to report these to a higher order control. This is done by outputting the error notifications with the help of the instruction `MT_ShowMessage` and the data type `msgdata`.

Explanation	Parameter Window
The output signal **Error active** reports that an error that is described by the group signals **Error domain** and **Error number** is pending. Optionally, the digital input **error confirmation** can be used to delete the error output.	 en1200000822

Continues on next page

Speed specifications

Two parameterizable speed specifications can be used to move the robot with a reduced speed instead of with the normal (programmed) speed. This can be used, for instance, after the new part type has been commissioned, to start production cautiously.

Explanation	Parameter Window
Through the digital input signals, one of two reduced speeds can be requested. The digital output signals can be used optionally for a feedback **requested speed is active**. Each of the two reduced speeds must be entered as a % value of the programmed speed.	 en1200000823

If both speed requests are set simultaneously, speed 1 gets the priority.

If both speed requests are set simultaneously and one of both speed requests is reset, then the remaining speed request becomes active.

Pressurized air monitoring

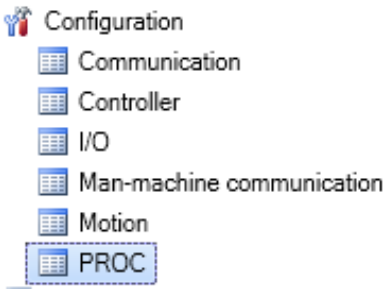
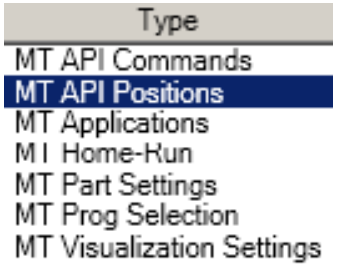
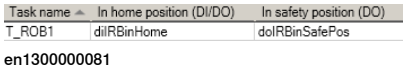
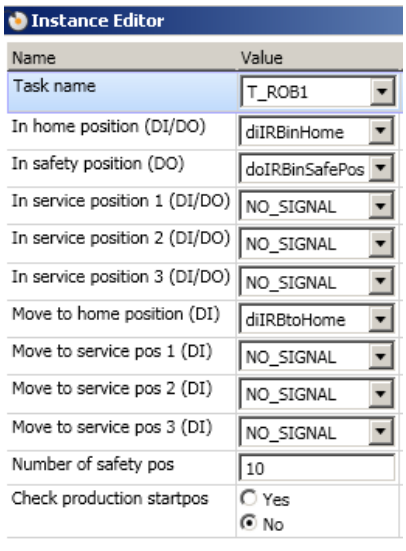
In most handling cells, components such as the robot gripper are operated with pressurized air. Hence, it is useful to monitor these through a pressurized air monitor, which reports the IO-state of the pressurized air to the robot. In case the pressurized air fails, then the robot will abort the production till the air is available again.

Explanation	Parameter Window
Through a digital input, the pressurized air can be monitored.	 en1200000824

14.4 Parameterization of the MT API Positions

Opening the parameter window

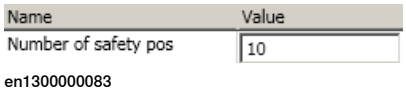
To enter or modify the individual position parameters, the corresponding parameter window must be opened first in RobotStudio, as shown in the following table.

Procedure in Robot Studio	Explanation
In the Explorer of the robot controls, under Configuration, select the process parameters.	 en1200000807
In the process parameters window, select the entry MT API Positions.	 en1300000080
By double clicking on the parameter row that is displayed, open the parameter window for the MT API Positions	 en1300000081  en1200000825  en1300000082

Continues on next page

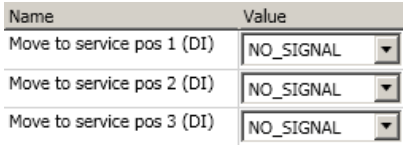
Number of the safety position

The safety position number refers to the use of the concept of dynamic movement calls and the naming conventions as described in the chapter *The program architecture on page 238*. Dynamic calls of movement routines are executed by using the instruction *MT_MoveTo – Dynamic execution of a movement routine on page 391*.

Explanation	Parameter Window
Number of the safe position **Note** The assignment of a position number <>0 may only be done, if there is only one safe-position. Otherwise a collision could happen because when having more than one safe position, the assigned number would be ambiguous.	 en1300000083

Position requests

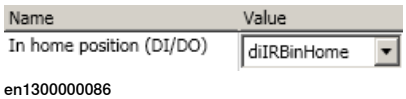
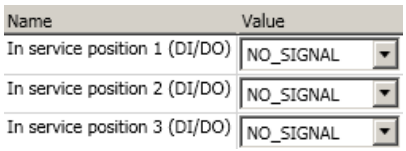
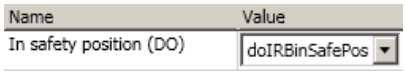
If RWMT is used, the robot can be moved to definite, pre-defined positions as described in the table.

Explanation	Parameter Window
With the help of a digital input signal, the robot can be sent to the home position. For implementing the trip to the home position, it is advisable to use the HomeRun (see the HomeRun chapter). It can be used, for instance, for restoring the initial conditions after an error. After having reached the home position, the program will terminate.	 en1300000084
With the help of a digital input signal, the robot can be send to one of a maximum of three service positions. For the run-in (start up) in one of these positions, RWMT will call the concerned, predefined routine, if requested, in accordance with the chapter *Special service routines on page 234*. To move to a service position, the cell operation mode must be switched to **Service** and the robot has to be located in home position.	 en1300000085

Continues on next page

Position feedback

If the robot is present in one of the pre-defined positions for RWMT, then a feedback must be sent through a digital input or output. Thus, it is mandatory to allocate the parameters described in the following table.

Explanation	Parameter Window
Response stating that the robot is in the home position. For implementing the trip to the home position, it is advisable to use the HomeRun (for this, See chapter HomeRun). This can be used, for instance, for restoring the initial conditions after an error.	 en1300000086
Response stating that the robot is in one of the three possible service positions. For the run-in (start up) of one of these positions, RWMT will call the concerned, predefined routine, if needed, in accordance with the Chapter*Special service routines on page 234*.	 en1300000087
Response stating that the robot is in the safe position (Safety Position). The safe position here normally refers to a position of the robot after a production cycle, which has to be located in an area outside all the machines in a safe area. In some applications, the robot does not need to return to a common safe position, but is allowed to remain in the end position of the previously served station. The robot can start from there into the next cycle. In such a configuration the output **In safe position** has to be set if the robot has reached any appropriate position outside all machines and has to be reset if the robot starts a new cycle.	 en1300000088

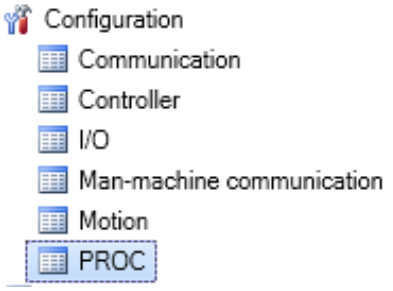
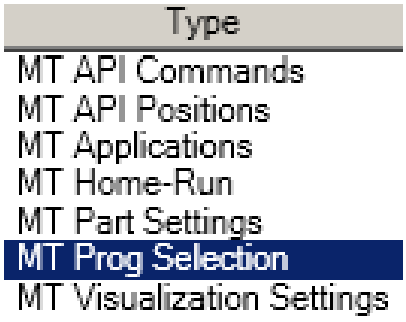
Note

To execute a production cycle, the robot must be either located in the safe position or in the home position.

14.5 Parameterization of the MT Program Selection

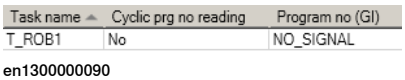
Opening the parameter window

To enter or modify the individual program selection parameters, the corresponding parameter window must be opened first in RobotStudio, as shown in the following table.

Procedure in Robot Studio	Explanation
In the Explorer of the robot controls, under **Configuration**, select the **process parameters**.	 en1200000807
In the process parameters window, now select the entry **MT Program selection**.	 en1300000089

Continues on next page

14.5 Parameterization of the MT Program Selection
Continued

Procedure in Robot Studio	Explanation
By double clicking on the displayed parameter row, open the parameter window for the **MT Program selection**.	 en1300000090  en1200000825  en1300000091

Communicating the program numbers

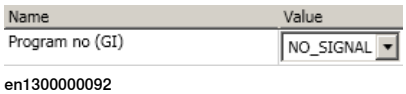
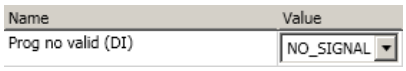
With an interface to the external program number communication, the production process for a specific part type can be selected in a remote controlled manner. Since experience shows that the interfaces for communicating the program numbers differ from customer to customer (for example, handshake signals), the signals that are known from different applications are considered.

The program number that is communicated successfully, will be compared by RWMT with the existing part data for production respectively menu data for service routines (See the chapters *Part data on page 63*, *Service routines on page 81*, and *Setup view on page 91*).

If the program number, which has been transferred, fits the program number of a component and if simulaneously the operation mode is set to ‘production’ then the production routine executed, which is related to the component.

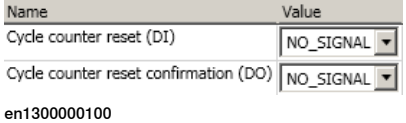
Continues on next page

If the program number, which has been transferred, fits the program number of a service routine and if simulaneously the operation mode is set to ‘service’ then the service routine is executed.

Explanation	Parameter Window
The minimum for communicating the program numbers is a signal group for reading the program number.	 en1300000092
Depending on the other, optional signals that have been selected, RWMT will modify its behavior with respect to the program number communication. If the program number has to be read before every production cycle, then this can be selected under Cyclic prg no reading. Through a group output **Program-No**, the program number that has been read can be reported back for checking. Through the parameter `Program request` it is possible for RWMT to actively request the program number. The higher order controls can communicate through **Prog no. valid** that a program number that is valid for reading is waiting. Through **Prog Confirmation** the robot controls can acknowledge the reading of the program number. The parameter `Prog running` is meant for reporting to the higher order controls that the program number that has been communicated has been accepted for the production and is executed. The signal remains **high** as long as the processing is ongoing.	 en1300000093  en1300000094  en1300000095  en1300000096  en1300000097  en1300000098

Cycle settings

As described in the chapter *Program cycles on page 68*, the production cycles for a part type that is to be handled could differ (for example, Start-up cycles, normal cycles, idle run cycles). For a remote selection of these cycles, RWMT offers the interface that is described below.

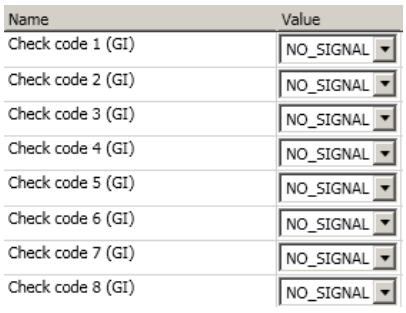
Explanation	Parameter Window
With the help of a group input, the cycle can be pre-selected through remote control	 en1300000099
RWMT manages a cycle counter for the process control. If it is necessary to abort the production due to an error, for instance, then the counter can be reset remotely, to restore the initial state. RWMT can acknowledge this resetting as an option. Alternatively the cycle counter can be reset through the robot station view of the graphical user interface.	 en1300000100

Continues on next page

Tool codes and check codes

For a particular part type that is to be handled, pre-conditions could exist for the production, such as the correct gripper should be clamped to the robot or that a processing machine is fitted with the mold that is necessary for the part.

In the event that the robot grippers or the tools of the processing machines provide a check code, this can be checked by RWMT before starting the production. This is done by comparison with the part that has been selected by the program number (See the chapter *Setting up the graphic user interface (GUI) => Part data* and the data type *partdata – Part data on page 284*).

Explanation	Parameter Window
Up to 8 check codes (test codes) for the system peripherals can be managed by RW-MT.	 en1300000101
The tool code is meant for checking whether the correct gripper has been flanged.	 en1300000102

Execute user defined programs

In RWMT, it is possible to execute the **user defined programs**. This makes it possible for instance, to bypass the RWMT mechanism for communicating the program numbers, if necessary. More information about this is provided in the chapter *User defined programs on page 230*.

Explanation	Parameter Window
If it should be possible to execute **User defined programs** , then the value YES should be set here.	 en1300000103

Time-out while waiting for a program number

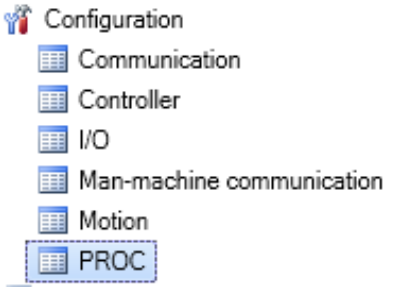
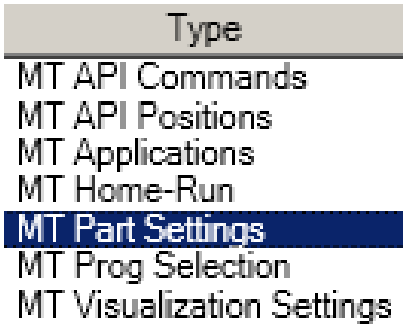
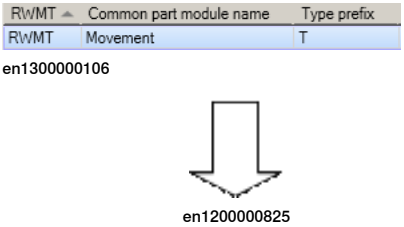
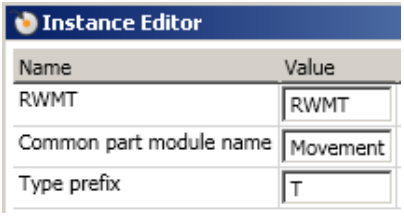
If no (valid) program number is provided by an external plc and no program is selected by the operator on the GUI, the user program cannot execute a **halt after end of cycle** request.

Explanation	Parameter Window
Maximum time in seconds while waiting for a valid program number. After the waiting time has exceeded, the event EE_TIMEOUT_PROG_NUMBER is triggered where the user can force **halt after end of cycle**.	 en1300000104

14.6 Parameterization of the MT part settings

Opening the parameter window

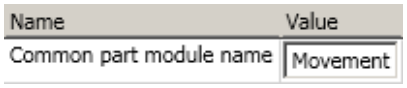
To enter the individual part settings parameters or to modify them, first the corresponding parameter window should be opened in RobotStudio, as shown in the following table.

Procedure in Robot Studio	Explanation
In the Explorer of the robot controls under **Configuration**, select the **process parameters**.	 en1200000807
In the process parameters window, now select the entry **MT Part Settings**.	 en1300000105
By double clicking on the parameter row that is displayed, open the parameter window for the **MT Part Settings**.	 en1300000106 en1200000825  en1300000107

Continues on next page

Part type module

Some of the RWMT RAPID instructions need to know, in which module the part type related movement routines are located. For example, *MT_MoveTo: Dynamic execution of a movement routine*. Please refer to those instructions for further information.

Explanation	Parameter Window
General module name of the part type modules without type prefix and without part type number. For example, if a specific part type module for part type 137 shall be `Movement_T137`, then the value for this parameter must be: **Movement** (without type prefix **T** and without part type number 137).	 en1300000108

Part Type prefix

Some of the RWMT RAPID instructions use part type prefixes. For example, *MT_MoveTo: Dynamic execution of a movement routine*. Please refer to those instructions for further information.

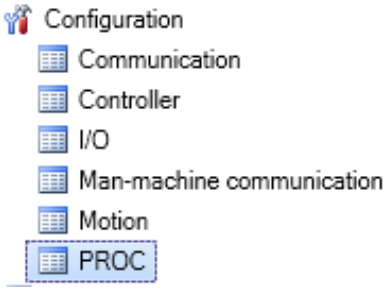
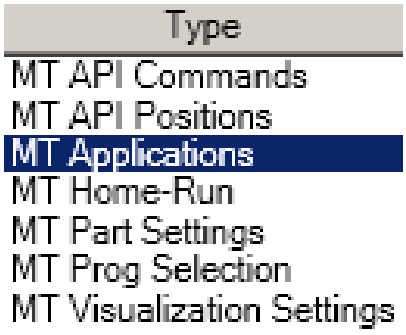
Explanation	Parameter Window
Type prefix to be used for type-dependent movement routines.	 en1300000109

14.7 Parameterization of the MT applications

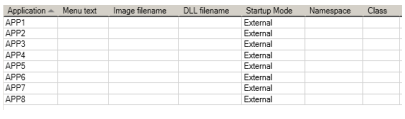
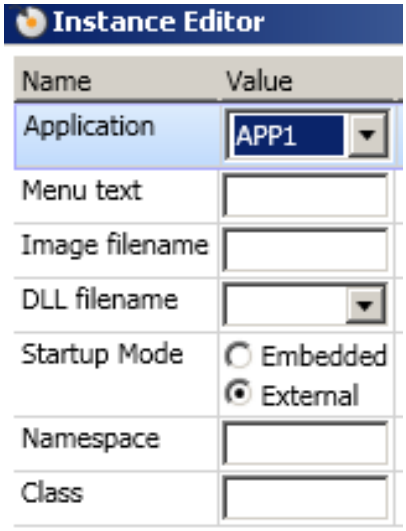
Opening the parameter window

The application parameter settings allow to add FlexPendant applications to the production view of the RWMT GUI, so that those applications can be started directely from there.

To enter or modify the individual application parameters, the corresponding parameter window must be opened first in Robot Studio, as shown in the following table.

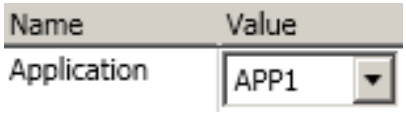
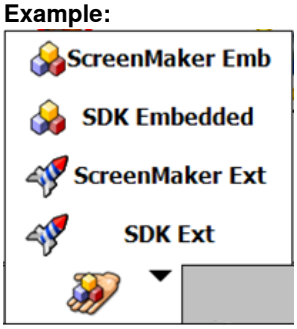
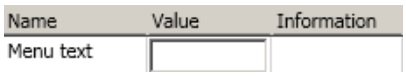
Procedure in RobotStudio	Explanation
In the Explorer of the robot controls, under **Configuration**, select the **process parameters**.	 en1200000807
In the process parameters window, select the entry **MT Applications**.	 en1300000110

Continues on next page

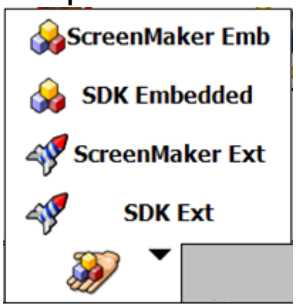
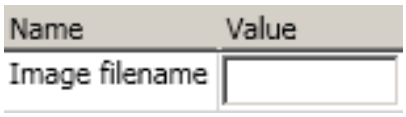
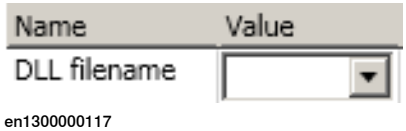
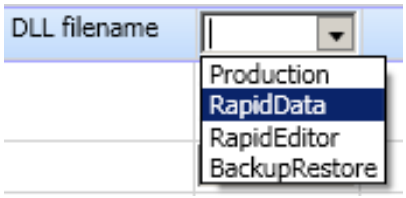
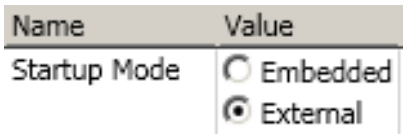
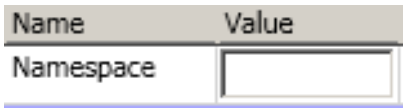
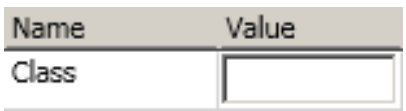
Procedure in RobotStudio	Explanation
By double clicking on one of the parameter rows APP1 – APP8, open the parameter window for the specific MT application settings of the selected application. As shown, up to 8 different applications can be configured. The application definitions in the process configuration are not station dependent. To define applications in the submenu of each station view, please refer to chapter *Setting up the grafic user interface => Stations => Station applications*.	 en1300000111  en1200000825  en1300000112

Declaration of a new FlexPendant application

These parameters define mainly the location of a FlexPendant application, its appearance in the RWMT GUI and its startup behaviour.

Explanation	Parameter Window
Unique name of the application settings, must be called **App1** upto **App10**	 en1300000113
Name of the application, appearing in the pull-down menu of the RWMT GUI. **Example:**  en1200000850	 en1300000115

Continues on next page

Explanation	Parameter Window
File name of the image that will be shown on the left side of the menu entry (max. 32x32 pixel) Menu height depends on the largest image **Example:**  en1200000850	 en1300000116
Name of the application DLL, including the file extender. Some entries are predefined and can be selected to open the following standard FlexPendant views: • Production view • Data editor • RAPID editor • Backup and restore view	 en1300000117  en1300000118
Start mode of the application. Can be either embedded (application is started as child of the RWMT GUI) or as a separate external application	 en1300000119
Namespace of application. (Standard namespace `ABB.Robotics.SDK.Views` is used, if field is empty and app should be launched (not embedded)	
Class (view) of the application that shall be started, not needed for external (not embedded) applications	

Setting example 1: External ScreenMaker application

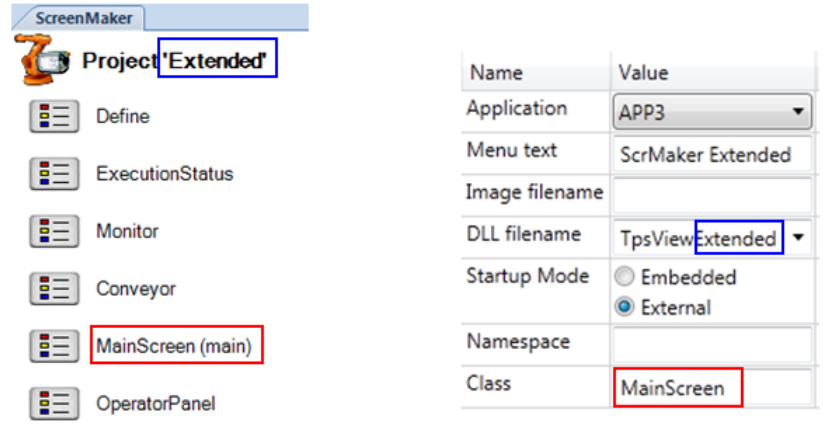
1. Copy ScreenMaker application and image file into home directory
2. Set *menu text*
3. Set `image file name`, for example, image.gif (path is not required)
4. Set DLL file name, use `TpsView+ ScreenMaker-project name`, for example, TpsViewExtended.gtpu.dll. (File extension `gtpu.dll` is not required, will be set automatically)

Continues on next page

5 Select `External` as startup mode

6 Namespace field could remain empty (`Namespace ABB.Robotics.SDK.Views` is used for external application)

7 Set name of the main screen as `class name`.



en1300000122

Setting example 2: Embedded ScreenMaker application

1 Set `menu text`

2 Set `image file name`, for example, `image.gif` (no path required)

3 Set `DLL file name`, use `TpsView + ScreenMaker-project name`, (for example, `TpsViewExtended.dll`. (File extension `.dll` is not required, will be set automatically)

4 Select `Embedded` as startup mode

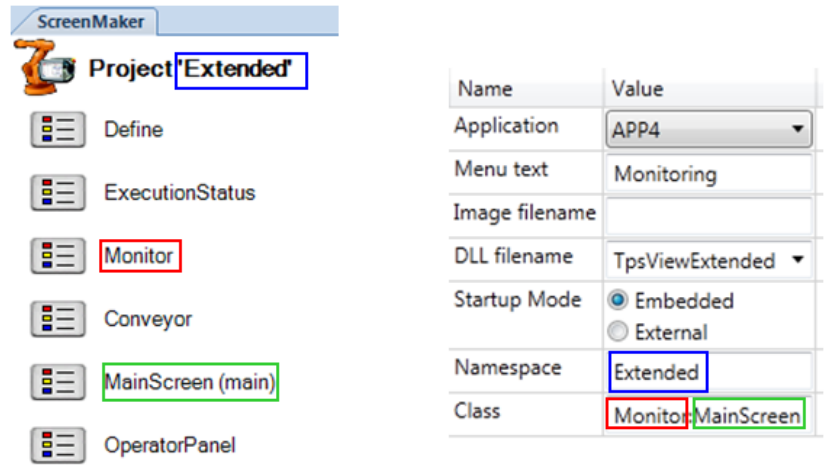
5 Set ScreenMaker-project name as `Namespace`.

6 Set name of the ScreenMaker screen as `class name`, for example, `Monitor`.

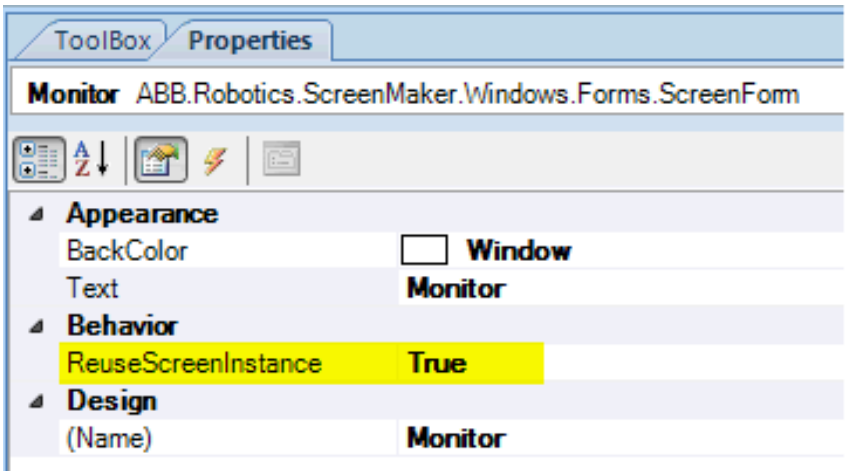
7 Add main screen name, if application variables are used in ScreenMaker project (for example, `Monitor:MainScreen`).

8 In the ScreenMaker view of RobotStudio set the property `ReuseScreenInstance` of the screen which is to be opened to TRUE.

Continues on next page



en1300000123



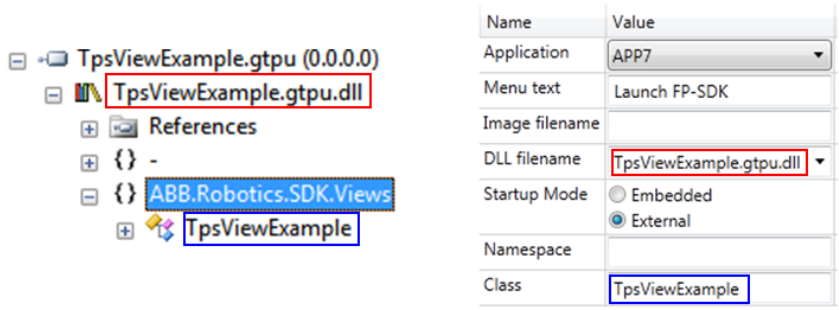
en1300000124

Setting example 3: External Flexpendant SDK application

1. Copy ScreenMaker application and image file into home directory
2. Set `menu text`
3. Set `image file name`, for example, `image.gif` (no path required)
4. Set DLL file name, use `TpsViewExample.gtpu.dll`name, (File extension `.dll` is not required, will be set automatically)
5. Select `External` as startup mode
6. Field `Namespace` could remain empty. (Namespace `ABB.Robotics.SDK.Views` is used for external application
7. Set main screen as `class name`, (for example, `TpsViewExample`. (Ask programmer of the application or use the object browser from Visual Studio).

Continues on next page

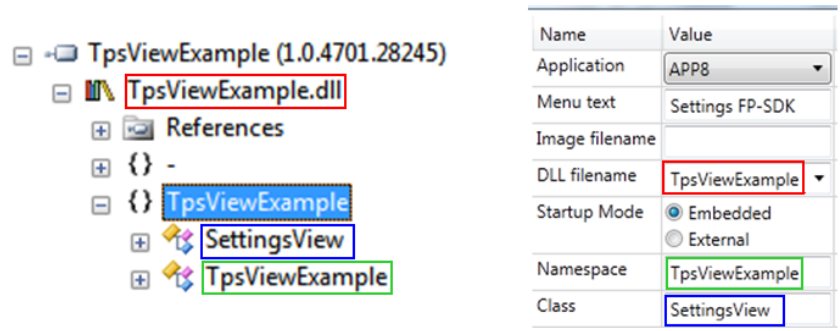
14.7 Parameterization of the MT applications
Continued



en1300000125

Setting example 4: Embedded FlexPendant SDK application

1. Set `menu text`
2. Set `image file name`, for example, `image.gif` (no path required)
3. Set DLL file name, use `TpsViewIRC5App1.dll` . (File extension `.dll` is not required, will be set automatically)
4. Select `Embeddedl` as startup mode
5. Set`Namespace`, please refer to FP-SDK project.
6. Set `class name` of the FP-SDK screen, for example, `SettingsView`.



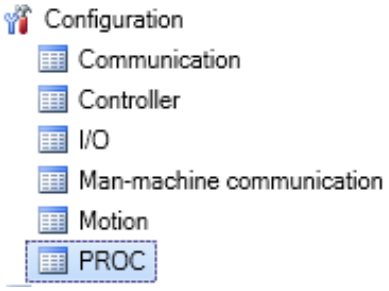
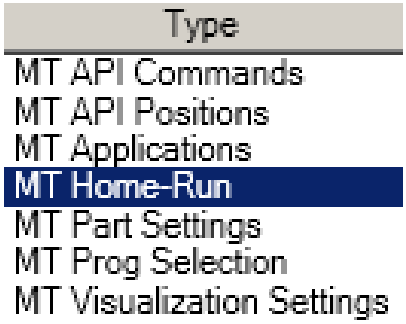
en1300000126

14.8 Parameterization of the MT HomeRun

Opening the parameter window

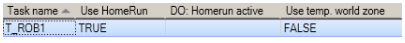
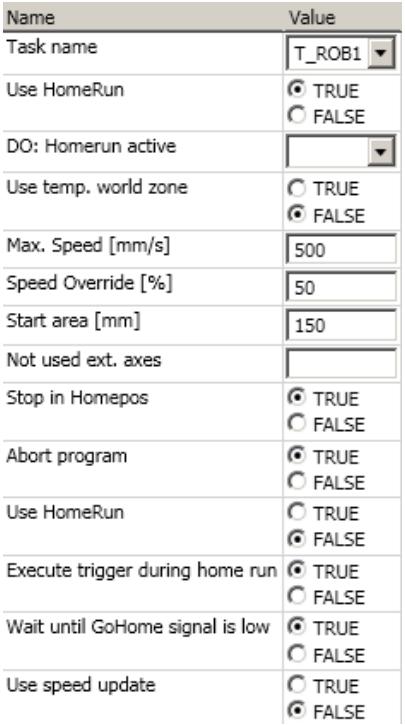
The HomeRun parameter settings allow to configure the HomeRun behaviour as needed for the individual production situation.

To enter or modify the individual HomeRun parameters, the corresponding parameter window must be opened first in Robot Studio, as shown in the following table.

Procedure in RobotStudio	Explanation
In the Explorer of the robot controls, under **Configuration**, select the **process parameters**.	 en1200000807
In the process parameters window, select the entry **MT Applications**.	 en1300000127

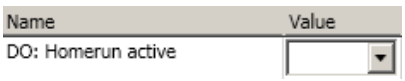
Continues on next page

14.8 Parameterization of the MT HomeRun
Continued

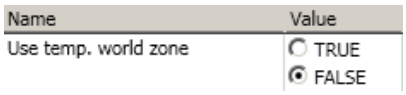
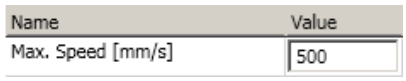
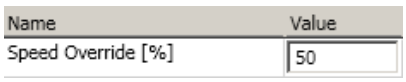
Procedure in RobotStudio	Explanation
By double clicking on the parameter row that is displayed, open the parameter window for the **MT HomeRun** settings.	 en1300000128  en1200000825  en1300000129

HomeRun setings

These parameters define the behaviour of HomeRun while it is executed

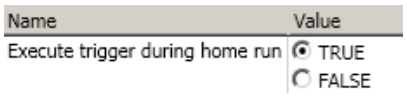
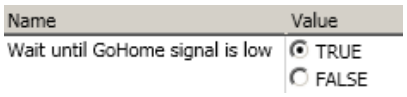
Explanation	Parameter Window
The programmer must take the decision wether to use HomeRun or even not. ABB recommends the usage of HomeRun since it will give a valuable support to each kind of machine tending application.	 en1300000130
Signal for example, to inform a cell controller that HomeRun is active. Active HomeRun means, that the HomeRun functionality has been triggered by the operator and the robot currently tries to go back to the homeposition	 en1300000131

Continues on next page

Explanation	Parameter Window
The purpose of a temporary world zone here is, to set the signal, which indicates, that the robot is in the home position. This parameter should only be used, if the home position is not indicated by a physical home switch. The signal which indicates the home position, has to be assigned in the MT API Positions parameters	 en1300000132
Maximum speed (mm/s) that is set when the move to the home position takes place (Vel-Set). To avoid the damage of the robot and peripheral devices, the maximum speed should be kept in a reasonable range.	 en1300000133
Maximum speed override (%) that is set when the move to the home position takes place. To avoid the damage of the robot and peripheral devices, the maximum speed override should be kept in a reasonable range.	 en1300000134
Maximum permitted distance (mm) that the robot may be moved away from the last automatically moved to position to enable automatic movement into the home position. A value of 0 disables the check. To avoid the damage of the robot and peripheral devices, the check should be enabled.	
External axes number 7-12 which shall not be used for home position verification. This is necessary if an external axis is controlled by an external device, for example, when using one of the options `MachineSync`or `Conveyor tracking`. The axis which is not required will be deactivated by representation of the axis number (7 -12). If several axes shall be deselected for home pos verification, the axis numbers have to be separated by a blank or a comma character, like **7 8 12** or like **7,8,12**.	 en1300000136 **Example**: Deactivation of axis no. 9  en1300000137 **Example:** Deactivation of axes no. 7, 8, 12  en1300000138
The setting of this parameter depends on if the production shall be continued directely after HomeRun execution or not.	 en1300000139
If this functionality is enabled, the operator can abort the program while execution. It should be disabled, if the production shall not be aborted. Then, HomeRun can only be triggered if the program has been stopped before.	 en1300000140

Continues on next page

14.8 Parameterization of the MT HomeRun
Continued

Explanation	Parameter Window
By enabling the **Hold to run** functionality, the operator can interrupt the HomeRun by simply releasing the HomeRun switch. This gives the operator more control while watching the robot going back to home position.	 en1300000141
HomeRun uses the normal movement routines of the production to return to home position. If trigger instructions are used in the movement routines, that shall not be executed while HomeRun, this functionality has to be disbled.	 en1300000142
To be sure that for example, the HomeRun switch works properly, this parameter has to be enabled. It will finish the HomeRun only, if the request signal for HomeRun becomes low (handshake).	 en1300000143
TRUE: The `MT_SpeedUpdate` user routine is called up before executing a movement instruction so that the speed data can be adapted. FALSE: The `MT_SpeedUpdate` user routine is not called up.	 en1300000144

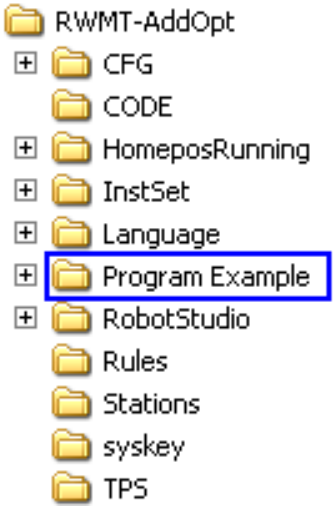
14.9 Preparation of the robot program

14.9.1 Sample programs and templates

Some sample programs that are delivered as part of the additional option can serve as the basis for an application program. It might be more convinient to start with a ready-made program and modify it to fit the individual application.

However, it is also possible to independently design the application program from scratch. In this case, please refer to the program module **MT_Main** as a template.

Both, the sample programs and the **MT_Main** template, can be found in the subfolder **Program example** of the RWMT additional option folder as shown below.



en1300000145

14.9.2 Declarations

Updating the version data

Every program module and system module of the application program should contain version information. This is done be entering a declaration of the type `versiondata`.

Example:

```
MODULE MT_MAIN
…
!Version of this module
CONST versiondata vdMT_MAIN_VERSION:=
["MT_MAIN.MOD","V 1.0","2011-03-30"];
…
ENDMODULE
```

Declare events

RWMT provides a mechanism for calling routines through system events and program events (See the chapter *Event handling on page 93*).

Example:

```
MODULE MT_MAIN
…
!Initialize world zones while warm start
CONST eventdata edPOWER_ON:=
[EE_POWERON,"MT_MAIN:MT_PowerOn",1];
!Initialize application whent starting the robot program
CONST eventdata edUSER_INIT:=
[EE_AFTER_INIT,"MT_MAIN:UserInit",1]; …
…

PROC MT_PowerOn()
!
!Define worldzone for home position
WZSphDef\Inside,shHomePos,p999.trans,50;
!Define which output to be set when located in home
WZDOSet\Stat,wzHomePos\Inside,shHomePos,doIRBinHome,1;
!
ENDPROC


PROC UserInit()
!
!Switch off vacuum gripper
MT_GripSet gsVacuumOff,gdSucker1;
!
…
!
ENDPROC
ENDMODULE
```

Continues on next page

In this example, the Power On routine is called in the MT_Main module as soon as the robot controls executes a hot start or is switched on.

The MT_UserInit routine is called as soon as the interne initialization of RWMT is complete after the `EE_AFTER_INIT` event.

> **Note**
>
> It is necessary to call the initialization routine through an event instead of as the first call in the main()-Routine since the only call in the main()-Routine MT_Execute should be for the Execution-Engine.

Declare part data

The part data declarations in RWMT play an important role, since they are the starting point for executing the production.

The use of part data is the pre-requisite for a meaningful use of RWMT.

Details pertaining to this are given in this manual in the chapter *Part data on page 63*.

Example:

```
MODULE MT_MAIN
…
!Part data example
CONST partdata MT_pdVCylinderHead4718:=
["Part
      4718","MT_Production_T2","",FALSE,4718,2,4,3,[-1,-1,-1,-1,-1,-1,-1,-1],
      "Part4738.GIF",
[1.5,[0,0,0.001],[1,0,0,0],0,0,0],""];

…
PROC MT_Production()
!
…
!
ENDPROC
ENDMODULE
```

The part of the type cylinder head 4718 that is declared in this example has the **Routine MT_Production** as production routine. The part can be selected for production by calling the **program number 4718** remotely or through the RWMT graphic user interface (GUI). In this way, then, the corresponding production routine can be called by RWMT at the start of production.

The part makes use of the concept of production cycles (see the chapter *Program cycles on page 68*).

Apart from this program number, it also has the **type number 2** for internal use in the robot program.

The production can be done with this part only if a robot gripper having the **gripper code 3** has been mounted. Other check codes (test codes) will not be queried.

Continues on next page

14.9.2 Declarations
Continued

Declare cycles

While declaring a part as described in the previous section, it is necessary to define using a flag whether this part works with the cycle concept of RWMT or not.

It is advisable to use production cycles whenever the process of production is not the same each time (Details and examples: see the Chapter *Program cycles on page 68*).

In the following example, the module that is provided with a part declaration in the previous section is extended to include a cycle declaration.

Example:

```
MODULE MT_MAIN
…
!Part data example
CONST partdata MT_pdVCylinderHead4718:=
["Part
     4718","Production","",FALSE,4718,10,4,3,[-1,-1,-1,-1,-1,-1,-1,-1]
     ,
"Part4738.GIF",[1.5,[0,0,0.001],[1,0,0,0],0,0,0],"pdvPart4738"];
…
!Definition of the cycle list
TASK PERS cycledata MT_CycleList{20}:=
[
["Filling","",1,1,10,0,2,0],
["Normal","",2,1,100,0,3,0],
["Emptying","",3.10,1,0,0,0],
["Part Outfeed","",5,2,3,0,0,0],
["","",0,0,0,0,0,0],
["","",0,0,0,0,0,0],
["","",0,0,0,0,0,0],
["","",0,0,0,0,0,0],
["","",0,0,0,0,0,0],
["","",0,0,0,0,0,0],
["","",0,0,0,0,0,0],
["","",0,0,0,0,0,0],
["","",0,0,0,0,0,0],
["","",0,0,0,0,0,0],
["","",0,0,0,0,0,0],
["","",0,0,0,0,0,0],
["","",0,0,0,0,0,0],
["","",0,0,0,0,0,0]
];
PROC Production()
!
!If filling cycle has been selected
IF MT_GetCycleIndex()=1 THEN
…
!If normal cycle has been selected
ELSEIF MT_GetCycleIndex()=2 THEN
```

Continues on next page

```
…
!If emptying cycle has been selected
ELSEIF MT_GetCycleIndex()=3 THEN
…
!If feed out cycle for inspection has been selected
ELSEIF MT_GetCycleIndex()=5 THEN
…
ENDIF
…
!
ENDPROC
ENDMODULE
```

The part data of the type cylinder head 4718 which has been declared in this example will be executed with four different cycles.

First, the filling cycle is executed 10 times, for example, for loading a buffer.

This is followed automatically by the normal cycle for 100 run throughs.

This is again followed automatically by the idle run cycle with 10 executions, for instance, to clear a buffer.

The selected cycle is queried through the RAPID function `MT_GetCycleIndex` (See the chapter *Functions on page 457*)

Declare Instruction Sets

Instruction Sets are meant for setting signals and persistents to a pre-defined value, if there is a change in the mode of operation from or to the RWMT production mode or if the robot operation mode is changed from **manual** to **automatic** or vice versa (For more details, See *Instruction sets on page 99*). Instruction Sets can only be used if the RobotWare option **Multitasking** is present.

Example:

```
MODULE IMM
…
!Flag <Production with robot>
TASK PERS bool IMM_bWithRobot:=TRUE;
…
!Instruction set for change to operation mode
!"with robot"
CONST InstSet IMM_RunWithRobot{10}:=
[
["do","doIRB_OPMode","0","T_ROB1"],
["bool","IMM_bWithRobot","TRUE","T_ROB1"],
["","","",""],["","","",""],["","","",""],
["","","",""],["","","",""],["","","",""],
["","","",""],["","","",""]
];
!Instruction set for change to operation mode
!"without robot"
CONST InstSet IMM_RunWithoutRobot{10}:=
[
```

Continues on next page

14.9.2 Declarations
Continued

```
["do","doIRB_OPMode","1","T_ROB1"],
["bool","IMM_bWithRobot","FALSE","T_ROB1"],
["","","",""],
["","","",""],
["","","",""],
["","","",""],
["","","",""],
["","","",""],
["","","",""],
["","","",""]
];
…
ENDMODULE
```

In the present example, an output signal is set to 0 and a Boolean persistent is set to TRUE if RWMT is changed to the mode of operation **With robot**, that is, the mode of operation for the production.

The output signal will be set to 1 and the Boolean variable will be set to FALSE if RWMT moves out of the **With robot** mode of operation, that is, if the mode of operation changes to production.

Declaring message data

For outputting messages on the programming device, the data and instructions provided by RWMT should be used, so that the messages are displayed on the RWMT user interface and do not overlap this screen. Details of the data declaration and the instructions for the message output are available in the chapter *User program messages on page 43*.

Example:

```
MODULE IMM
…
CONST msgdata IMM_msgEnter:=[10,1,btnNone,
"Waiting for conditions to enter the IMM.",
"","","","","",1,""];
…
PROC IMM_Unload()
…
!Wait for conditions to enter the machine area
MT_WaitMsgDi IMM_sdiMouldOpenPos,high,IMM_msgEnter;
…
ENDPROC
…
ENDMODULE
```

The robot program outputs a message on the RWMT user interface, which is expected for the permission for entering the machine.

Once this release or permission has been granted, the message will be deleted again.

Continues on next page

Declaring station data

Station data enable a visualization on the RWMT graphic user interface (GUI) of the stations that are to be served by the robot, as described in the chapter *Stations on page 44*. See chapter *The station concept on page 235* for the optimum use of the station concept.

Example:

```
MODULE IMM
…
!Base information about this station to be shown in GUI
LOCAL PERS stationdata IMM_Station:=
["IMM","IMM","Injection Moulding Machine",
"station-IMM.png","diIMM_En_OPMode","diIMM_MouldClosed",
"diIMM_Error","",TRUE,FALSE,1,1];
…
ENDMODULE
```

The station data `IMM_Station` maps an injection moulding machine. Through the declaration, a station image is created on the RWMT screen with the image `station-IMM.png` with the caption as IMM .

en1300000146

	The station shows readiness (green ring) for loading on the RWMT user interface, if the digital input signal `diIMM_En_OPMode` is high".
	The station reports **busy** (yellow ring), if the digital input signal `diIMM_MouldClosed` is **high**.
	If the error signal `diIMM_Error` is **high**, then the station will show this as an error (red ring)
	None of the above mentioned states is present (grey ring).

Declaring the station variables

Station variables are variables that play a role in the programming of a specific station (machine). This can be represented, as part of the station view on the RWMT graphic user interface with its values and these could also be manipulated if needed.

The details for this can be obtained from the section *Station variables on page 54*.

Example:

```
MODULE IMM
…
!Base information about this station to be shown in GUI
LOCAL PERS stationdata IMM_Station:=
["IMM","SGM","Injection Moulding Machine",
"station-IMM.png","diIMM_En_OPMode","diIMM_MouldClosed",
```

Continues on next page

14.9.2 Declarations
Continued

```
"diIMM_Error","",TRUE,FALSE,1,1];
!
!Station variables to be monitored or modified in GUI
LOCAL CONST stationvariable IMM_Variables1{5}:=
[
["IMM operation mode with robot","bWithRobot",
"IMM","T_ROB1",0,0,FALSE,FALSE,FALSE,0,1],
["IMM mould has been closed","bMouldClosed",
"IMM","T_ROB1",0,0,FALSE,FALSE,FALSE,0,1],
["Part rejected by IMM","bRejectedPart","IMM",
"T_ROB1",0,0,FALSE,FALSE,FALSE,0,1],
["IMM unloading time","nUnloadTime","IMM",
"T_ROB1",0,99999,TRUE,TRUE,TRUE,0,1],
["IMM condition wait time","ntWaitTime","IMM",
"T_ROB1",0,99999,TRUE,TRUE,TRUE,0,1]];
…
ENDMODULE
```

The declaration shows 5 variables, which will be displayed in RWMT if the station symbol of the injection moulding machine is selected.

The values of the variables `bWithRobot`, `bMouldClosed` and `bRejectedPart` cannot be edited here, the values of the variables `nUnloadTime` and `ntWaitTime` can be modified both in the automatic mode as well as in the set up mode between *0* and*99999* .

Declaring station signals

Station signals are signals that play a role in the programming of a specific station (machine). This can be represented, under certain circumstances, as part of the station view on the RWMT graphic user interface with its current states, and these states could also be manipulated if necessary.

The details pertaining to this can be obtained from the Chapter *Station variables on page 54*.

Example:

```
MODULE IMM
…
!Digital inputs of IMM
VAR signaldi IMM_sdiMouldOpenPos;
VAR signaldi IMM_sdiEjec_BackPos;
VAR signaldi IMM_sdiEjec_ForwPos;
VAR signaldi IMM_sdiCorePullPos1;
VAR signaldi IMM_sdiCorePullPos2;
VAR signaldi IMM_sdiReject;
VAR signaldi IMM_sdiEn_OPMode;
VAR signaldi IMM_sdiMouldClosed;
VAR signaldi IMM_sdiInterMouldPos;
VAR signaldi IMM_sdiNoPartAvaible;
!
!Digital outputs of IMM
VAR signaldo IMM_sdoEn_Mould;
VAR signaldo IMM_sdoIRB_OPMode;
```

Continues on next page

```
VAR signaldo IMM_sdoEn_EjecBack;
VAR signaldo IMM_sdoEn_EjecForw;
VAR signaldo IMM_sdoEn_CPullPos2;
VAR signaldo IMM_sdoEn_CPullPos1;
VAR signaldo IMM_sdoEn_FMouldOp;
VAR signaldo IMM_sdoMouldAreaFree;


!Base information about this station to be shown in GUI
LOCAL PERS stationdata IMM_Station:=
["IMM","SGM","Injection Moulding Machine",
"station-IMM.png","diIMM_En_OPMode","diIMM_MouldClosed",
"diIMM_Error","",TRUE,FALSE,1,1];
!
!IO mappings for signals / alias signals
LOCAL CONST stationsignal IMM_Signals{18}:=
[
["Mould is open","diMouldOpenPos","IMM_sdiMouldOpenPos"],
["Ejectors back","diEjec_BackPos","IMM_sdiEjec_BackPos"],
["Ejectors fwd","diEjec_ForwPos","IMM_sdiEjec_ForwPos"],
["Core pullers pos 1","diCorePullPos1","IMM_sdiCorePullPos1"],
["Core pullers pos 2","diCorePullPos2","IMM_sdiCorePullPos2"],
["Reject","diReject","IMM_sdiReject"],
["Automatic mode","diEn_OPMode","IMM_sdiEn_OPMode"],
["Mould is closed","diMouldClosed","IMM_sdiMouldClosed"],
["Mould inter. pos","diInterMouldPos","IMM_sdiInterMouldPos"],
["No part available","diNoPartAvaible","IMM_sdiNoPartAvaible"],
["Start IMM","doEn_Mould","IMM_sdoEn_Mould"],
["IRB operation mode","doIRB_OPMode","IMM_sdoIRB_OPMode"],
["retrace ejectors","doEn_EjecBack","IMM_sdoEn_EjecBack"],
["Push ejectors","doEn_EjecForw","IMM_sdoEn_EjecForw"],
["Core pullers pos 2","doEn_CPullPos2","IMM_sdoEn_CPullPos2"],
["Core pullers pos 1","doEn_CPullPos1","IMM_sdoEn_CPullPos1"],
["Enable mould opening","doEn_FMouldOp","IMM_sdoEn_FMouldOp"],
["Mould area is free","doMouldAreaFree","IMM_sdoMouldAreaFree"]
];
```

For each of the digital interface signals, an alias signal has been declared in the example. The assignment of an alias signal is not mandatory. It is just a possibility to standardize the program code, independent from customized naming of physical signals.

The alias signals are automatically linked with the physical signals through the stationsignal declaration by RWMT, if the instruction MT_AliasIO is called with a stationsignal argument.

The physical signals will also be displayed in RWMT, if the station symbol of the injection moulding machine is selected.

Thus, the `stationsignal` declaration has two functions:

- Displaying the signals in the GUI
- Assignment of the physical signals to the alias signals

Continues on next page

If one wishes to display only the signals in the GUI and not use the alias signals, then these should be omitted simply in the `stationsignal` declaration.

HotEdit declarations

With the HotEdit declarations, a filter can be defined for using the standard HotEdit, so that only selected robot positions such as positions from a specific machine are displayed. More information is available in the chapter *Advanced HotEdit on page 84*.

Example:

```
MODULE IMM
…
!Hotedit position declarations
LOCAL CONST hoteditdata MT_he_IMM{1}:=
[
["IMM","station-IMM.png","Movement","mv101_102","p102","",""]
];
…
ENDMODULE

MODULE Movement
…
!Hotedit position declarations
CONST robtarget p102:=…;

!Movement routine from start position of IMM
!to enter position of IMM
PROC mv101_102
MoveJ p101.v1000.z10.tGripper;
MoveL p102,v1000.fine,tGripper;
ENDPROC
…
ENDMODULE
```

The `hoteditdata` declaration in the IMM module creates an entry in the HotEdit-menu of the RWMT user interface.

If this entry is selected, then the Standard-HotEdit will be loaded with the sole position `p102`.

14.9.3 Program initialization

Initialization through the RWMT event handling

Normally, the initialization is first done through a handling program on start from main() , that is, variables and digital output signals, for instance, will be reset to a pre-defined value.

This process of initialization cannot take place in programs with RWMT implementation, because there must not be something added to the main routine, excepting the call of the RWMT engine and a stop instruction. Otherwise it cannot be guaranteed, that the program executes safely, because some of the RWMT features have to be initialized internally, before they are allowed to be used.

```
PROC main()
!
!Call the execution engine
MT_Execute;
Stop;
!
ENDPROC
```

To carry out an initialization nevertheless, the initialization routine must be called through the event mechanism that is provided by RWMT, as shown in the following example:

```
MODULE MT_MAIN
…
!Initialize application whent starting the robot program
CONST eventdata edUSER_INIT:=
[EE_AFTER_INIT,"MT_MAIN:UserInit","",1]; …
…
PROC UserInit()
!Switch off vacuum gripper
MT_GripSet gsVacuumOff,gdSucker1;
!Reset part load
GripLoad load0;
!
ENDPROC

ENDMODULE
```

More information about events is available in the chapter *Event handling on page 93*.

14.9.4 Design of the production routines

Call (load)

After the initialization on starting from main(), a handling program will request a program number if there are different part types or production flows to be considered. Then it will call a production routine in cyclic manner. The production routine describes in which order the different machines inside a production cell shall be served.

RWMT cares about the transfer of the program number as well as for the call of the depending production routine.

The production routine is called automatically through a `partdata` declaration that should be envisaged. The following condition must be fulfilled if the corresponding production routine is to be called:

- The partdata declaration contains the name of the production routine
- The tool codes and check codes (test codes), if present, are ok
- The production was started at the GUI or by remote control with the correct program number of the partdata declaration.

Example:

```
MODULE MT_MAIN
…
!Part data example
CONST partdata pdVCylinderHead:=
["Part
      4718","Production","",FALSE,2,10,2,3,[-1,-1,-1,-1,-1,-1,-1,-1],"Part1.GIF",
[1.5,[0,0,0.001],[1,0,0,0],0,0,0],"pdvPart1"];

…

PROC Production()
!
…
!
ENDPROC

ENDMODULE
```

Querying of cycles in the production routine

In the simplest case, a partdata declaration without additional cycle declaration is present. In this case, there is no need for RWMT to query any cycle information in the production routine.

Example:

```
MODULE MT_MAIN
…
!Part data example
TASK PERS partdata pdVCylinderHead4718:=
```

Continues on next page

```
["Part
     4718","Production","",TRUE,2,10,2,3,[-1,-1,-1,-1,-1,-1,-1,-1],
     "Part4738.GIF",
[1.5,[0,0,0.001],[1,0,0,0],0,0,0],"pdvPart4738"];

…

PROC Production()
…
UnloadMachine1;
LoadMachine2;
UnloadMachine2;
LoadConveyor;
…
ENDPROC
…
ENDMODULE
```

On the other hand, if a partdata declaration with additional cycle declaration is present, then, must in the production routine, this cycle information should be queried through the `MT_GetCycleIndex ()` function for calling the correct production cycle.

Example:

```
MODULE MT_MAIN
…
!Part data example
TASK PERS partdata pdVCylinderHead:=
["Part
     4718","Production","",FALSE,2,10,4,3,[-1,-1,-1,-1,-1,-1,-1,-1],
     "Part1.GIF",
[1.5,[0,0,0.001],[1,0,0,0],0,0,0],"pdvPart1"];
…
!Definition of the cycle list
TASK PERS cycledata MT_CycleList{20}:=
[
["Filling cycle","",1,1,10,0,2,0],
["Normal cycle","",2,1,100,0,3,0],
["Emptying cycle","",3,10,1,0,0,0],
["Part Outfeed","",5,2,3,0,0,0],
["","",0,0,0,0,0,0],["","",0,0,0,0,0,0],
["","",0,0,0,0,0,0],["","",0,0,0,0,0,0],
["","",0,0,0,0,0,0],["","",0,0,0,0,0,0],
["","",0,0,0,0,0,0],["","",0,0,0,0,0,0],
["","",0,0,0,0,0,0],["","",0,0,0,0,0,0],
["","",0,0,0,0,0,0],["","",0,0,0,0,0,0],
["","",0,0,0,0,0,0],["","",0,0,0,0,0,0]
];
PROC Production()
…
…
```

Continues on next page

14.9.4 Design of the production routines
Continued

```
!If filling cycle has been selected
IF MT_GetCycleIndex()=1 THEN
FillingCycle;
!If normal cycle has been selected
ELSEIF MT_GetCycleIndex()=2 THEN
NormalCycle;
!If emptying cycle has been selected
ELSEIF MT_GetCycleIndex()=3 THEN
EmptyingCycle;
!If feed out cycle for inspection has been selected
ELSEIF MT_GetCycleIndex()=5 THEN
FeedOutCycle;

…
ENDPROC

PROC FillingCycle()
…
UnloadInfeedConveyor;
LoadBuffer;
…
ENDPROC

PROC NormalCycle()
…
UnloadBuffer;
LoadMachine;
UnloadInfeedConveyor;
LoadBuffer;
UnloadMachine;
LoadOutfeedConveyor;

…
ENDPROC

PROC EmptyingCycle()
…
UnloadBuffer;
LoadMachine;
UnloadMachine;
LoadOutfeedConveyor;
…
ENDPROC

PROC FeedOutCycle()
…
UnloadBuffer;
LoadMachine;
UnloadMachine;
LoadInspectionConveyor;
…
```

Continues on next page

```
ENDPROC
…
ENDMODULE
```

14.9.5 Halt after end of cycle

Triggering

Halt after end of cycle is normally requested by the operator if the production is to be ended. The robot will complete the handling tasks that are still pending, and then move to the home position.

The triggering takes place through a request on the graphic user interface or by a request from outside, through a digital input signal (for this, see the chapter *Parameterization of the MT API Commands on page 183*).

Evaluation in the robot program

The evaluation in the robot program includes the following points in the case of a **Halt after end of cycle** request in RWMT:

- Recognition of the request
- Executing the necessary processes before the end of production
- Acknowledgement of the request at the end of production

For recognizing the request **Halt after end of cycle**, the `MT_EndOfCycleReq()` function is available in RWMT.

With the help of the instruction `MT_EndOfCycleAck`, the request can be acknowledged. The RWMT Engine recognizes this acknowledgement after quitting the production routine that has been called through the partdata declaration and stops the execution of the production.

Furthermore, `MT_EndOfCycleOk()` provides an instruction for querying whether or not **Halt after end of cycle** has been acknowledged already in the previous production cycle.

Halt after end of cycle while Production is running

In many cases, it is enough to recognize a **Halt after end of cycle** request at the end of a production cycle an to stop the program execution immediately.

Example 1:

```
PROC MT_Production()
...
!If stop after cycle has been requested
IF MT_EndOfCycleReq() THEN
!Move to home position
MoveTo 999;
!Acknowledge stop after cycle to end production
MT_EndOfCycleAck;
ENDIF
!
ENDPROC
```

This is mostly the case if the request **Halt after end of cycle** is not to be followed by a run-out cycle where emtpying of machines or buffers is necessary.

In many other cases, the program cannot be terminated directly since for example, machines or buffers have to be emptied before.

Continues on next page

In those cases, the function `MT_EndOfCycle` is used to start the run-out cycles.

Example 2:

```
PROC Select()
!
!call production routine
Production;
!
!if end of cycle request has been acknowledged
IF MT_EndOfCycleReqOK() THEN
!move to home position
MoveTo 999;
ENDIF
!
ENDPROC

PROC Production()
!
!if stop after cycle has not been requested
IF MT_EndOfCycleReq()=FALSE THEN
!unload the machine
UnloadMachine;
!if the part quality is ok
IF bPartControlOK=TRUE THEN
!start next machine cycle
StartMachine1;
!Place part at conveyor
loadConveyor;
If the part quality is not ok
ELSE
!put the part to the scrap box
LoadScrapBox;
ENDIF
!if stop after cycle has been requested
ELSE
!unload the machine
UnloadMachine;
!if the part quality is ok
IF bPartControlOK=TRUE THEN
!Place part at conveyor
LoadConveyor;
If the part quality is not ok
ELSE
!put the part to the scrap box
LoadScrapBox;
ENDIF
!confirm stop after cycle
MT_EndOfCycleAck;
ENDIF
!
ENDPROC
```

Continues on next page

14.9.5 Halt after end of cycle
Continued

Stop after cycle while waiting for an order

If the robot does not execute a production routine but currently waits for a new program number, the above mentionned strategies are not useful since the program pointer is located inside the RWMT engine and cannot execute any user code.

In this case, the request for **Halt after end of cycle** can be triggered by means of the event EE_WAIT_ORDER. This event is executed if RWMT waits for a new order (program number).

Example:

```
MODULE MT_MAIN
…
!Initialize world zones while warm start
CONST eventdata edWaitOrder:=
[EE_WAIT_ORDER,"MT_MAIN:StopAfterCycle","",1];
…
…
PROC StopAfterCycle()
!
!confirm stop after cycle directely
!when request is present
If MT_EndOfCycleReq()MT_EndOfCycleAck;
!
ENDPROC
…
…

ENDMODULE
```

14.9.6 Error handling and return to the home position

With HomeRun

If an error occurs in the program run, then, it may be necessary to send the robot back to the home position.

If RWMT is run with HomeRun, then the instructions that are provided by this functionality can be used (refer to the chapter, *HomeRun on page 105*).

> **Note**
>
> ABB explicitly recommends the use of HomeRun for the safe return to the home position. If HomeRun is not used, then the following request signals for the return to the home position must be evaluated by the Integrator in the application program itself:
>
> - Parameter `DI_GOHOME` (see the chapter *MT API Positions on page 157*)
> - Digital outputs `doT_ROB1_GoHome` to `doT_ROB4_GoHome`, that have been set by RWMT on requesting the home position in the user interface.

Without HomeRun, regular error handling

If HomeRun shall not be used so the next appropriate but less comfortable way to bring the robot back to home position is to use the normal error handling.

For this purpose, the loading and unloading routines contain their own error handling section to handle errors inside the station. and raise them to the calling routines through using the RAISE instruction.

Example:

```
MODULE MT_MAIN
!Part data example
TASK PERS partdata MT_pdVCylinderHead4718:=
["Part
      4718","Production","",TRUE,2,10,4,3,[-1,-1,-1,-1,-1,-1,-1,-1],"Part4738.GIF",
[1.5,[0,0,0.001],[1,0,0,0],0,0,0],"pdvPart4738"];
…
PROC Production()
!unload 1st machine
UnloadMachine1;
!load 2nd machine
LoadMachine1;
ERROR
IF ERRNO= ERR_MT_ABORT THEN
!force end of cycle
MT_EndOfCycleAck;
!jump to error handler of RWMT engine
RAISE;
ENDIF
ENDPROC
PROC UnloadMachine1()
VAR bool bTimeOut;
```

Continues on next page

14.9.6 Error handling and return to the home position
Continued

```
!move to preposition outside machine 1
MoveTo 10;
!wait for release or timeout
WaitUntil diReleaseMachine1=high\MaxTime:=5
\TimeFlag:= bTimeOut;
!release has not been given => jump to error handler
IF bTimeOut RAISE ERR MT ABORT;
…
!do unloading here
…
ERROR
!jump to error handler of calling routine
IF ERRNO= ERR MT ABORT RAISE;
ENDPROC


PROC LoadMachine2()
VAR bool bTimeOut;

!move to preposition outside machine 2
MoveTo 20;
!wait for release or timeout
WaitUntil diReleaseMachine2=high\MaxTime:=5
\TimeFlag:= bTimeOut;
!release has not been given => jump to error handler
IF bTimeOut RAISE ERR MT ABORT;
…
!do loading here
…
ERROR
IF ERRNO= ERR MT ABORT THEN
!move to scrap box
LoadScrapBox;
!jump to error handler of calling routine
RAISE;
ENDIF
ENDPROC
ENDMODULE
```

Note

RWMT provides an own error number ERR_MT_ABORT for this purpose

This error number can be used to jump from each desired position inside the user program to the error handler of the RWMT engine. If no **Halt after end of cycle** has been triggered before, the production continues from the beginning.

The robot must be sent to *home position* (see *Strategy for automatic movement into the home position on page 134*) or *safe position* (see *Movement continuation*

Continues on next page

in intermediate positions on page 142) before jumping to the error handler of the RWMT engine because otherwise the production cannot be continued afterwards.

Further information regarding error handling concepts can be found in the *Technical Reference Manual RAPID - Overview* listed in the section *References on page 11*.

Without HomeRun, simple error handling

If no HomeRun and no regular error handling shall be used, there is another very simple mechanism of RWMT to send the robot back to home position.

Therefor the user program triggers an error through a `RAISE` instruction with error number `ERR_MT_HOMERUN`. As a result, the program pointer jumps back into the RWMT engine.

The engine sets the status **Halt after end of cycle reached** and calls the routine `MT_HomeDirect`. This routine is to be provided by the integrator in the user program and must be filled with appropriate instructions to move the robot back to home position.

After execution of this routine, the program terminates.

Example:

```
MODULE MT_MAIN
…
!Part data example
TASK PERS partdata MT_pdVCylinderHead4718:=
["Part
     4718","Production","",TRUE,2,10,4,3,[-1,-1,-1,-1,-1,-1,-1,-1],"Part4738.GIF",
[1.5,[0,0,0.001],[1,0,0,0],0,0,0],"pdvPart4738"];
…

PROC Production()
…
…
IF diReleaseMachine=0 THEN
!Trigger error handling to go back to home position
RAISE ERR_MT_HOME_RUN;
ENDIF
…
ERROR
RAISE;
ENDPROC

PROC MT_HomeDirect()
…
!Insert here your code to go to homeposition
!after you have raised with error number
!ERR_MT_HOME_RUN
…
ENDPROC
…
ENDMODULE
```

Continues on next page

14.9.6 Error handling and return to the home position
Continued

Note

ABB does not suggest to use this alternative because in a central routine, it cannot be clearly said where the robot is physically. So a danger of collision remains.

However this alternative is useful, if the way back to home position is always the same, which sometimes is the case in very simple applications.

ABB suggests the use of *HomeRun on page 105* or to implement *Error handling and return to the home position on page 225*.

14.9.7 Change of tools

Triggering and handling in the program

If a gripper replacement is necessary, because different products have to be handled and if robot gripper tools are operated with a coding, then RWMT can compare the current coding with the specification for the product by using the partdata declaration of the current product.

If there is a difference between the current gripper coding and the gripper code for the chosen product, it is possible that one has forgotten to re-equip the gripper.

In this case, RWMT calls a `MT_ChangeTool` routine instead of the production routine, which should be provided for by the application programmer and with equipped with suitable instructions for a gripper replacement.

In this routine, RWMT hands over the code of the current gripper tool as the first argument and the code of the gripper that is to be used as the second argument.

After calling the `MT_ChangeTool` routine, the RWMT Engine will once again check if the current gripper code matches with the specification. If this is not yet the case, perhaps because the routine does not exist or the gripper has not been changed, then RWMT will output an error message. In this case, the production routine will no be executed.

Routine to be provided:

```
PROC MT_ChangeTool(
VAR signalgi CurrToolCode,
num ReqToolCode)
!
!Insert your tool changing instructions here
!
ENDPROC
```

14.9.8 User defined programs

Usage and conventions

In some cases, it may be necessary to execute production routines in a user defined manner. This happens, for instance, if

- the mechanism that has been integrated in RWMT for communicating the program numbers should or can not be used or
- if both the mechanism that has been integrated with RWMT for communicating the program numbers as well as other RWMT functionalities such as cycle handling and check code monitoring should not be used.

For these cases, RWMT provides the option for calling the user defined routine MT_GetUserProgNo, where the user specific program number transfer is done and where, if required, the current production routine can be selected.

In order that this routine can be executed by RWMT, this should be released first in the process parameters (see chapter *Parameterization of the MT Program Selection on page 191*).

The **user defined routine** must be provided for by the programmer in the application program and should be in accordance with the following convention:

Convention	Value	Explanation
Name	`MT_GetUserProgNo`	The user defined routine is called by RWMT under its defined name.
Argument1	`INOUT dnum ProgNo`	Argument that can be described in the routine for the program number that is to be evaluated by RWMT. Value allocation <=0 is necessary, to prevent the argument from being evaluated.
Argument 2	`INOUT string Routine`	Argument that can be described in the routine for the name of the routine that is to be called by RWMT. Value allocation "" (blank string) necessary, to prevent the argument from being evaluated.

Within the `MT_GetUserProgNo` routine, only one of the two arguments `ProgNo` or `Routine` may be used. If both are used, then only the first argument `ProgNo` will be evaluated.

Handling in the program

If the argument ProgNo is used, then RWMT will search for a partdata declaration, which contains the corresponding program number and will execute the production routine that is defined in the declaration.

In this case, only the RWMT mechanism for reading a program number will be bypassed. All the other mechanisms such as the handling of cycles as well as the evaluation of check codes (test codes) will be executed.

Example 1:

Continues on next page

In the following design of the `MT_GetUserProgNo` routine, the RWMT searches for a `partdata` declaration with the program number 7 and the corresponding production routine will be called.

```
PROC MT_GetUserProgNo(
INOUT dnum ProgNo,
INOUT string Routine)
!
!Program number 7 assigned
ProgNo:=7;
!
ENDPROC
```

Example 2 :

In this example, the routine `MT_GetUserProgNo` is used to build an individual handshake for data transfer. This might be useful if the handshake signals, provided by RWMT, do not fit the requirements.

```
PROC MT_GetUserProgNo(
INOUT dnum ProgNo,
INOUT string Routine)
!
!Program number assigned by serial interface
ProgNo:=GetProgNoFromRS232();
!
ENDPROC
```

If the argument `Routine` is used, then RWMT will try to call a production routine, whose name fits the value of Routine.

If so, the RWMT mechanism for reading a program number as all other mechanisms like the cycle handling as well as the check code evaluation are bypassed.

Example 3:

The routine `UserProduction` has to be called.

```
PROC MT_GetUserProgNo(
INOUT dnum ProgNo,
INOUT string Routine)
!
!Production routine name assigned
Routine:="UserProduction";
!
ENDPROC
```

Depending on the specific conditions, either the program number or the routine name can be assigned, as shown by the following example.

Example 4:

```
PROC MT_GetUserProgNo(
INOUT dnum ProgNo,
INOUT string Routine)
!
VAR dnum dnProgNo
!user defined hand shake
!to read the program number
IF diProgNoReady=high THEN
```

Continues on next page

14.9.8 User defined programs
Continued

```
dnProgNo=giProgNo;
Set doProgNoAck;
...
!select the routine name which shall
!be executed
TEST dnProgNo
CASE 1,2,3,4,5:
Routine:="Production_T"+Valtostr(dnProgNo)
CASE 100,101:
!call service routines (menudata) or standard
!partdata
ProgNo:=dnProgNo;
ENDTEST
ENDIF
!
ENDPROC
```

14.10 Designing the service routines

14.10.1 Normal service routines

Data declaration and service routine

Similar to the declarations of the type `partdata` and the corresponding production routines, there also exists a data type *menudata – Menu declaration for service routines or set up routines on page 276*, which controls the execution of service routines under RWMT.

The setting up is discussed in detail in the chapters *Service routines on page 81*, and *Setup view on page 91* .

The following example illustrate the use once again:

```
MODULE MT_MAIN
…
!Menu data for service routines to be called
!
CONST PERS menudata mnChangeGripper:=
["Change gripper","Service","","ChangeGripper",
"",1,TRUE,7,1,1];

CONST PERS menudata mnCleanGripper:=
["Clean the gripper ","Service","","CleanGripper",
"",1,TRUE,9,1,1];
…
…
PROC ChangeGripper()
!
!Implement here your instructions
!to change the gripper
!
ENDPROC
…
…
PROC CleanGripper()
!
!Implement here your instructions
!to clean the gripper
!
ENDPROC
```

The two service routines for changing or cleaning the gripper can be called either in remote manner through the program numbers 7 or 9 or directly from the RWMT user interface.

14.10.2 Special service routines

RWMT provides the possibility to move the robot to 3 different service positions.

These service positions can be requested remotely through digital inputs that have to be assigned as shown in the chapter *Parameterization of the MT API Positions on page 188*.

To select those service positions in the graphical user interface, menudata declarations have to be provided as for the normal service routines.

Those declarations must be applied the predefined names `mnuServicePos1`, `mnuServicePos2`, and `mnuServicePos3`.

The declarations are predefined in the template file `MT_Main.mod`. The application programmer should equip them with the appropriate data or delete them, if not used. Furthermore, the service position routines have to be created and equipped by the application programmer.

Example:

```
CONST menudata mnuServicePos1:=
["Service position 1","Positions","",
     "ServicePos1","",3,TRUE,5,1,50];

PROC ServicePos1()
!
!Move to service position 1
MoveTo 991;
!Wait a little bit there...
WaitTime\InPos, 2;
!Move back to home position
MoveTo 999;
!
ENDPROC
```

The menu data declaration is of **type 1**, which means that the execution of the corresponding routine also sets the program pointer to this routine. See chapter *menudata – Menu declaration for service routines or set up routines on page 276* to learn more about menu data declarations and their different types.

The **ServicePos1** routine is called in the service menu. The menu will be displayed only if at least one user has logged in with service permissions. The menu can be called in the automatic mode of the robot through the GUI or through the external program number selection (program number **5**), if the robot is in the home position or safe position.

The corresponding routine first of all moves the robot to the service position (position number **991**). After having reached the position and 2 more seconds waiting time, the robot is moved back to the home position.

14.11 Suggestions for designing the program

14.11.1 The station concept

Explanation

The station concept of RWMT describes the perspective of the production unit and the corresponding mapping or representation in the robot program.

A robot cell consists of individual machines, conveyor belts and devices, apart from the robot. These are referred to in general as a station.

As described in the chapter *Stations on page 44*, these stations are represented graphically with their properties (state, station signals, station variables, station applications) by RWMT

The station concept can be considered in the actual robot program too, apart from the visualization, by placing all the data and routines of a station at one location. This can be achieved by adhering to the following rules:

- All the data and routines of a station are present in one single RAPID module and are declared as LOCAL.
- Physical signals of a station will be associated with the local alias signals.
- The station routines are called with the help of Late Binding specifying name of the module.

Example:

```
MODULE MT_MAIN
…
PROC Production()
…
!Unload the injection moulding machine
%"IMM1: Unload"%;
!Place the part on the conveyor
%"CNV1:Load"%;
…
ENDPROC
MODULE CNV1
!Version of this station
LOCAL CONST versiondata vd_CNV1:=
["CNV1.mod","V 0.10","2011-03-01"];
!
!Digital inputs
LOCAL VAR signaldi sdiLoadCNV;
!Digital outputs
LOCAL VAR signaldo sdoIRBOutOfCNV;
LOCAL VAR signaldo sdoPulseCNV;
!
!IO mappings for signals / alias signals
LOCAL CONST stationsignal CNV1_Signals{3}:=[
["Release to load conveyor","diLoadCNV1","sdiLoadCNV"],
["Robot out of conveyor","doIRBOutOfCNV1","sdoIRBOutOfCNV"],
["Pulse conveyor","doPulseCNV1","sdoPulseCNV"]];
```

Continues on next page

14.11.1 The station concept
Continued

```
!
!Base information about this station to be shown in GUI
PERS stationdata CNV1_Station:=
["CNV1","Conveyor 1","Conveyor 1 to outfeed good parts",
"station-conveyor.png","","","","",TRUE,FALSE,5,3];
!
!Station variables to be monitored or modified in GUI
LOCAL CONST stationvariable CNV1_Variables1{1}:=
[["Conveyor condition wait time”, "ntWaitTime",
"Conveyor","T_ROB1",0,99999,TRUE,TRUE,TRUE,0,1]];
!
!Event definition for station initialization
LOCAL CONST eventdata evStart:=
[EE_START,"CNV1:Initialize","",1];
!
!Waiting time for conditions
LOCAL PERS num ntWaitTime:=0;
!
LOCAL PERS msgdata msgLoad:=[30,1,btnNone,
"Wait for conveyor loading release.","","","","","",1,""];
!
!Hotedit position declarations
LOCAL CONST hoteditdata MT_he_CNV1{1}:=
[
["Conveyor 1","station-conveyor.png","Movement",
"mv300_301","p301","",""]
];
!
LOCAL PROC Initialize()
!
!Connect the physical signals to the alias
!signals of this station
MT_AliasIO CNV1_Signals\ModuleName:=CNV1;
!
!Set signals to their initial state
Set sdoIRBOutOfCNV;
Reset sdoPulseCNV;
!
ENDPROC
LOCAL PROC Load()
!
!Set this station as the active station at the GUI
MT_SetActiveStation CNV1_Station;
!
!Move to preposition of conveyor
MoveTo 300;
!Wait for conditions to enter conveyor area
MT_WaitMsgDi sdiLoadCNV,high,msgLoad;
!Move to place position at conveyor
MoveTo 301;
!
```

Continues on next page

```
!Switch off vacuum
MT_GripSet gsVacuumOff,gdSucker1\PartLoad:= load0;
!
!Move to end position outside conveyor
MoveTo 302;
!
ERROR
RAISE;
ENDPROC
ENDMODULE
```

In the example shown above the station CNV1 (first conveyor) can be duplicated by means of a text editor. This is done by copying the module CNV1 and insert it. Then the prefix CNV1 must be replaced by CNV2

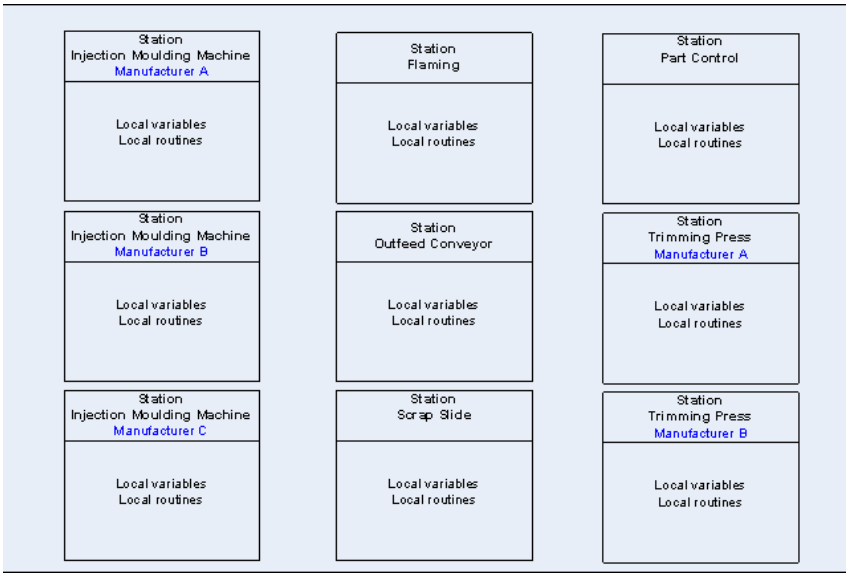
In addition a new number group must be provided because the second conveyor will need different robot positions than the first one.

A new number group is not mandatory if the movement content of the new CNV2 module is declared as local.

Advantages of the station concept

- Through the local data maintenance, identical copies of station modules can be used within a program, for example, if a processing machine appears several time sin a row in the same execution.
- During regular program creation for different plants or systems, a library can be built up, in which the individual station modules represent the behavior and the actuation of the corresponding machine completely. In subsequent projects, this library can be accessed, thereby saving time spent on the programming.

Example for a station library:



en1300000151

14.11.2 The program architecture

Task description

As an example, a robot program should be prepared with the following conditions:

- There is one single logical program run of the kind
  - Unload Station A
  - Load Station B
- There are two types of parts, which should not be handled by the robot with this program run.
- Later on, more part types could be added to this program run.
- There should be only one single robot program for all the part types.
- All the part types have different loading and unloading positions on the machines.
- It should be possible to test the movement processes independent of the logical program run.

Solution approach

The simplest solution to this task consists in,

- decoupling the movement calls completely from the process logic and
- calling individual movement processes dynamically in routines, during run time.

The following sections show the conventions for implementation.

Modularization

A program which should contain different part types for the same run, essentially differs only in terms of the gripper positions and depositing positions that should be approached in the stations / machines.

For this reason, it is useful to separate the actual production process from the movement instructions and movement data.

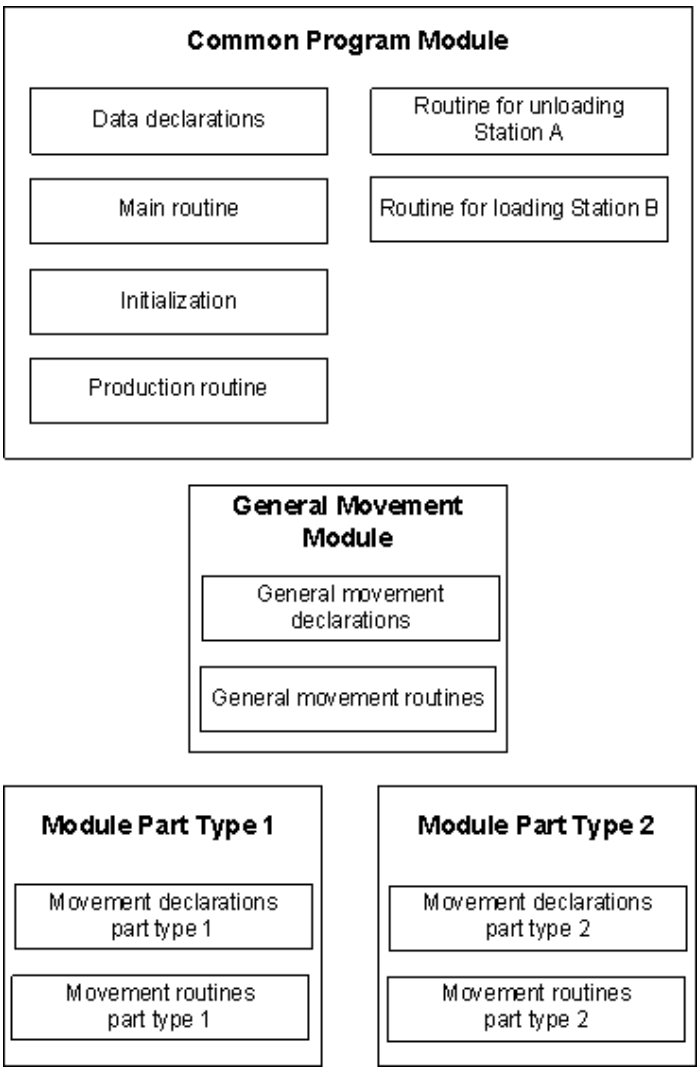
In our example, this means in concrete terms that routines for the program initialization as well as the loading and unloading of stations will be placed along with the production routine in a common program module.

All the movement instructions pertaining to a part type will be located along with the corresponding movement data (robtarget declarations, tool data declarations, wobjdata declarations) in a separate **part type module**.

All the other movement instructions, which do not belong to any part type, will be placed along with all the corresponding movement data (robtarget declarations,

Continues on next page

tool data declarations, wobjdata declarations) in a general movement module. The following figure illustrates the architecture:



en1300000152

Naming conventions for positions

Position names basically begin with the letter `p`.

Special standard positions can be given descriptive names (`pHome`, `pBereit`, and so on.).

All the other positions are named in accordance with the following convention:

Format	Description
`pXXX` `pXXX_Tt`	p:Position-Prefix XXX: 2- or 3-digit numerical station name Tt:type number for indexing by part types with or without the type prefix **T**

Examples of valid position names:

p100	Preliminary position outside the processing machine

Continues on next page

14.11.2 The program architecture
Continued

p101	Grip position within processing machine
p200_T7	Preliminary position outside the press for part type with Index 7
p201_T7	Loading position in the press for part type with Index 7

For subordinate, intermediate positions, sequential numbering or even the selection of the asterisk * position is found to be useful, since it is not possible to find an intelligent, self explanatory name for every position, which also fits the maximum length of a position name.

A sequence of *-positions (even individual positions) should always be enclosed by uniquely named positions, for example, p100, *,...,p101.

Use and naming of movement routines

The name for movement routines always denotes the starting point and the end point of a movement. In general, all the movement routines begin with the prefix `mv`.

```
mvStart_End [ _T<TypNr>]
```

Examples:

`mv100_101`	Moves the robot from the pre-position of the processing machine into the gripping position in the processing machine.
`mv200_201_T5`	Moves the robot from the pre-position of the processing machine to the gripping position inside this machine for part type 5.

Basically, the first position (start position) is approached only in the programming mode and at low speed. In this way, an uncontrolled movement from the end position back to the start position while testing the movements of the robot can be prevented.

A system operator should not be prevented from running a movement routine in the continuous mode, which means that after the last instruction in the routine, the 1st instruction will be processed once again. A low speed can thus prevent uncontrolled movement and heavy damages.

Example:

```
PROC mv100_101()
!From : Prepos processing machine
!To: Gripping pos processing machine
IF OpMode()<>OP_AUTO MoveJ p100,v200.z10.tGripper;
MoveJ *,v2500.z10.tGripper;
MoveL p101,v2500.z10.tGripper;
ENDPROC
```

If different part types with different movement routines have to be handled in a robot program, but with the same program run, then these are described by a type dependent index, which is provided with an additional prefix, for example, **T**.

Example:

Movement from Position 200 to 201 for type number 7 with the type prefix **T**:

```
PROC mv200_201_T7()
!From : Prepos trimming press
!To: Loading pos trimming press
```

Continues on next page

```
IF OpMode()<>OP_AUTO MoveJ p200_T7,v200.z10.tGripper;
MoveJ *,v2500.z10. tGripper;
MoveL p201_T7,v2500.z10. tGripper;
ENDPROC
```

Calling the movement routines

By **relocating** the data that is relevant to movement, the administrative program can now be formulated in a very general manner.

However, a mechanism is required for calling the respective, required movement routines for the part type that is to be handled currently during the program run.

This is done preferably by calling routines with subsequent binding (Late Binding `%string%`).

Example:

```
%“mv100_101_T“+ValToStr(nTypNr)%; (with type prefix)
or with
CallByVar „mv100_101_T“,nTypNr;
```

Alternatively, the instruction `MT_MoveTo` could be used. Its use is described in the chapter *RAPID Reference => Instructions => MoveTo -Dynamic execution of a movement routine* ..

Example for the implementation

Additional program can be found in the RWMT additional option folder in subfolder **Program Example**.

14.12 Program test

14.12.1 General safety measures

Extra caution is advised in general while testing a robot program. This includes the following, among other things:

- Careful inspection of the program code before starting the test.
- Ruling out of all risks to other persons who may be working in the vicinity of the system.
- Testing at low speed in the set up mode of the robot.

14.12.2 Validating the gripper functions

If the data type `grpdata` as well as the RWMT instructions for gripper actuation are used in the application program, then, in the grpdata declaration of a gripper that has not been tested yet, the element `Valid` should be set to FALSE (validation status).

This is to prevent the starting of production in an inadvertent manner with an untested gripper. This is relevant in particular to more complex grippers.

After a gripper has been tested successfully, in its `grpdata` declaration the element `Valid` should be set to TRUE, so as to be able to carry out production with this gripper.

14.12.3 Test mode

With the help of the test mode, a station can be activated even if it announces itself currently as not **ready** through an existing, external signal.

This is necessary during the commissioning for testing the program run even with stations that are currently not yet ready for operation.

The test mode is activated and deactivated through the RWMT user interface. To do so, the robot icon should be selected first in the station view.

Robot

In the view that opens, now switch to the **Debugging** area of the test mode (green tick mark is set) or switched off (green tick mark is not set).

Test mode activated

Switching on the test mode ensures that the station is activated in RWMT, if

- the station has been activated from the GUI AND
  - the deactivation on the user interface has been released in the process parameters

    OR
  - an external signal has been defined, which can activate or deactivate the station through remote controls. The state of the signal does not play any role here.
- the deactivation on the user interface has not been released in the process parameters, nor has an external signal been defined, which can activate or deactivate the station through remote control.

With the test mode, also the disabled buttons for the cycle handling are shown in the graphical user interface, if the cycle handling is done exclusively externally through a group input.

15 RAPID references

15.1 Data types

15.1.1 cellopmode – Cell operation mode

Usage

`cellopmode` is used for defining the available cell operation modes of RWMT. This must not be mistaken for the robot operation mode.

The predefined cell operation mode constants can be used for comparison with the function `MT_GetOperationMode` (see, *MT_GetOperationMode – Current cell operation mode on page 467*).

Pre-defined data

cellopmode can contain the following values:

Constant	Number	Operation mode
`OP_NO_ROBOT`	0	Without robot
`OP_SERVICE`	1	Service
`OP_PRODUCTION`	2	Production

Properties

`cellopmode` is an alias data type for num and hence inherits its properties.

15.1.2 Cycledata – Program cycle setting

Usage

`cycledata` is used to define program cycles for the finishing process of a part.

Description

In RWMT, program cycles are used for the process control of a part program and are called by the system operator through the GUI or through an external selection.

Four different types of cycles are supported:

- Continuous cycles
- Counter cycles
- Action cycles
- Periodic cycles

The cycle setting is done through the array `MT_CycleList{20}` of the data type `cycledata`.

The evaluation of the cycle that is to be processed is done by RWMT, which assigns the required cycle index that is necessary in each case to the persistent entity `MT_nCycleIndex`. In the processing program, the runs of the required cycles must be created in a corresponding manner.

Restriction

Note

The cycle data field must be declared as `TASK PERS`, since the GUI should be in a position to describe the data field.

The field is reduced to 20 cycles.

Basic example

```
TASK PERS cycledata MT_CycleList{20}:=[
["Production","",1,0,0,0,0,0],
["Warm-up","cycle-warm-up-robot.gif",2,1,3,0,1,0],
["","",0,0,0,0,0,0],["","",0,0,0,0,0,0],
["","",0,0,0,0,0,0],["","",0,0,0,0,0,0],
["","",0,0,0,0,0,0],["","",0,0,0,0,0,0],
["","",0,0,0,0,0,0],["","",0,0,0,0,0,0],
["","",0,0,0,0,0,0],["","",0,0,0,0,0,0],
["","",0,0,0,0,0,0],["","",0,0,0,0,0,0],
["","",0,0,0,0,0,0],["","",0,0,0,0,0,0],
["","",0,0,0,0,0,0],["","",0,0,0,0,0,0]]
PROC Bauteil1()
TEST MT_nCycleIndex
CASE 1:
Production;
CASE 2:
Start up;
```

Continues on next page

```
ENDTEST
ENDPROC
```

Definition of a production cycle and of a start up cycle.

Components

`Name`

Data type: `string`

Name of the cycle in the GUI (maximum 10 characters).

`Image`

Data type: `string`

The file name of the image button of size 62x55 pixel that is to be displayed in front of the menu entry. For this, the image must be present either directly in the **HOME:** directory or in the **SYSTEM:** directory.

Number	Image	
""	 xx1300000171	Furthermore, it is also possible to use one of the images shown on the left by specifying the image number. **Example**: `Cycledata.Image:= "1";`
"1"	 xx1300000172	
"2"	 xx1300000173	
"3"	 xx1300000174	
"4"	 xx1300000175	
"5"	 xx1300000176	

`Index`

Data type: `num`

Unique cycle index, which is queried by an external cycle selection or is communicated through the GUI and is queried in the robot program.

Continues on next page

15.1.2 Cycledata – Program cycle setting
Continued

`Type`

Data type: `cycletype`

The cycle type is defined through the following values:

Value	Cycle
0	Continuous cycle
1	Counter cycle
2	Action cycle
3	Periodic cycle

`NoOfExecSet`

Data type: `num`

Number of cycles to be executed.

(Not applicable to continuous cycles).

`Interval`

Data type: `num`

Number of processing cycles to be executed before the periodic cycle is executed.

(Is applicable only to periodic cycles).

`ContIndex`

Data type: `num`

Index of the cycle, which is to be executed after a counter cycle ends. If the value 0 is used, then the program will be ended.

Example:

```
Cycledata.ContIndex: = 1;
```

In the basic example, after 3 cycles of the start up cycle have been executed, the cycle 1 (production) will be executed.

Note

To continue a counter cycle, only indices of continuous cycles or counter cycles can be used.

(Applicable only for counter cycles).

`NoOfExecAct`

Data type: `num`

Internal cycle counter for counting the cycles executed.

Structure

```
< Dataobject of cycledata>
< Name of string >
< Image of string >
< Index of num >
< Type of cycletype>
< NoOfExecSet of num >
< Interval of num >
< ContIndex of num >
< NoOfExecAct of num >
```

15.1.3 cycletype – Type of cycle

Usage

`cycletype` is used for defining the cycle type of a processing cycle (See *Cycledata – Program cycle setting on page 246*).

Pre-defined data

`cycletype` can contain the following values:

Constant	No.	Cycle
`CT_CONTINUOUS`	0	Continuous cycle
`CT_COUNT_CYCLES`	1	Counter cycle
`CT_COUNT_CYC_ACTION`	2	Action cycle
`CT_PERIODICAL`	3	Periodic cycle

Properties

`cycletype` is an alias data type for `num` and hence inherits its properties.

15.1.4 eventdata – Execute routine on program event or system event

Usage

`eventdata` is used to define which routine is to be processed in the case of a system event or program event.

Description

If a system event or program event occurs, then all the event declarations in the robot program (`eventdata`) will be checked with the help of the event number (`eventnum`) and the event routines that are appropriate for the respective event will be executed in the sorting sequence (`eventdata.Sortorder`).

All the system events (for example, Start, Stop, , and so on.) can be declared by specifying the event number, the routine and the sorting sequence in the RAPID program, without necessitating a change in the system parameters.

If program modules or system modules are loaded only where required, then, a specific routine can be executed automatically through the event declaration, which is present in the module which is to be loaded, without necessitating a change in the robot system.

Basic examples

```
CONST eventdata edMy_POWERON:=
[EE_POWERON, "MyModule:MyPOWERON",10];
```

System event: The `MyPOWERON` routine in the `MyModule` module will be executed when the controls are switched on.

```
CONST eventdata edMy_INIT:=
[EE_BEFORE_INIT, "MyModule:MyInit",10];
```

Program event: The `MyInit` routine in the `MyModule` module of the RWMT Engine will be called at program start from **Main** before the program initialization.

Components

`Event`	Data type: `eventnum` Event for which the specified routine is to be executed
`Routine`	Data type: `string` The name of the routine contains a RAPID procedure without any transfer of parameters (for example, `MyINIT`), where even routines that are local to the module can be used. Here, the module name is written separated by means of a colon before the name of the routine (f `MyModule:MyINIT`)
`SortOrder`	Data type: `num` Processing sequence (0-100) of the routines for the same system event, that is, the routine of an event declaration with a `SortOrder 1` will be called before a routine with the `SortOrder 10`.

Properties

The event must be declared as a **constant**.

It may never be `LOCAL`.

Continues on next page

The name of the declaration is of no consequence, since the system will be searched for all the declarations of the data type `eventdata`.

However the declaration name must be unique.

Structure

```
< Dataobject of eventdata>
< event of eventnum >
< Routine of string >
< SortOrder of num >
```

15.1.5 eventnum – Program event number or system event number

Usage

`eventnum` is used to define the system event or program event for which the assigned event routine is to be executed.

Description

If a system event or program event occurs, then all the event declarations in the robot program will be checked with the help of the event number and the routines of the event declaration that are appropriate for the respective event will be executed.

Basic examples

```
CONST eventdata edMy_START:=
[EE_START, "MyModule:MySTART","",10];
```

The `MySTART` routine in the `MyModule` module will be processed from the beginning on starting (start from **Main**).

Pre-defined data

The system events and program events are defined through the following event number:

Constant	Event No.	Event
EE_POWERON	1	Restarting the robot controller.
EE_START	2	Processing is started from the beginning of the program.
EE_POWERON_OR_START	3	Robot was restarted or the processing will start when the program starts.
EE_RESTART	4	Processing starts from the current position of the program pointer.
EE_START_OR_RESTART	5	Processing will be started from the beginning of the program or from the current position of the cursor.
EE_STOP	6	The program was stopped: • with the help of the Stop key, • with the help of a Stop instruction, • or with a stop after the current instruction.
EE_QSTOP	7	A quick stop of the robot has been executed (for example, emergency stop)
EE_STOP_OR_QSTOP	8	Program was paused by Stop or emergency stop.
EE_RESET	9	A program was closed or loaded. The event will not be activated once a system module or a program module has been loaded.
EE_STEP	10	Step-wise execution of the program forwards or backwards
EE_STEP_FWD	11	Step-wise execution of the program forwards
EE_STEP_BCK	12	Step-wise execution of the program backwards

Continues on next page

Constant	Event No.	Event
EE_BEFORE_INIT	20	MT_Execute was started and the initialization is done in the next step.
EE_AFTER_INIT	21	One-time initializing after start from main() before calling any production routine (part program). The robot is already located in home position.
EE_WAIT_ORDER	22	Execution is done always if the robot is waiting for a program number.
EE_BEFORE_PROD	23	Execution will be done always before processing of the part program.
EE_AFTER_PROD	27	Execution will be done after the processing of the part program and after the event EE_AFTER_PART.
EE_BEFORE_MENU	28	Execution will be done before the processing of a menu routine.
EE_AFTER_MENU	30	Execution will be done after the processing of a menu routine.
EE_ERROR	31	Event will be done, if one of the error numbers ERR_MT_HOME (termination of program with direct **Halt after end of cycle**) respectively ERR_MT_ABORT (termination of the current production cycle) is used.
EE_HOMERUN	32	Execution takes place at program start of MT_Execute, if the robot is not in the home position.
EE_PROG_END	33	Event will be triggered if the RWMT engine is left and shortly before the robot program is terminated.
EE_AFTER_PROG_NUMBER	34	Event will be done if an external program number has successfully been read.
EE_PROGNO_UNKNOWN	35	Triggered, if a program number has been read by RWMT but this program number neither matches a partdata nor a menudata declaration. The event can be used for example, to load a module which contains the missing partdata or menudata declaration. After execution of the event, RWMT verifies again, if the program number matches a partdata or menudata.if this is still not the case, the progrem execution is aborted with an appropriate message.
EE_PROD_UNKNOWN	36	Triggered, if a routine name has been specified in the <routine> item of a partdata declaration, which has been selected for production, but the specified routine does not exist in the program. The event can be used for example, to load a module which contains the missing routine. After execution of the event, RWMT verifies again, if the routine is available in the program. If the routine is still not available, the progrem execution is aborted with an appropriate message.

Continues on next page

15.1.5 eventnum – Program event number or system event number
Continued

Constant	Event No.	Event
EE_BLOCKED	100	Event which is triggered by a task, after having called a setup routine by means of the RWMT GUI.

Properties

`cycle type` is an alias data type for num and hence inherits its properties.

15.1.6 grpaction – Set and check actions in gripper sequences

Usage

`grpaction` supports the data type `gripseq`(see *grpseq – Gripper sequence for actuating several control elements on page 264*). While `gripseq` is used for a sequential setting of gripper actuators, the data type `grpaction` defines for each sequence, if the actuators only have to be set or also checked.

Predefined gripper actions

Only the following grpaction may be used:

Constant	Event number	Event
`gaSetAndCheck`	0	Gripper action to set the outputs and wait until sensor signals have the required state.
`gaSet`	1	Gripper action to set the ouputs without checking the sensor signals.
`gaCheck`	2	Gripper action to only check the sensor signals.
`gaCheckClose`	3	Gripper action to only check status of the **Closed** sensor signals.
`gaCheckClose`	4	Gripper action to check status only of the **Open** sensor signals.

Basic example

```
TASK PERS grpdata gdY1_T127:=[];
TASK PERS grpdata gdY2_T127:=[];
TASK PERS grpdata gdY3_T127:=[];
TASK PERS grpdata gdY4_T127:=[];
!gripper sequence
const grpseq gsOpen_T127{3}:=
[[gsClose,gaSetAndCheck,"gdY1_T127","","","","","",0],
[gsOpen,gaSet,"gdY2_T127","gdY3_T127","", "","","",0],
[gsClose,gaCheck,"gdY4_T127","","", "","","",0]];
```

In the sequence for opening the gripper for part 127, the control element Y1 is first closed, then the control elements Y2 and Y3 are opened and then the control element Y4 is closed.

In the first sequential step, the actuators are to be set and then checked afterwards for the expected condition. In the second step, the actuators are only set but not checked. In the third step the condition for the the actuator, no actuator is set but only checked.

Components

`Position`	Datatype: `grppos` The new position of the gripper that is expected by the actuation.
`Action`	Data type: `grpaction` Action, which has to be executed (setting and checking actions of the gripper actuators).

Continues on next page

15.1.6 grpaction – Set and check actions in gripper sequences
Continued

`Grp1`	Data type: `string` Variable name of the actuator 1 (grpdata-declaration name)
`Grp2`	Data type: `string` Variable name of the actuator 2 (grpdata-declaration name)
`Grp3`	Data type: `string` Variable name of the actuator 3 (grpdata-declaration name)
`Grp4`	Data type: `string` Variable name of the actuator 4 (grpdata-declaration name)
`Grp5`	Data type: `string` Variable name of the actuator 5 (grpdata-declaration name)
`Grp6`	Data type: `string` Variable name of the actuator 6 (grpdata-declaration name)
`WaitTme`	Data type: `num` Common waiting time after the complete sequence is executed.

Structure

```
<dataobject of grpseq>
< Position of grppos>
< Action of grpaction>
< Grp1 of string >
< Grp2 of string >
< Grp3 of string >
< Grp4 of string >
< Grp5 of string >
< Grp6 of string >
< WaitTime of num >
```

15.1.7 grpdata – Configuration of a control element of the gripper

Usage

`grpdata` is meant for the configuration of a control element of a gripper.

With a **Control element**, **an action** can be controlled at a gripper, which normally consists in the actuation of a valve. The concerned action can be monitored with a maximum of 4 pairs of sensors, for example, the end positions of cylinders (open or closed position) or a vacuum sensor can be queried.

If all the pairs of sensors or sensor positions are not required, then the corresponding parameter for the name of the signal can remain blank.

The control element configuration is used with the instruction `GripSet` or `GripCheck`.

Components

`Label`	Data type: `string` Name of the gripper and/or of the control element
`ToolCode`	Data type: `byte` Gripper coding for which this configuration is valid.
`Valid`	Data type: `bool` Validation status of the gripper function
`WaitTime`	Data type: `num` Maximum waiting period after setting of the gripper outputs, before the sensors are checked and, if the expected conditions are not given, an error message is displayed.
`NoGhostSet`	Data type: `bool` Activate the actuation of the valve in the ghost mode (FALSE) or deactivate it (TRUE).
`NoGhostCheck`	Data type: `bool` Activate checking of sensors in the ghost mode (FALSE) or deactivate it (TRUE). The flag is used, for instance, if the control element sensors do not report the expected state in the ghost mode, for example, in the case of vacuum grippers.
`Valve`	Data type: `grpvalue` Valve actuation for the control element.
`Sensor1`	Data type: `grpsensor` Sensor configuration for the 1st pair of sensors for checking the **Open** or **Closed** position.
`Sensor2`	Data type: `grpsensor` Sensor configuration for the 2nd pair of sensors for checking the **Open** or **Closed** position.
`Sensor3`	Data type: `grpsensor` Sensor configuration for the 3rd pair of sensors for checking the **Open** or **Closed** position.
`Sensor4`	Data type: `grpsensor` Sensor configuration for the 4th pair of sensors for checking the **Open** or **Closed** position.

Continues on next page

15.1.7 grpdata – Configuration of a control element of the gripper
Continued

Basic example

```
TASK PERS grpdata gdGripper
[
"Gripper",1,TRUE,0.5,TRUE,TRUE,
["doCloseGripper",0,"doOpenGripper",0,
"Close Gripper","Open Gripper"],
["Gripper","diGripperClosed","diGripperOpened"],
["","",""],["","",""],["","",""]
];
```

Gripper with an output signal for closing and an output signal for opening. The gripper status is reported with an input for **Gripper is open** and an input for **Gripper is closed**.

The setting and checking of the gripper is disabled in the ghost mode. The gripper code for this gripper function is 1.

After setting the actuators and after a waiting period of 0.5 seconds, a check will be done, if the gripper sensors reflect the expected gripper position.

Restriction

Note

In order to be able to represent the gripper data on the RWMT user interface (GUI), it must be declared as a **persistent** entity (TASK PERS).

Structure

```
<dataobject of grpdata>
< Label of string>
< Toolcode of num>
< WaitTme of num>
< NoGhostSet of bool>
< NoGhostCheck of bool>
< Valve of grpvalve>
< Outp_Close of string>
< PulseLenClose of num >
< Outp_Open of string>
< PulseLenOpen of num >
< CloseText of string>
< OpenText of string>
< Sensor1 of grpsensor >
< Label of string>
< Inp_Close of string >
< Inp_Open of string >
< Sensor2 of grpsensor >
< Label of string>
< Inp_Close of string >
< Inp_Open of string >
< Sensor3 of grpsensor >
< Label of string>
< Inp_Close of string >
```

Continues on next page

```
< Inp_Open of string >
< Sensor4 of grpsensor >
< Label of string>
< Inp_Close of string >
< Inp_Open of string >
```

15.1.8 grppart – Part control configuration

Usage

`grppart` is meant for the configuration of the part controls of a gripper.

Components

`Label`	Datatype: `string` Name of the part.
`ToolCode`	Datatype: `num` Gripper coding for which this configuration is valid.
`Sensor1`	Datatype: `grpsignal` Signal description and signal name of the robot input for the 1st part sensor
`Sensor2`	Datatype: `grpsignal` Signal description and signal name of the robot input for the 2nd part sensor
`Sensor3`	Datatype: `grpsignal` Signal description and signal name of the robot input for the 3rd part sensor
`Sensor4`	Datatype: `grpsignal` Signal description and signal name of the robot input for the 4th part sensor
`Sensor5`	Datatype: `grpsignal` Signal description and signal name of the robot input for the 5th part sensor
`Sensor6`	Datatype: `grpsignal` Signal description and signal name of the robot input for the 6th part sensor
`Sensor7`	Datatype: `grpsignal` Signal description and signal name of the robot input for the 7th part sensor
`Sensor8`	Datatype: `grpsignal` Signal description and signal name of the robot input for the 8th part sensor

Restriction

Note

In order to be able to represent the gripper part data at the RWMT user interface (GUI), this must be declared as persistent (TASK PERS).

Basic example

```
TASK PERS grppart gpComponent1:=
[["Component1",13,["Check1","diSensor1"],
["Check2","diSensor2"],["",""],["",""],
["",""],["",""],["",""],["",""]];
```

Continues on next page

Configuration of the part controls for the part **component 1** which is gripped with the gripper with gripper coding 13. The checking is done through two sensors, which are labeled as **Check1** and **Check2** and which use the two robot inputs **diSensor1** and **diSensor2** for the query.

Structure

```
<dataobject of grppart>
< Label of string>
< Toolcode of num>
< Sensor1 of grpsignal >
< Label of string>
< SignalName of string>
< Sensor2 of grpsignal >
< Label of string>
< SignalName of string>
< Sensor3 of grpsignal >
< Label of string>
< SignalName of string>
< Sensor4 of grpsignal >
< Label of string>
< SignalName of string>
< Sensor5 of grpsignal >
< Label of string>
< SignalName of string>
< Sensor6 of grpsignal >
< Label of string>
< SignalName of string>
< Sensor7 of grpsignal >
< Label of string>
< SignalName of string>
< Sensor8 of grpsignal >
< Label of string>
< SignalName of string>
```

15.1.9 grppos – Gripper position

Usage

grpposis used to query the position of the control element (GripCheck) or to set it (GripSet).

Basic example

```
GripSet gsOpen,gdGripper1;
```

The control element of the gripper 1 will be opened.

Pre-defined data

Value	Symbolic constant	Description
1	gsOpen	Open control element
1	gsVacuumOff	Turn off vacuum (open)
1	gsBackward	Withdraw control element (actuator)
0	gsClose	Close control element (actuator)
0	gsVacuumOn	Turn on vacuum (Close)
0	gsForward	Move control element forward
-1	gsReset	Reset both outputs

Properties

grppos is an alias data type for num and hence inherits its properties.

15.1.10 grpsensor – Sensor configuration for the control elements of a gripper

Usage

`grpsensor` is meant for the configuration of the signals for the opened or closed sensors of a gripper within the data type `grpdata` (see *grpdata – Configuration of a control element of the gripper on page 257*).

Components

`Label`	**Datatype:** `string` Symbolic name of the sensor
`Inp_Close`	**Datatype:** `string` Signal name, referred to `EIO.CFG`, for the **Closed** sensor.
`Inp_Open`	**Datatype:** `string` Signal name, referred to `EIO.CFG`, for the **Open** sensor.

Structure

```
<dataobject of grpsensor>
< Label of string>
< Inp_Close of string >
< Inp_Open of string >
```

15.1.11 grpseq – Gripper sequence for actuating several control elements

Usage

`grpseq` is meant for the sequential actuation of several control elements. The data type is used by the instruction `MT_GripSequence`. Maximum six actuators that are defined by `grpdata` (see *grpdata – Configuration of a control element of the gripper on page 257*) can be used.

Gripper sequences are to be declared as an array of the data type grpseq. The size of the array may be defined between 1 and 20.

Basic example

```
TASK PERS grpdata gdY1_T127:=[];
TASK PERS grpdata gdY2_T127:=[];
TASK PERS grpdata gdY3_T127:=[];
TASK PERS grpdata gdY4_T127:=[];

!gripper sequence
const grpseq gsOpen_T127{3}:=
[[gsClose,gaSetAndCheck,"gdY1_T127","","","","","",0],
[gsOpen,gaSetAndCheck,"gdY2_T127","gdY3_T127","", "","","",0],
[gsClose,gaSetAndCheck,"gdY4_T127","","", "","","",0]];
```

In the sequence for opening the gripper for part 127, the control element Y1 is first closed, then the control elements Y2 and Y3 are opened and then the control element Y4 is closed.

A `grpaction` (see *grpaction – Set and check actions in gripper sequences on page 255*) constant defines the specific setting and checking action for each sequential step. In the first sequential step, the actuators are to be set and then checked afterwards for the expected condition. In the second step, the actuators are only set but not checked. In the third step the condition for the the actuator, no actuator is set but only checked.

Components

`Position`	Datatype: `grppos` The new position of the gripper that is expected by the actuation.
`Action`	Datatype: `grpaction` Action, which has to be executed (set and check actions on the gripper actuators)
`Grp1`	Datatype: `string` Variable name of the actuator 1 (grpdata-declaration name)
`Grp2`	Datatype: `string` Variable name of the actuator 2 (grpdata-declaration name)
`Grp3`	Datatype: `string` Variable name of the actuator 3 (grpdata-declaration name)
`Grp4`	Datatype: `string` Variable name of the actuator 4 (grpdata-declaration name)

Continues on next page

`Grp5`	**Datatype:** `string` **Variable name of the actuator 5 (grpdata-declaration name)**
`Grp6`	**Datatype:** `string` **Variable name of the actuator 6 (grpdata-declaration name)**
`WaitTme`	**Datatype:** `num` **Waiting time after sequence is executed**

Structure

```
<dataobject of grpseq>
< Position of grppos>
< Action of grpaction>
< Grp1 of string >
< Grp2 of string >
< Grp3 of string >
< Grp4 of string >
< Grp5 of string >
< Grp6 of string >
< WaitTme of num >
```

15.1.12 grpsignal – Configuration of a gripper signal

Usage

`grpsignal` is meant for the configuration of a digital input signal within the data type `grppart` (see *grppart – Part control configuration on page 260*).

Components

`Label`	**Datatype:** `string` Symbolic name of the sensor
`SignalName`	**Datatype:** `string` Name of the robot digital input, referred to `EIO.CFG`.

Structure

```
<dataobject of grpsignal>
< Label of string>
< SignalName of string>
```

15.1.13 grpvalve – Valve actuation for the control element of a gripper

Usage

`grpvalve` is meant for the configuration of the digital signals (outputs) for the opening or closing of a control element within the data type `grpdata` (see *grpdata – Configuration of a control element of the gripper on page 257*).

Components

`Outp_Close`	**Datatype:** `string` Physical name of the robot output for closing the control element (See System parameter `EIO.CFG`).
`PulseLenClose`	**Datatype:** `num` Pulse length [s] for closing the control element. If the pulse length is 0, then the output will be set permanently (static valve). If the pulse length is greater than 0, then the output will be pulsed for the specified time (pulse valve).
`Outp_Open`	**Datatype:** `string` Physical name of the robot output for opening the control element (See System parameter `EIO.CFG`).
`PulseLenOpen`	**Datatype:** `num` Pulse length for opening the control element. If the pulse length is 0, then the output will be set permanently (static valve). If the pulse length is greater than 0, then the output will be pulsed for the specified time (pulse valve).
`CloxeText`	**Datatype:** `string` Label text for the button for **Closing** the gripper in the Flexpendant application RWMT.
`OpenText`	**Datatype:** `string` Label text for the button for **Opening** the gripper in the FlexPendant application RWMT.

Structure

```
<dataobject of grpvalve>
< Outp_Close of string>
< PulseLenClose of num >
< Outp_Open of string>
< PulseLenOpen of num >
< CloseText of string>
< OpenText of string>
```

15.1.14 hoteditdata – Menu declaration for the HotEdit-pre-selection

Usage

`hoteditdata` is used to select the robot positions in a specific routine for the Standard HotEdit.

Description

The program module, the routine as well as the `robtargets` are saved in a menu entry for a change of position, through the Standard HotEdit, and loaded by selection through the GUI.

The use of program dependent or type number dependent modules, routines or positions is supported here through the use of wildcards.

Basic examples

Example 1:

```
CONST hoteditdata hedIMM_GripPos:=["IMM Gripping Pos",
     "station-IMM.png", "", "mv11_12", "p12","",""];
MODULE Movement
PROC mv11_12()
MOVEL p11,…;
MOVEL p12,…;
ENDPROC
ENDMODULE
```

The grip position p12 in the global routine `mv11_12` will be displayed in the HotEdit.

Example 2:

```
CONST hoteditdata hedIMM_GripPos:=["IMM gripping Pos",
     "station-IMM.png", "PART_T&", "mv11_12", "p12","",""];
MODULE PART_T10
LOCAL PROC mv11_12()
MOVEL p11,…;
MOVEL p12,…;
ENDPROC
ENDMODULE
```

The grip position p12 in the local routine of the module `mv11_12` in the module `PART_T10` will be displayed in the HotEdit, if the current type number has been set to 10.

Example 3:

```
CONST hoteditdata hedIMM_GripPos:=["IMM gripping Pos",
     "station-IMM.png", "", "mv11_12_T&", "p12_T&","",""];
MODULE Movement
PROC mv11_12_T10()
MOVEL p11,…;
MOVEL p12_T10,…;
ENDPROC
ENDMODULE
```

Continues on next page

The grip position `p12_T10` in the global routine `mv11_12_T10` in the module `Movement` will be displayed in the HotEdit, if the current type number has been set to 10.

Components

`Description`

Datatype: `string`

Descriptive text that will be displayed in the HotEdit selection menu.

`Icon`

Datatype: `string`

The name of the 32x32 pixel icon that is to be displayed in front of the menu entry. To make the background of the icon invisible, it should be filled with the color **Magenta**. If the icon is larger than or smaller than 32x32 pixels, it will be scaled to the required size.

For this, self-made icons must be present either directly in the **HOME:** directory or in the **SYSTEM:** directory.

`ModuleName`

Datatype: `string`

Name of the module, in which the routine is present along with the positions that are to be modified.

If program number dependent or type number dependent modules are used, then, the following wildcards can be used for the sake of simplicity:

- **&**: for the part type code (please refer to *partdata – Part data on page 284*)
- **%**: for the part program code (please refer to *partdata – Part data on page 284*)

Example:

```
hoteditdata.ModuleName:= "PART_&"
```

Module **Part_10** is used for the type number 10.

`RoutineName`

Datatype: `string`

Name of the routine, in which the positions that are to be modified are located.

If program number dependent or type number dependent routines are used, then, the following wildcards can be used for the sake of simplicity:

- **&**: for the part type code (please refer to *partdata – Part data on page 284*)
- **%**: for the part program code (please refer to *partdata – Part data on page 284*)

Example:

```
"hoteditdata.Routinename:= "mv10_20_T%"
```

Routine `mv10_20_T10` is used for the program number 10.

Continues on next page

15.1.14 hoteditdata – Menu declaration for the HotEdit-pre-selection
Continued

`Positions1`	Datatype: `string` Names of the positions that are to be changed. Due to the limited length of 80 characters of a string, 3 strings are used for specifying the positions. If several positions are used, then these must be written separated by a comma (for example, "p10,p20,p30") If program number dependent or type number dependent positions are used, then, the following wildcards can be used for the sake of simplicity: • &: for the part type code (please refer to *partdata – Part data on page 284*) • %: for the part program code (please refer to *partdata – Part data on page 284* **Example:** `"hoteditdata. Positions1:= "p10_T%, P11_T%, p20_T%"` The position `"p10_T5", "p11_T5" and "p20_T5"` will be displayed for the program number 5,
`Positions2`	Datatype: `string` Names of the positions that are to be changed, See `Positions1`.
`Positions3`	Datatype: `string` Names of the positions that are to be changed; See `Positions1`.

Structure

```
< Dataobject of hoteditdata>
< Description of string >
< Icon of string >
< ModuleName of string >
< RoutineName of string >
< Positions1 of string >
< Positions2 of string >
< Positions3 of string >
```

15.1.15 infodata – Displaying the information in the production window

Usage

`Infodata` are used to display information in the production data display of the production windows.

Description

In order to be able to represent data of the type **bool**, **num**, **dnum**, or **string** in the production data display, the data field `MT_InfoView` of the data type `infodata` should be used.

For the information to be updated automatically, this must be declared as `PERS`, `LOCAL PERS` or as `TASK PERS`.

Constants and variables will be read only while starting the GUI and will not be updated any more after this.

The size of the array `MT_InfoView` is the result of the number of data that is to be displayed and is limited only by the available memory space.

If more than 7 additional bits of information are displayed, then arrows will be displayed in the list, which can be used to scroll through the production data display.

Tip

To reduce the starting time of the GUI, it is advisable to specify the module names of the data that is to be displayed.

Basic example

```
PROC MainModule
LOCAL PERS num nUnloadTime:=35;
TASK PERS bool bCutterProtect:=TRUE;
PERS num nCaliperPos:=0:
CONST infodata MT_InfoView{3}:=[
["Unloading Time","nUnloadTime","MainModule",""],
["Cutter Protection","bCutterProtect","MainModule",""],
["Caliper Pos","nCaliperPos","MainModule",""]];
…
ENDMODULE
```

Display of the values of the persistent entity `nUnloadTime`, `bCutterProtect` and `nCaliperPos` in the production data display.

Components

`Description`	**Datatype:** `string` Text that is displayed in front of the data value as a description.
`VariableName`	**Datatype:** `string` Name of the data declaration (Variable, persistent entity or constant).

Continues on next page

15.1.15 infodata – Displaying the information in the production window
Continued

`ModuleName`	**Datatype:** `string` Name of the task that contains the data declaration. If the module name is not specified, then the module name is searched with the help of the name of the variable in the task.
`Taskname`	**Datatype:** `string` Name of the task that contains the data declaration. The task name should be specified only if the data declaration is in another task.

Structure

```
< Dataobject of infodata>
< Description of string >
< VariableName of string >
< ModuleName of string >
< Taskname of string >
```

15.1.16 instset – Execute instruction while change of cell mode of operation

Usage

`instset` is used while switching the RWMT mode of operation from **Production -> Service**, **Production -> Without robot** or from **Without robot -> Production** or from **Service-> Production** to set the outputs or to assign new values to persistent entities.

In addition, instset declarations can also be used according to the change of the robot operation mode (key switch).

Description

For instance in the injection moulding cells, there is the requirement that special outputs must be set depending on the RWMT mode of operation of the injection moulding machine, if the operations are to be carried out **With** or **Without** robot.

To ensure this, the **instruction sets** are declared in the robot program, which contain the data type, the data name as well as the desired value of the persistent entity or of the signal that is to be set.

This instruction set is executed through a background task, so that the instructions that need to be executed while switching the RWMT mode of operation can be executed even during the manual mode of the robot or when the program is stopped.

The instruction sets for the RWMT operation modes are entered in two generally valid arrays having the names `MT_RunWithRobot` and `MT_RunWithOutRobot`. Each of the arrays can record 15 instruction sets.

Furthermore, it is possible to declare for every station corresponding arrays with a data field size of 10 elements (for example, `<stationdata.Name>_RunWithRobot` or `<stationdata.Name>_RunWithOutRobot`)

The arrays which contain the `RunWithRobot` in the name will be executed as soon as there is a change to the cell mode of operation **Production**.

The arrays that contain the `RunWithOutRobot` in the name will be executed as soon as the cell mode of operation **Production** changes to the cell mode of operation **Service** or **Without robot**.

By analogy to the instruction set declarations for the RWMT operation modes, instruction sets can also be declared for the available robot operation modes **manual** and **automatic**, which are induced by using the key switch of the robot controller.

In the same way as for the RWMT operation modes, special data declaration names have to be used for them.

The declaration names `MT_OpModeManual` and `MT_OpModeAuto` are used for general purpose and their arrays can record 15 instruction sets. For station related instruction sets, the naming must be `<stationdata.Name>_OpModeManual` and `<stationdata.Name>_OpModeAuto`. The data field size must be 10 elements in this case.

The arrays which contain the `OpModeManual` in the name will be executed as soon as the key switch of the robot is set to manual mode.

Continues on next page

15.1.16 instset – Execute instruction while change of cell mode of operation
Continued

The arrays that contain the `OpModeAuto` in the name will be executed as soon as the key switch of the robot is set to automatic mode and the safety dialog at the FlexPendant has been confirmed.

Basic example

```
PERS bool bExample:= FALSE;
TASK PERS num nExample:= 0;
TASK PERS dnum dnExample:= 0;
TASK PERS string stExample:= "";


CONST instset MT_RunWithOutRobot{15}:=[
["DO","doSignalA","1",""],
["DO","doSignalB","1",""],
["BOOL","bExample ","FALSE",""]
["NUM","nExample ","10",""]
["DNUM","dnExample ","500",""]
["STRING","stExample ","Run without Robot",""],
["","","",""],["","","",""],["","","",""],["","","",""],
["","","",""],["","","",""],["","","",""],["","","",""],
["","","",""]];


CONST instset MT_RunWithRobot{15}:=[
["DO","doSignalA","0",""],
["DO","doSignalB","0",""],
["BOOL","bExample ","TRUE",""]
["NUM","nExample ","200",""]
["DNUM","dnExample ","12345",""]
["STRING","stExample ","Run with Robot",""],
["","","",""],["","","",""],["","","",""],["","","",""],
["","","",""],["","","",""],["","","",""],["","","",""],
["","","",""]];
```

The outputs `doSignalA` and `doSignalB` will be set to "low" if the robot is switched to the "Production" cell mode and to "high" when switching the robot to “Service mode” or mode “Without robot”.

Furthermore, the Boolean persistent entity changes from "FALSE" to "TRUE" for a change to "Production" mode, the numerical persistent entity changes from "10" to "200", the dnum persistent entity changes from "500" to "12345" and the string persistent entity changes from `Run without Robot` to `Run with Robot`.

Continues on next page

Components

`Type`

Datatype: `string`

Data type of the data element that is to be modified.

The following data types can be used:

Type	Data type
`AO`	Analog Output
`DO`	Digital Output
`GO`	Group Output
`BOOL`	Boolean persistent entity
`NUM`	Numerical persistent entity
`DNUM`	Double numerical persistent entity
`STRING`	String Persistent

`DataName`

Datatype: `string`

Name of the persistent entity or of the signal, for example, "doSignal1".

`Value`

Datatype: `string`

Value of the persistent entity or of the signal, for example, "1".

`TaskName`

Datatype: `string`

Name of the task that contains the data declaration.

The task name should be specified only if the data declarations is in another task.

Restriction

Only persistent data declarations can be modified ("PERS" or "TASK PERS").

Structure

```
< Dataobject of instset>
< Type of string >
< DataName of string >
< Value of string >
< TaskName of string >
```

15.1.17 menudata – Menu declaration for service routines or set up routines

Usage

`menudata` is used to define a service routine or set up routine, which can be called through the service menu or set up menu. Service routines can also be started through an external program number.

Description

The data type `menudata` contains all the information for displaying a menu entry in the set up menu or service menu for calling a service routine.

The menu entries can be defined as service routine or set up routine and will be displayed depending on the user who has logged in and the "Minimum User Level" that has been set.

With the help of the program code that has been specified, it is possible to call the service routine through an external program number selection (for example, from a PLC). For this, the program must be present in the cell mode of operation "**Service**" and should wait for the program number communication of the PLC.

Setup routines can be started simultaneously in several tasks in the case of MultiMove systems, by specifying the corresponding task in the parameter "`RunInTasks`".

The setup routines are executed in the manual mode by explicitly setting the program pointer to the corresponding routine, so that the original position of the program pointer is lost.

There are two types of service routines, which differ in their manner of working:

- **Type I**: service routine will be executed by the RWMT-Engine instead of a production program and can also be called through the external program selection.
- **Type II**: service routine will be executed only in the manual mode in parallel with the program pointer, so that the robot program can be continued after the service routine has ended. “In parallel with the program pointer“ means, that the program pointer remains on the position, where the production has been interrupted.

Basic examples

Service routine Type I

```
CONST menudata mnuGripperChange:=
["Change Gripper","Gripper","station-gripper.png",
"GripperChange","",3.TRUE,1000,1,50];
```

The `GripperChange` routine is called in the service menu. The menu will be displayed only if at least one user has logged in with service permissions. The menu can be called in the automatic mode of the robot through the GUI or through the external program number selection (program number "1000"), if the robot is in the home position or safe position.

Service routine Type II

```
CONST menudata mnuTC_Unlock:=
```

Continues on next page

```
["Unlock Tool Changer","Gripper","station-gripper.png",
"TC_Unlock","",255.FALSE,0,2,50];
```

The "TC_Unlock" routine is called in the service menu. The menu will be displayed only if at least one user has logged in with service permissions. The menu can be called in the manual mode of operation of the robot at any desired position.

Setup routine

```
CONST menudata mnuWObj:=
["Define Workobject","Werkobjekte","WobjBox.gif",
"Setup_Wobj","T_ROB1:T_ROB2",1,FALSE,0,3,150];
```

The "Setup_Wobj" routine is called through the set up menu. The menu will be displayed only if at least one user has logged in with programmer permissions. The set up routine will be called in the Tasks "T_ROB1" and "T_ROB2", if both the robots are in the home position.

Tip

The declaration of the service routine should be present in every task for this. All the data except the routine that is to be called and the valid position must be identical here.

Components

`Description`
Datatype: `string`
Descriptive text that will be displayed in the service menu for selection. If no description is defined, then the name of the routine that is to be called will be displayed (ProcName).

`Category`
Datatype: `string`
Category of the menu entry (for example, work objects, applications, , and so on.) is meant for filtering the menu entries.

`Image`
Datatype: `string`
The name of the 32x32 pixel icon that is to be displayed in front of the menu entry.

For this, the icon must be present either directly in the "HOME:" directory or in the "`System:`" directory.

If no separate image file is specified, then, the following icons will be represented:


Service routine Type I


Service routine Type II


Setup routine

Tip

To make the background of the icon invisible, it should be filled with the color "Magenta".

Continues on next page

15.1.17 menudata – Menu declaration for service routines or set up routines
Continued

`ProcName`
Datatype: `string`
Name of the routine which is to be executed as service routine, without handing over any parameters.
In order to be able to execute routines which are local to modules, then module name must be specified in front of the name of the routine, separated from it by a colon (for example, "`Module Name:RoutinenName`")

`RunInTasks`
(Only for setup routines)
Datatype: `string`
Task list of the set up routines that are to be executed simultaneously in several tasks in the case of a Multi-Move system.
Here, the tasks are separated from each other by colons (for example, "`T_ROB1:T_ROB2`")
For this, the same set up routine declaration must be contained in every task of the robot, that is, the name of the declaration as well as all the other data must be identical. Only the name of the routine that is to be called as well as the valid position could be different.

`ValidPosition`
Datatype: `num`
Position(s) in which the robot should be present, in order to be able to call the service routine or set up routine.

Value	Description
1	Home position (Bit 1)
2	Safe position (Bit 2)
4	Service position 1 (Bit 3)
8	Service position 2 (Bit 4)
16	Service position 3 (Bit 5)
255	Any position (arbitrary)

`ValidPosition` is bit coded, so that several valid positions can be grouped together (for example, Home position or safe position -> ValidPosition =3)

`RunInAutoMode`
(Only for service rouines of Type I)
Datatype: `bool`
TRUE, if the service routine is to be executable in the automatic mode and in the manual mode of the robot.
FALSE, if the service routine is to be executable only in the manual mode of the robot.

`ProgCode`
(Only for service rouines of Type I)
Datatype: `dnum`
Unique program coding, with which the service routine can be called from outside (external call) (for example, PLC).
Here, only "service routines" of the Type I (`MenuType=1`) are called, the "RunInAutoMode" flag will not be considered.

`MenuType`
Datatype: `num`
Type of menu entry

Menu type	Description
1	Service-Routine Type I (Execution takes place through the RWMT-Engine)
2	Service-Routine Type II (Execution takes place in parallel with the program pointer)
3	Setup-Routines

Continues on next page

`MinUserLevel`	**Datatype:** `num` Minimum user level, which is necessary for displaying the routine in the service menu.

Restrictions

A program code may be used only once

The program code should not be identical with that of a processing program.

The size of the icon is 32x32 pixels. Larger images will be modified accordingly, but should be kept small owing to the limited size of the Flexpendant memory.

Structure

```
< Data object of menudata>
< Description of string >
< Category of string >
< Image of string >
< ProcName of string >
< RunInTasks of string >
< ValidPosition of num >
< RunInAutoMode of bool >
< ProgCode of dnum >
< MenuType of num >
< MinUserLevel of num >
```

15.1.18 msgdata – Message declaration

Usage

The data type `msgdata` is used to define a message that can be displayed through the instructions like `MT_WaitMsgDI, MT_WaitMsgGI, MT_WaitMsgGI32, MT_ShowMessage` and so on.

Description

Data of the data type `msgdata` contain an error domain number, an error number, a title, a message text, an icon, buttons as well as the path to an image.

Components

`Domain`
Datatype: `num`
Error domain number

`Number`
Datatype: `num`
Error number

`Type`
Datatype: `buttondata`
Specifies the buttons that are to be displayed. Only one of the pre-defined button combinations of the type buttondata may be used.

```
!Buttons:
CONST buttondata btnNone := -1;
CONST buttondata btnOK := 0;
CONST buttondata btnAbrtRtryIgn := 1;
CONST buttondata btnOKCancel := 2;
CONST buttondata btnRetryCancel := 3;
CONST buttondata btnYesNo := 4;
CONST buttondata btnYesNoCancel := 5;
```

If the selected type is btnNone, then the message is shown embedded in the graphical user interface, otherwise the standard UI-dialog is used..

`Header`
Datatype: `string`
The heading of the message window (max. 40 characters).

`Text 1`
Datatype: `string`
Text row 1, which will be shown on the display. Max. 55 characters.

`Text 2`
Datatype: `string`
An additional, second row of text, which will be shown on the display. Max. 55 characters.

`Text 3`
Datatype: `string`
An additional, third row of text, which will be shown on the display. Max. 55 characters.

`Text 4`
Datatype: `string`
An additional, fourth row of text, which will be shown on the display. Max. 55 characters.

`Text 5`
Datatype: `string`
An additional, fifth row of text, which will be shown on the display. Max. 55 characters.

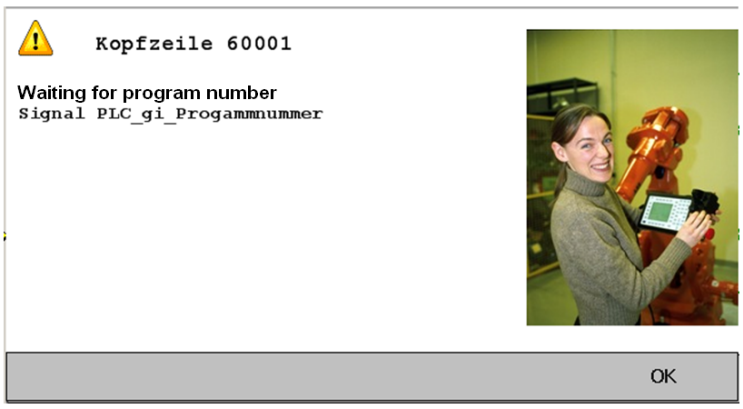
Continues on next page

`Icon`	Datatype: `icondata` Defines the icon that is to be displayed. Only one of the pre-defined icons of the type icondata may be used. Please also refer to the Rapid Reference Manual. Depending on the selected item, the message is shown as info (green background), warning (yellow background) oder error (red background). By default, no icon will be displayed.
`Image`	Datatype: `string` The name of the user-defined image, which will be shown beside of the message. To display your own images, they must be saved in the HOME: directory or the `SYSTEM:` directory of the active system. It is advisable to save the files in the HOME: directory, so that they will be saved during a backup and restore procedure. A hot start is necessary. After this, the FlexPendant will load the new images. The image that is displayed can have a breadth of 185 pixels and a height of 300 pixels. If the image is larger, only a range of 185 x 300 pixels of the image will be displayed, from the top left corner of the image. It is not possible to specify a more accurate value for the permitted size of an image or the number of images that can be loaded on to the FlexPendant. This depends on the size of other files, which have been loaded on to the FlexPendant. The program execution will be continued if an image which has not been loaded on to the FlexPendant is used.

Basic examples

```
CONST msgdata msgProgNumber:=[6,1,btnOK,"Header",
"Waiting for program number","Signal PLC_gi_Progammnummer", "",
     "","", iconWarning,"Happy.jpg"];
MT_WaitMsgGI giProgNumber, NOTEQ, 0, msgProgNumber;
```

Display of a text message after a certain time. While displaying a text message, an additional image, an icon and the button will be displayed, in accordance with the specifications in the data `msgProgNumber`.



en1200000785

Structure

```
< Dataobject of msgdata>
< Domain of num >
```

Continues on next page

15.1.18 msgdata – Message declaration
Continued

```
< Number of num >
< Type of buttondata >
< Header of string >
< Text1 of string >
< Text2 of string >
< Text3 of string >
< Text4 of string >
< Text5 of string >
< Icon of icondata >
< Image of string >
```

15.1.19 partcodes – Check codes for a part

Usage

`partcodes` is used to define the check codes for a part in the `partdata` (see *partdata – Part data on page 284*) declarations.

Description

`partcodes` is used exclusively through the data type `partdata`.

The eight check codes (test codes) correspond to the eight check code group inputs which are set in the system parameters `PROC/ MT_PRG_SELECT`. (See also *MT Program Selection on page 159*).

Components

`Code1`	**Datatype:** `dnum` **Test code 1**
`Code2`	**Datatype:** `dnum` **Test code 2**
`Code3`	**Datatype:** `dnum` **Test code 3**
`Code4`	**Datatype:** `dnum` **Test code 4**
`Code5`	**Datatype:** `dnum` **Test code 5**
`Code6`	**Datatype:** `dnum` **Test code 6**
`Code7`	**Datatype:** `dnum` **Test code 7**
`Code8`	**Datatype:** `dnum` **Test code 8**

Structure

```
< Dataobject of partcodes>
< Code1 of dnum > < Code2 of dnum >
< Code3 of dnum > < Code4 of dnum >
< Code5 of dnum > < Code6 of dnum >
< Code7 of dnum > < Code8 of dnum >
```

15.1.20 partdata – Part data

Usage

`partdata` is used to define the data for a part which are required for processing it.

Description

Every part requires a part declaration, which contains for example, the name, the processing routine as well as the required coding (program number, gripper code and a max. of 8 additional check codes (test codes)) for starting the processing of the part.

The file name can still be specified in an image file, which can be displayed in the parts view of the GUI.

If the program code does not correspond to the required coding that is to be used in the robot program, then the parameter "`TypeCode`" can be used.

Through this, for instance, a part can be called using the program number "12" by the PLC and the type code "4711" will be used in the robot program for the routines and positions.

The parameter “AuxCode” is a unique code as the program code and the type code and allows to address for example, a vision system, which might have different numbers for part types than a PLC or the robot.

The function `MT_GetAuxCode` (see *MT_GetAuxCode – Reading the auxiliary code of the current part type on page 463*)provides the auxiliary code to the user program.

Through the parameter `AdvPart`, the data name of a data declaration belonging to a part, and which contains more part specific data, can be specified.

For this, a data type such as `partadv`, which contains the required data contains and which creates a declaration of this data type for every part, must be declared.

Example:

```
RECORD partadv
num Value1;
num Value2;
ENDRECORD
CONST partadv pdvPart_T1:=[10.5];

CONST partdata pdPart_T1:=["Bauteil 1","Part_T1","",TRUE,
     1,4,3,1,[0,0,0,0,0,0,0,0],"Part1.GIF",
[1.5,[0,0,0.001],[1,0,0,0],0,0,0],"pdvPart_T1"];
```

The extended data for the current part is got through the following function "`GetCurrPartAdv`":

```
nValue1:=GetCurrPartAdv().Value1;
```

Basic example

```
CONST partdata pdPart_T1:=["Bauteil 1","Part_T1","",
TRUE,1,4,3,-1,[-1,-1,-1,-1,-1,-1,-1,-1],
"Part1.GIF",[1.5,[0,0,0.001],[1,0,0,0],0,0,0],""];
```

Continues on next page

Declaration for the part with program number 1, part type number 4 and auxiliary number 3.

Components

`Description`	Data type: `string` Name of the part
`Routine`	Data type: `string` Name of the processing routine
`RunInTasks`	Data type: `string` `RunInTask` is used within the GUI to select the part in the specified tasks for execution.
`NoCycles`	Data type: `bool` `TRUE`, part does not use any program cycles. `FALSE`, part uses program cycles.
`ProgCode`	Data type: `dnum` Program number through which the part is started by the external part selection
`TypeCode`	Data type: `dnum` Type coding that can be used in the robot program.
`AuxCode`	Data type: `dnum` Alternative program number to address for example, a vision system which uses program numbers that are different from the robot's program numbers.
`ToolCode`	Data type: `num` Gripper coding. If a gripper coding greater than "-1" is used, then the part will be processed only if the group input for the gripper code has the corresponding value.
`CheckCode`	Data type: `partcodes` Eight additional check codes (for example, form code). If a value greater than "-1" is used for a check code, then the part will be processed only if the corresponding group input for the check code has the corresponding value.
`Image`	Data type: `string` The file name of the image of the part, having a size of 270x270 pixel (.JPG,.GIF or.PNG). For this, the image must be present either directly in the "HOME:" directory or in the "System:" directory or in the directory.
`PartLoad`	Data type: `loaddata` Load of the part. This is used as the default load, if the instructions MT_GripSet and MT_GripSequence do not provide their own optional loaddata argument.
`AdvPart`	Data type: `string` Reference (name of the constant or persistent entity) for an extended part data declaration (`partadv`).

Continues on next page

15.1.20 partdata – Part data
Continued

Structure

```
< Dataobject of partdata>
< Description of string >
< Routine of string >
< RunInTasks of string >
< NoCycles of bool >
< ProgCode of dnum >
< TypeCode of dnum >
< AuxCode of dnum >
< ToolCode of num >
< CheckCode of partcodes >
< Image of string >
< PartLoad of loaddata >
< AdvPart of string >
```

15.1.21 posname – Assigning position description for HomeRun

Usage

`posname` is used to assign a description to each position number, when using *HomeRun* (see *HomeRun on page 105*).

Description

If triggered, HomeRun shows a user menu, in which the operator finds information about, where the robot is currently located and where it goes next to move back to home position.

For this purpose, the menu shows the position numbers and a related descriptive text.

This information is obtained from a declaration of the type posname.

Conventions:

- This declaration has to be named `pnPositions`,
- The declaration must be located in any user module,that is, `MT_MAIN`
- It may not be declared as LOCAL.
- The declaration must be an array which must contain 10 (or a multiple of) 10 elements.

Basic examples

Example:

```
CONST posname pnPositions{10}:=
[
[100,"Prepos IMM"], [101,"Grip pos IMM"], [102,"Endpos IMM"],
[200,"Prepos conveyor"],[201,"Drop pos conveyor"],
[202,"Endpos conveyor"],[0,""],[0,""],[0,""],[0,""]
]
```

This information will be shown in the HomeRun menu, if a home run is requested.

Components

`Position`	**Data type:** `num` Number of the position
`Name`	**Data type:** `string` Description of the position

Structure

```
< Dataobject of posname >
< Position of num >
< Name of string >
```

15.1.22 projectinfo – Project definition for grapical user interface

Usage

`projectinfo` is used to define a RWMT project.

Description

A “RWMT project” describes, which RAPID modules and signal parameters belong to an application. A project file is normally to be created through the *Machine Tending PowerPac* (see *Operating Manual - Machine Tending PowerPac* listed in the section *References on page 11*), which is also used to download the project to the robot controller.

The RWMT GUI provides the possibility to load, unload, import and export so called ‘projects’.

To enable the RWMT GUI to do an assignment between program modules and system modules, being loaded into controller RAM, and a RWMT project, the data type `projectinfo` is used.

In a SingleMove application, one (and only one) declaration of the data type `projectinfo`has to be implemented in any program module.

In a MultiMove application, one (and only one) declaration of the data type projectinfo has to be implemented in any program module in task T_ROB1.

To learn more about projects, please refer to the *Operating Manual - RobotWare Machine Tending* and the *Operating Manual - Machine Tending Power Pack* listed in the section *References on page 11*.

Basic examples

Example:

```
CONST projectinfo piProject:=
[
"DCM_Buehler_01",
"Unloading of DCM, cooling, loading of trimming press",
"1.0",
"2012-04-29"
]
```

This information will be shown in project manager of the RWMT GUI.

Components

`Title`	Data type: `string` Title of the project. It must not contain empty spaces and should be a brief name because it is used as the name of a project folder on the robot controller.
`Details`	Data type: `string` Description of the project.
`Version`	Data type: `string` Version of the project.

Continues on next page

`Date`	**Data type:** `string` **Date of project creation or modification.**

Structure

```
< Dataobject of projectinfo>
< Title of string >
< Details of string >
< Version of string >
< Date of string >
```

15.1.23 signalpage – Definition of a signal page for the GUI

Usage

`signalPage` is used to define a tab pane for displaying a max.of 20 signals in the GUI.

Description

In the signal window of the GUI, up to eight tab panes containing digital inputs and outputs, group inputs and outputs or analog inputs and outputs can be displayed.

The outputs can be set or reset by the operator in the manual mode of operation. In the automatic mode, the respective signal should have the corresponding permissions (Accesslevel in EIO.CFG) so that it can be set by the GUI.

In order to be able to accommodate more than 20 signals on a tab pane, muss another signal page with the same `PageName` element must be declared.

Basic examples

Example 1

```
CONST signalpage spEUROMAP1:=["Euromap 67",1,
"diIMM_MouldOpen","diIMM_PusherRetracted","diIMM_PusherFwd",
"diIMM_Core1Retr","diIMM_ Core1Fwd","diIMM_Scrap",
"diIMM_Automatic","diIMM_MouldClosed","diIMM_MouldInterPos",
"diIMM_Core2_Retr","diIMM_Core2_Fwd","diIMM_Manufact1",
"","","","","","","","","];
```

Display inputs of the EUROMAP 67 interface of an injection moulding machine.

Example 2

```
CONST signalpage spEUROMAP2:=["Euromap 67",1,
"doIMM_CloseMould","doIMM_NoIRB","doIMM_OpenMould",
"doIMM_RetrPusher","doIMM_FwdPusher","doIMM_RetrCore1 ",
"doIMM_FwdCore1","doIMM_FwdCore2","doIMM_RetrCore2",
"","","","","","","","","","","","","];
```

Since the EUROMAP 67 interface consists of more than as 20 signals, the outputs are declared in a second signalpage declaration with identical page name, so that all the signals appear in one tab pane.

Components

`PageName`	Data type: `string` Name of the tab pane. If there are several declarations with the same page name, then, their signals will be displayed in only one page.
`PageIndex`	Data type: `num` Index for determining the position of the tab pane in the signal window. If several pages are found with the same page index, then the sequence is determined with the help of the page name.
`Signal1`	Data type: `string` Signal name as per the signal configuration (EIO.CFG).

Continues on next page

`Signal2` Data type: `string`
Signal name as per the signal configuration (EIO.CFG).

`Signal3` Data type: `string`
Signal name as per the signal configuration (EIO.CFG).

`Signal4` Data type: `string`
Signal name as per the signal configuration (EIO.CFG).

`Signal5` Data type: `string`
Signal name as per the signal configuration (EIO.CFG).

`Signal6` Data type: `string`
Signal name as per the signal configuration (EIO.CFG).

`Signal7` Data type: `string`
Signal name as per the signal configuration (EIO.CFG).

`Signal8` Data type: `string`
Signnal name as per the signal configuration (EIO.CFG).

`Signal9` Data type: `string`
Signal name as per the signal configuration (EIO.CFG).

`Signal10` Data type: `string`
Signal name as per the signal configuration (EIO.CFG).

`Signal11` Data type: `string`
Signal name as per the signal configuration (EIO.CFG).

`Signal12` Data type: `string`
Signal name as per the signal configuration (EIO.CFG).

`Signal13` Data type: `string`
Signal name as per the signal configuration (EIO.CFG).

`Signal14` Data type: `string`
Signal name as per the signal configuration (EIO.CFG).

`Signal15` Data type: `string`
Signal name as per the signal configuration (EIO.CFG).

`Signal16` Data type: `string`
Signal name as per the signal configuration (EIO.CFG).

`Signal17` Data type: `string`
Signal name as per the signal configuration (EIO.CFG).

`Signal18` Data type: `string`
Signal name as per the signal configuration (EIO.CFG).

`Signal19` Data type: `string`
Signal name as per the signal configuration (EIO.CFG).

`Signal20` Data type: `string`
Signal name as per the signal configuration (EIO.CFG).

Structure

```
< Dataobject of signalpage>
< PageName of string >
< PageIndex of num >
< Signal1 of string > < Signal2 of string >
< Signal3 of string > < Signal4 of string >
```

Continues on next page

15.1.23 signalpage – Definition of a signal page for the GUI
Continued

```
< Signal5 of string > < Signal6 of string >
< Signal7 of string > < Signal8 of string >
< Signal9 of string > < Signal10 of string >
< Signal11 of string > < Signal12 of string >
< Signal13 of string > < Signal14 of string >
< Signal15 of string > < Signal16 of string >
< Signal17 of string > < Signal18 of string >
< Signal19 of string > < Signal20 of string >
```

15.1.24 stationapp – External applications to be started in GUI

Usage

`stationapp` is used to define external applications, that can be started through the GUI menu.

Description

Since there might be functionalities for stations, that are not provided by RWMT but nevertheless needed by the customer, RWMT provides an interface to start external FlexPendant FlexPendant applications in a specific station's sub-view.

These applications can be programmed by the integrator or customer, using the *Flexpendant SDK* or *ScreenMaker* (for more information, see F*lexpendant SDK* and *ScreenMaker* manuals listed in the section *References on page 11*.

Conventions:

Declarations of type `stationapp` have to keep some conventions, so that the GUI can assign the application to the appropriate station view.

- The name must start with a prefix, which is the same as the name element of the related `stationdata` declaration.
- The prefix is followed by underline and the postfix "Applications"
- The declarations have to be arrays. The minimum array size is "1" if there is only one application for a specific station, the maximum array size is "8" for at least 8 customized applications for a station.
- The entries in the declarations are case sensitive.

Basic example

Assuming, there are 3 different applications, belonging to the station IMM (Injection Moulding Machine):

Related station data is:

```
LOCAL PERS stationdata IMM_Station:=
[
"IMM","SGM","Machine to build plastic parts",
"station-IMM.png","IMM_sdiEn_OPMode",
"IMM_sdiMouldClosed","","",TRUE,FALSE,1,1
];
```

3 applications to be assigned to the IMM station:

```
LOCAL CONST stationapp IMM_Applications{3}:=
[
["Prog 1","","TPSViewMyApp1.dll","MyApp1","MainScreen"],
["Prog 2","","TPSViewMyApp2.dll","MyApp2","Page1"],
["Prog 3","","TPSViewMyApp3.dll","MyApp3","Frogpage"]
];
```

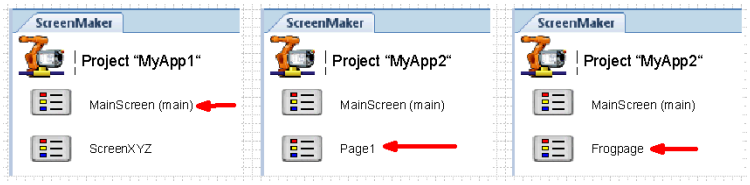
We have 3 different ScreenMaker apps that can be started. They will get the entries "Prog1", "Prog 2" and "Prog 3" in the RWMT GUI inside the station view of the IMM station.

Continues on next page

15.1.24 stationapp – External applications to be started in GUI
Continued

The namespace (= project name: “MyApp1”, “MyApp2”, “MyApp3”) of each application can be retrieved from the project view of ScreenMaker, as well as the screen of each project that shall be shown (“MainScreen”, ”Page1”, “Frogpage”).

Related project views in ScreenMaker for this example:



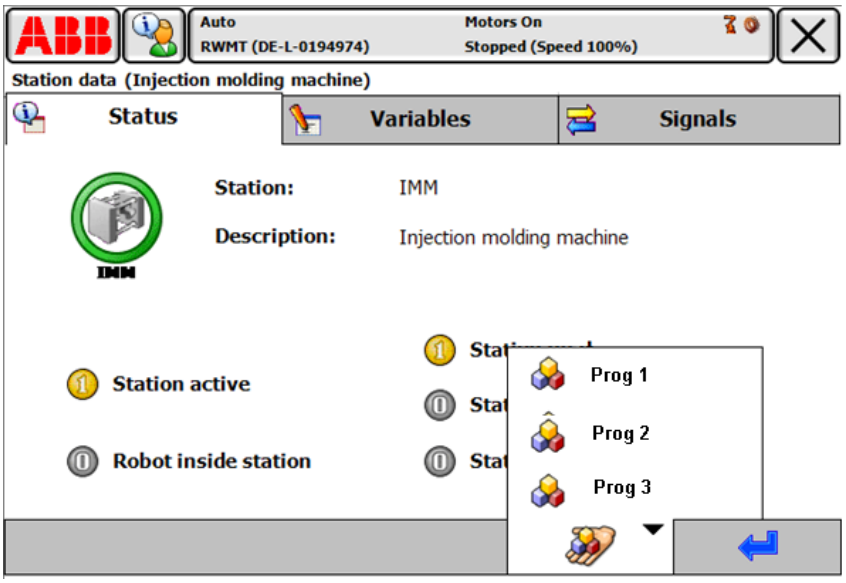
en1200000786

If a FlexPendant application shall be integrated, then the namespace (project name) and the screen of the project to be called can be retrieved directely from the code as shown in the example below, where the namespace is “TpsViewExample” and the screen to be shown is “ViewSettings”.

Example:

```
namespace TpsViewExample
{
public class ViewSettings : TpsForm, ITpsViewActivation
{
…
```

The following view is the result of the declaration of the example above:

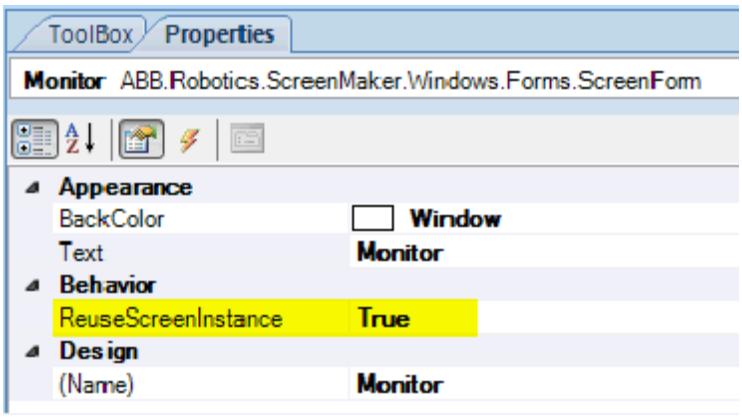


en1300000153

Continues on next page

Limitations and characteristics

- In the ScreenMaker view of RobotStudio the property “ReuseScreenInstance” of the screen which is to be opened, must be set to TRUE.



en1300000265

- If there is just one station application for a specific station, its name, respectively its icon will be shown directely in the menu bar in the GUI station view.
- To be able to run an embedded applications on a virtual controller, the application files (DLL’s) must be located in the “Home” or “System” directory. At the real controller there a no limitations.
- The embedded applictions should have a button which closes the view, otherwise, there is no way to close the application.
- (FP-SDK): The app should use error handling, so that each error will be handled by the application itself.
- (FP-SDK): The constructor of the application should be the standard constructor which has no parameters. Each application must create the required data by itself.
- (FP-SDK): When closing the application, all used resources must be released by means of the “Dispose” method, so that no memory leaks appear.
- (FP-SDK): The Interface “ITpsViewActivation“ should be implemented, so that the methods “Activate” and “Deactivate” will be called if the control goes from the passive state to the active state, i.e. becomes visible in the client view. Normally, this is where subscriptions to controller events are set up.
- (FP-SDK): Only “TpsForms” can be used.

Components

`MenuName`
Data type: `string`
The name of the application that appears in the station menu.

`Image`
Data type: `string`
The image of the application that appears in the station menu.

`DLLname`
Data type: `string`
Name of the DLL to be launched.

Continues on next page

15.1.24 stationapp – External applications to be started in GUI
Continued

`NameSpace`	Data type: `string` Namespace (project name) of the application.
`Class`	Data type: `string` View of the application that shall be started.

Structure

```
< Dataobject of stationapp >
< MenuName of string >
< Image of string >
< DLLname of string >
< NameSpace of string >
< Class of string >
```

15.1.25 stationdata – Definition of a station

Usage

`stationdata` is used to display a station in the GUI with all the status information.

Description

The station data contain the station name, the station image as well as the signal definitions for the states "Ready", "Busy", "Error" and "Station deselected".

This data is used for visualizing the station in the production view.

In the robot program, this data is used to display the active station in the GUI (see the instruction *MT_SetActiveStation – Set station symbol to "active" on page 408*) as well as to check if the station has been selected or deselected (see function *MT_StationIsEnabled – Checking station pre-selection for production on page 481*).

Properties

Station declarations must be declared as **PERS**, so that these can be modified by the GUI.

Basic example

```
PERS stationdata sdIMM:=
["IMM","","Injection Moulding machine",
"station-IMM.png", "diIMM_Automatic","*diIMM_MouldOpen",
     "","",TRUE,FALSE,2,1];
```

Station declaration for an injection moulding machine that cannot be deselected, and the signals for the "Ready" and "Busy" status.

Components

`Name`	Data type: `string` Name of the station is used as a prefix for the station variables and station signal definition. It is mandatory to use a prefix. Maximum length: 8 characters
`Label`	Data type: `string` Label text of the station symbol in the production window. If the text for the label is blank, then the Name of the station will be displayed. The station label can be used to display the label in the GUI in another language, while the name of the station remains the same. Maximum length: 10 characters
`Description`	Data type: `string` Description of the station.
`Image`	Data type: `string` File name of the station symbol of the type ".GIF", ".JPG" or ".PNG". Own station symbols must be contained in the directory "HOME:", "SYSTEM:".

Continues on next page

15.1.25 stationdata – Definition of a station
Continued

`ReadyState`

Data type: `string`

Signal definition for the station status "Ready".

For the definition of a status message, digital inputs and outputs, as well as Boolean persistent entities can be used.

To link several signals for a status message, the following notation must be used:

*	Invert the signal or the Boolean persistent entity
&	AND connection
!	OR connection

The restriction to 80 characters for a string should be adhered to while selecting the name of the signal, that is, the longer the names of the signals, the fewer the number of signals that can be joined together.

Note

If a blank string is used for the "ReadyState", then, this will be interpreted as "Station is ready".

Example:

```
Station.ReadyState:="diIMM_Automatic &
diIMM_MouldOpen"
```

`BusyState`

Data type: `string`

Signal definition for the station status "Busy" (See ReadyState).

Note

If a blank string is used for the "BusyState" then, this status will not be considered in the evaluation.

`ErrorState`

Data type: `string`

Signal definition for the station status "Error" (See ReadyState).

Note

If a blank string is used for the "ErrorState" then, his status will not be considered in the evaluation.

`ExtEnable`

Data type: `string`

Name of the digital inputs or output through which the station is selected or deselected. Here, the "high" state of the signal means "Station selected" and the "low" state means "Station deselected".

If no signal name is defined, then the station is selected or deselected through the GUI.

To invert the function of the signal, the characters "*" should be added before the name of the signal.

Example:`Station.ExtEnable:="*diWithoutMachine";`

`Enabled`

Data type: `bool`

Station has been selected (TRUE) or deselected (FALSE).

Continues on next page

AllowDisable	Data type: bool The selection or deselection of the station through the GUI is allowed (TRUE) or not allowed (FALSE). If a signal is used for selecting or deselecting the station (See ExtEnable), then this parameter will be ignored and the station selection cannot be modified through the GUI.
Column	Data type: num Column of the station symbol in the production window (Permitted values: 1-5)
Row	Data type: num Row of the station symbol in the production window (Permitted values: 1-3)

Structure

```
< Dataobject of stationdata >
< Name of string >
< Label of string >
< Description of string >
< Image of string >
< ReadyState of string >
< BusyState of string >
< ErrorState of string >
< ExtEnable of string >
< Enabled of bool >
< AllowDisable of bool >
< Column of num >
< Row of num >
```

15.1.26 stationsignal – Allocation of station signals to alias names

Usage

`stationsignal` is used to assign a station signal (`signaldi, signaldo, signalgi, signalgo, signalai` or `signalao`) to an alias name, which is used in a station module.

Description

Several stations, which differ in their function only in terms of the signals used, can exist in a robot cell (for example, loading of several conveyor belts)

In order to be able to create a station module as a generally applicable template, the station signals are accessed within the robot program through alias names.

The allocation of the signals of the respective station to the alias names in the program module is done through the *MT_ALIASIO* instruction (for more details, see *MT_AliasIO – Connecting of alias signals on page 307*).

In order that this allocation can take place in an automated manner in the robot program, and so that even the GUI can access the station status signals, this signal allocation takes place in a station specific manner through an array of the data type `stationsignal`.

This data type provides an additional description to clarify the meaning of the signal. This is used, when editing a project in the *MachineTending Power Pack*.

Basic example

```
MODULE CNV1
!Digital signals
LOCAL VAR signaldi adiReadyToLoad;
LOCAL VAR signaldo adoIRBOutOfArea;
LOCAL VAR signaldo adoStartCNV;

!IO mappings for signals / alias signals
LOCAL CONST stationsignal CNV1_Signals{3}:=[
["Conveyor loading release","diLoadCNV",
"adiReadyToLoad"],
["Robot outside conveyor","doIRBOutOfCNV1",
"adoIRBOutOfArea"],
["Start conveyor","doCNV_Start",adoStartCNV"]];
```

Allocation of the signals of the first conveyor belt.

Components

`Description`	Data type: `string` Description of the signal's purpose
`Signalname`	Data type: `string` The signal name from the E/A-configuration

Continues on next page

`AliasName`	**Data type:** `string` The variable name of the signal that has been declared in the RAPID-program.

Structure

```
< Dataobject of station signal >
< Description of string >
< Signalname of string >
< AliasName of string >
```

The GUI shows the new signal description within the station signal view. If this description is empty the GUI uses the signal label in the EIO.CFG.

15.1.27 stationvariable – Display the data declarations of a station

Usage

`stationvariable` is used to display the data declarations of the type "`bool`", "`num`", `dnum`" or "`string`" in a station data page of the GUI or to modify their values.

Description

Station specific data declarations (variables or persistent entities) are displayed in the station view and can be modified depending on the parameterization, constants can only be displayed.

In opposite to variable entities, the persistent entities are updated automatically.

In order to be able to modify the values of data declarations, the user should have logged in to the controls with the user permission "`MT_VAR_WRITE`" and the corresponding "`MinUserLevel`".

In order to be able to modify the values by using the "Reset" button, the user should have logged in to the controls with the user permission "`MT_VAR_RESET`".

Basic example

```
MODULE IMM
LOCAL PERS num nIMM_UnloadTime:=0;
LOCAL PERS bool bIMM_WithCorePullers:=FALSE;
LOCAL PERS num nIMM_WaitTime:=5;
TASK PERS num nIMM_PartCounter:=0;

CONST stationvariable IMM_VARIABLES{4}:=[
["Unloading time","nIMM_UnloadTime","IMM","",0,0,FALSE,
FALSE,FALSE,0,0],

["With Core Pullers","bIMM_WithCorePullers","IMM","",0,0,
TRUE,FALSE,FALSE,0,100],

["Waiting Time","nIMM_WaitTime","IMM","",0,20,
TRUE,TRUE,FALSE,0,10],

["Part Counter","nIMM_PartCounter","IMM","",0,0,
FALSE,TRUE,TRUE,0,10]];

ENDMODULE
```

In the station "IMM", four data declarations are displayed as follows:

The numerical persistent entity `nIMM_UnloadTime` will be displayed, but cannot be modified.

The Boolean persistent entity `bIMM_WithCorePullers` will be displayed as a selection field and can be modified by users who have the `MinUserlevel` of at least 100 in the manual mode of the robot.

The numerical persistent entity `nIMM_WaitTime` can be modified in the automatic and manual modes of operation of the robot in the range between 0 and 20 by users having the `MinUserlevel` of at least 10.

Continues on next page

The numerical persistent entity `nIMM_PartCounter` can be set in the automatic and manual modes of operation of the robot with the "Reset" button to the value 0 if the user has the `MinUserlevel` of at least 10.

Components

`Description`
Data type: `string`
Descriptive text that will be displayed in the GUI.

`VariableName`
Data type: `string`
Name of the variable, persistent entity or constant

`ModuleName`
Data type: `string`
Name of the module in which the `VariableName` has been declared.
If no module name has been specified, then the data declaration will be searched in the specified task.

`TaskName`
Data type: `string`
Name of the task in which the VariableName has been declared.
If no task name is specified, then the task name that is contained in the `stationvariable declaration` will be used.

`MinValue`
Data type: `dnum`
Lower limit for the "**num**" or "**dnum**" data declaration

`MaxValue`
Data type: `dnum`
Upper limit for the "**num**" or "**dnum**" data declaration.

Tip

If MinValue and MaxValue are identical (for example, "0", then no limitation is used.

`Editable`
Data type: `bool`
TRUE, if the value of the data declaration can be changed in the GUI.
FALSE, if the value of the data declaration is meant only for display in the GUI.

CAUTION

Constants cannot be edited

`ChangeInAuto`
Data type: `bool`
TRUE, if the value of the data declaration can be changed in the GUI in the automatic mode.
FALSE, if the value of the data declaration can be changed only in the manual mode in the GUI.

`ResetButton`
Data type: `bool`
TRUE, if a "Reset" button is to be displayed in the GUI, using which the "**num**" or "**dnum**"-data declaration can be set to the previously set value.
FALSE, if no "Reset" button is to be used.

`ResetValue`
Data type: `dnum`
Value that is to be assigned on activating the "Reset" button to the "num" or "dnum"-data declaration.

Continues on next page

15.1.27 stationvariable – Display the data declarations of a station
Continued

`MinUserLevel`	Data type: `num` Minimum user level that is necessary in order to be able to change the data declaration. If the user permissions are not enough, then only the value of the data declaration will be displayed.

Structure

```
< Dataobject of stationvariable >
< Description of string >
< VariableName of string >
< ModuleName of string >
< Taskname of string >
< MinValue of dnum >
< MaxValue of dnum >
< Editable of bool >
< ChangeInAuto of bool >
< ResetButton of bool >
< ResetValue of dnum >
< MinUserLevel of num >
```

15.1.28 userbutton – User button on the Touchscreen

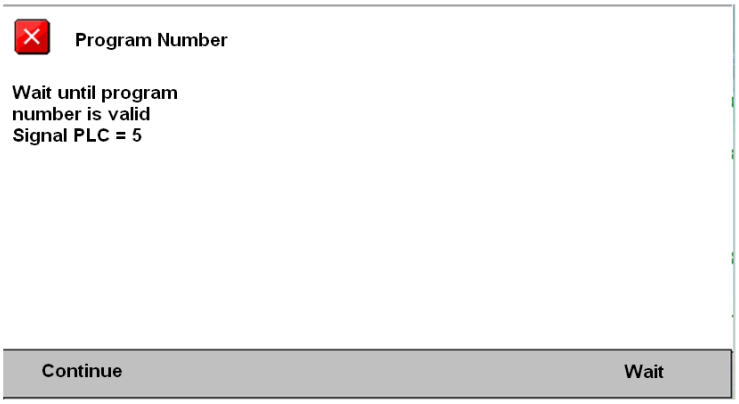
Usage

The data type `userbutton` is used for assigning separate texts to the function keys in a dialog box, which is displayed with MT_UIMessage.

Basic examples

```
Const userbutton btntest:=["Continue”,"","","","Wait"];
MT_UIMessage msgProgTestUserBtn\BtnArray:=btntest;
```

Display of a text message with the function keys "Continue" and "Wait".



en1200000788

Structure

```
< Dataobject of userbutton>
< Button1 of string >
< Button2 of string >
< Button3 of string >
< Button4 of string >
< Button5 of string >
```

15.1.29 versiondata – Version data of the application module

Usage

`versiondata` is used to capture the version information of the program modules or system modules of the applications.

Description

The version information of the applications as well as of the program modules and system modules are saved in data declarations, so that these can be called easily.

The version data must be modified in the event of changes to the modules.

Basic example

```
CONST versiondata vdMyModule:=["MY_MODULE","1.0","2012-05-03"];
```

The module "MY_MODULE" is located in the Version "1.0" dated "2012-05-03" in the robot controls.

Components

`ModuleName`	Data type: `string` Name of the RAPID-Module (e.g: "MY_MODULE")
`Version`	Data type: `string` Serial version number of the module (for example, "1.1")
`Date`	Data type: `string` Date of change (for example, "2012-05-03").

Structure

```
< Dataobject of versiondata>
< ModuleName of string >
< Version of string >
< Date of string >
```

15.2 Instructions

15.2.1 MT_AliasIO – Connecting of alias signals

Usage

`MT_AliasIO` connects all alias signals, that are inside a `stationsignal` (see *stationsignal – Allocation of station signals to alias names on page 300*) declaration to the appropriate physical signals that are also part of this declaration.

Basic Example

```
MODULE CNV1
!Digital signals
LOCAL VAR signaldi adiReadyToLoad;
LOCAL VAR signaldo adoIRBOutOfArea;
LOCAL VAR signaldo adoStartCNV;
!IO mappings for signals / alias signals
LOCAL CONST stationsignal CNV1_Signals{3}:=[
[”CNV1 loading release”,"diLoadCNV","adiReadyToLoad"],
[”Robot out of CNV1”"doIRBOutOfCNV1","adoIRBOutOfArea"],
[”Start conveyor CNV1”,"doCNV_Start",adoStartCNV"]];
…
LOCAL PROC Init()
!Connect alias signals to their physical representations.
MT_AliasIO CNV1_Signals\ModuleName:=CNV1;
ENDPROC
…
ENDMODULE
```

Program Execution

`MT_AliasIO` has to be called with the desired `stationsignal` declaration before the depending alias signals can be used in the robot program.

Arguments

```
MT_AliasIO iomaplist \ModuleName
iomaplist Data type: stationsignal
```

List with alias signals and their physical representations.

```
ModuleName Data type: string
```

Name of the module, which contains the stationsignal declaration.

Syntax

```
MT_AliasIO
[ iomaplist ’:=’ ] < expression (IN) of stationsignal >
[ ’\’ ModuleName ’:=’ < expression (IN) of string > ] ’;’
```

15.2.2 MT_ChangeTool – Changing the current tool

Usage

The procedure MT_ChangeTool is used, if the part does not fit the gripper, which is mounted on the robot, after a program number has been passed.

Using the handed over gripper code, a manual or automatic tool change can be executed.

This routine can be used by the application programmer to change a wrong tool by the appropriate one.

The routine has to be created by the application programmer and has to be filled with reasonable code for gripper changing actions.

Basic example

```
!Part type 1
CONST partdata pdPart_T1:=
["Part type 1"," Production ","", TRUE, 1,-1,-1,3,
[-1,-1,-1,-1,-1,-1,-1,-1], "Part1.GIF",
[1.5,[0,0,0.001],[1,0,0,0],0,0,0],"pdvPart_T1"];

PROC MT_ChangeTool(
VAR signalgi CurrToolCode,
num ReqToolCode)
!Undock the current tool
UndockCurrentTool;
!Dock the required tool
DockReqTool ReqToolCode;
ENDPROC
```

In the routine MT_ChangeTool first the current tool is undocked and then the required one is docked.

Program run

Before calling the production routine for the selected part, RWMT will check if a tool code has been assigned to this and if this matches with the current tool code.

If this is not the case, then the routine MT_ChangeTool is called, which will offer the system programmer the possibility to switch to the correct tool.

After this, RWMT once again checks the current tool. If this is still not the same as the one required for the production, then an error message is output.

Syntax

```
MT_ChangeTool
[ CurrToolCode ’:=’ ] < expression (VAR) of signalgi > ’,’
[ ReqToolCode’:=’ ] < expression (IN) of num > ’;’
```

15.2.3 MT_ClearMessage – Delete message on the RWMT user interface

Usage

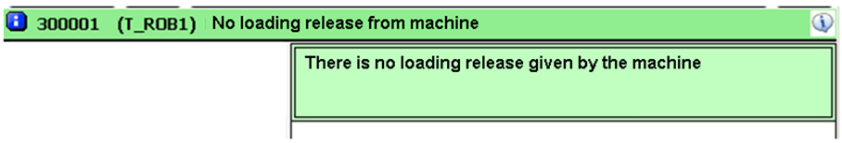
With the instruction `MT_ClearMessage`, a message that has been displayed on the RWMT user interface with the help of `MT_ShowMessage` can be deleted.

The instruction `MT_ShowMessage` is also available for background tasks.

Basic example

```
const msgdata msgLoadMachine:=[30,1,0,"No loading release from
     machine","There is no loading release given","by the
     machine.","","","",1,""];
!Show message that the machine is not ready for loading
MT_ShowMessage msgLoadMachine;
…
…
!Delete message
MT_ClearMessage;
```

The program will output on the RWMT user interface the message to the effect that the machine is not ready for loading.



en1300000154

The message will be deleted by executing the `MT_ClearMessage` instruction.

Program execution

By using the instruction `MT_ShowMessage`, a message will be displayed on the RWMT user interface until a new message is output, or the message will be deleted again with the help of the instruction `MT_ClearMessage`.

Syntax

```
MT_ClearMessage ';'
```

More information

Information about	See
Data type `Msgdata`	*msgdata – Message declaration on page 280*
Displaying a message on the RWMT user interface	*MT_ShowMessage – Show message on the RW-MT user interface on page 413*

15.2.4 MT_ContHomeRun – Continue a movement routine

Usage

MT_ContHomeRun is used to be able to automatically continue a movement from an intermediate position within the "MT_HomeRun" routine.

Using the instruction outside the "MT_HomeRun" routine is not permitted!

Basic example

```
PROC MT_HomeRun(num Position)
TEST Position
CASE 10:
mv10_999
CASE ...
...
DEFAULT:
MT_ContHomeRun Position;
ENDTEST
ENDPROC
```

An arbitrarily started movement, which is identified by a four to six-digit path designation, is continued automatically.

Arguments

```
MT_ContHomeRun Position [\ModName] [\Prefix] [\Index] [|DIndex]
     [\NoAutoBackw] [\ERR]
```

`Position` Data type: `num`

Position or path designation from which the movement routine to be selected is determined.

`ModName` Data type: `num`

Name of the module where the required movement routines are located. This is only needed if the movement routines are declared as LOCAL.

`\Prefix` Data type: `string`

Type prefix. Only necessary for type-depending movement routines and only if the standard type prefix “T” is not used. Example of a type related declaration name: mv10_30_T137, if the prefix is “T” and the part type number of the current partdata declaration is 137

`[\Index]` Data type: `num`

Index to select movement routines type-dependently. The index is appended to the routine name with an underscore, using the standard type prefix or the explicitly specified prefix. Only one of the parameters \Index or \DIndex may be used.

`[\DIndex]` Data type: `dnum`

Index to select movement routines type-dependently. The index is appended to the routine name with an underscore, using the standard type prefix or the explicitly specified prefix. Only one of the parameters \Index or \DIndex may be used.

Continues on next page

[\NoAutoBackw]	Data type: switch If this switch is not set and the required movement routine does not exist, it is checked whether the movement routine exists in the opposite direction. Should this be the case, then this movement routine is executed backwards. If this movement routine does not exist either, then an error message is output.
[\ERR]	Data type: bool A variable that is set to TRUE if no movement routine for continuing the movement was found. If this optional variable is omitted, an error message is output.

Program execution

In order to obtain the path designation, the last two digits are taken from the six to eight-digit current intermediate position number and passed to the MT_HomeRun routine.

This four to six-digit path designation is passed to the MT_ContHomeRun instruction, which uses it to determine the name of the required movement routine.

Example:

```
Position = 1020 ( Movement routine "mv10_20"
```

Type index

If an index for type-dependent routines is transferred as optional argument (\Index), then all movement routines with this index are called up, paying attention to the standard type prefix.

Example:

```
TEST Position
CASE 10:
mv10_999;
DEFAULT:
MT_ContHomeRun Position\Index:=nTypeNo;
ENDTEST
```

If no index for type-dependent routines is transferred as optional argument (\Index), then all movement routines with the type index of the current part type (see *partdata – Part data on page 284*) are called up, paying attention to the standard type prefix.

If the position number contains the value 1020 and the variable nTypeNo (index) contains the value 5, the routine "mv10_20_5" is called. If the routine "mv10_20_5" does not exist, "mv10_20" is called automatically.

If a type prefix has been defined in the system parameters (for example, "T"), all routines with a "T" in front of the index are called up (for example, "mv10_20_T5").

Movement routines may exist either only with index or only without index (for example, "mv10_20" or "mv10_20_T1").

Type prefix

The \Prefix parameter is used if movement routines with different type prefixes are used in a program (for example, "T" for general and "G" for gripper-dependent movement routines).

Continues on next page

15.2.4 MT_ContHomeRun – Continue a movement routine
Continued

Proceed as follows in order to call up the movement routine with the required type prefix with MT_ContHomeRun:

The generally valid type prefix is defined in the system parameters.

An attempt is made to call up the movement routine with the standard type prefix by using the \ERR error variables.

If no movement routine is found, the error variable is set to TRUE and then MT_ContHomeRun is called up again with the alternative type prefix.

Example:

```
PROC MT_HomeRun()
VAR bool bError;
TEST Position
CASE 10:
mv10_999;

DEFAULT:
!Call up routines with standard prefix and
!current type number
MT_ContHomeRun Position\Index:=nTypeNo\ERR:=bError;
!If no movement routine is found,
!use alternative type prefix
IF bError THEN
!If no movement routine is found,
!an error message is issued
MT_ContHomeRun Position\Prefix:=”G”\Index:=nGripCode;

ENDTEST
ENDPROC
```

Type module

By default, the standard module name of the system parameters is used together with the standard type prefix (see also chapter *MT Part Settings on page 162*).

Example for the module name, if the current part type is 3:

In order to be able to use module-local movement routines, the module name can be passed to `MT_ContHomeRun` as an optional parameter.

Example:

```
PROC MT_HOMERUN(num Position)
TEST Position
…
DEFAULT:
MT_ContHomeRun Position \ModName:="PROG_"+ValToStr(nTypeNo);
ENDTEST
ENDPROC
MODULE PROG_1
LOCAL PROC mv10_20()
…
ENDPROC
ENDMODULE
```

Continues on next page

Automatic backwards processing

If movements are to be continued automatically and the required movement routine is only available in the opposite direction, the optional argument \NoAutoBackw should not be used.

As a result, the routine to the opposite direction is executed backwards.

If the position number contains 2010 and only the routine “mv10_20” exists, then the routine “mv10_20” is performed starting with the last position (20) backwards to the first position (10) (see also *MT_MoveRoutine – Execute a movement routine at HomeRun on page 389*).

Note

Here it must be noted that only HomeRun-specific movement instructions (for example, `MT_MoveL, MT_TriggL`) may be in this movement routine, since the robot may otherwise collide with the peripherals.

Movement continuation checking order

In order for all module-local and globally declared movement routines to be taken into consideration by the `MT_ContHomeRun` instruction, the following calling order is used in the absence of a movement routine, depending on the parameters that are used:

Routine	NoAutoBackw
Local movement routine with index (forwards)	☑
Local movement routine without index (forwards)	☑
Local movement routine with index (backwards)	☐
Local movement routine without index (backwards)	☐
Global movement routine with index (forwards)	☑
Global movement routine without index (forwards)	☑
Global movement routine with index (backwards)	☐
Global movement routine without index (backwards)	☐

Syntax

```
MT_ContHomeRun
[ Position ':=' ] < expression (IN) of num> ','
[ '\' ModName ':=' < expression (IN) of string > ]
[ '\' Prefix ':=' < expression (IN) of string > ]
[ '\' Index ':=' < expression (IN) of num > ]
[ '\' DIndex ':=' < expression (IN) of dnum > ]
[ '\' NoAutoBackw ]
[ '\' ERR ':=' < expression (INOUT) of bool > ]';'
```

Continues on next page

15.2.4 MT_ContHomeRun – Continue a movement routine
Continued

Other information

Information about	See
HomeRun strategy	*MT_HomeRun – HomeRun Strategy on page 357*
Backwards processing of a movement routine	*MT_MoveRoutine – Execute a movement routine at HomeRun on page 389*
Programming the HomeRun strategy	*Strategy for automatic movement into the home position on page 134*

15.2.5 MT_CSSDeactMoveL – Linear movement and cartesian softservo disabling

Usage

CSSDeactMoveL is used to move the tool center point (TCP) linearly to a given stop point destination while shifting the robot back to stiff control. Any force offset applied by CSSForceOffsetAct is also deactivated.

When the robot reaches the destination position, the numeric value that is passed is saved as the current position.

This instruction can only be used in the Main task T_ROB1 or in motion tasks in the case of a MultiMove system.

The instruction basically corresponds to a CSSDeactMoveL with some additions.

Basic example

Position No. 11:

```
MT_CSSDeactMoveL 11, p11, v200, tGripper;
```

The TCP of the tGripper tool moves linearly at speed v200 to a stop point, which is position p11. When the position is reached, "11" is stored as the current position.

Intermediate position of movement from 10 to 11:

```
MT_CSSDeactMoveL 101101,* ,vmax, tGripper;
```

The TCP of the tGripper tool moves linearly to the position programmed in the instruction (marked with an *) and then saves “111001” as current position.

Arguments

```
MT_CSSDeactMoveL ActPos ToPoint speed Tool [ \Wobj]
```

`ActPos`	Data type: `dnum` Contains the position number of the position to be moved to.
`ToPoint`	Data type: `robtarget` The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).
`speed`	Data type: `speeddata` The speed programmed for the movement. The speed data define the velocity of the TCP, of the tool reorientation and of external axes. The TCP speed allowed for CSSDeactMoveL is limited to 500 mm/s
`Tool`	Data type: `tooldata` The tool in use when the robot moves. The tool centre point is moved to the specified destination point.
`[\Wobj]`	Data type: `wobjdata` The work object (tool coordinate system) to which the robot position in the instruction is related. This argument can be omitted. In this case the position relates to the world coordinate system.

Continues on next page

15.2.5 MT_CSSDeactMoveL – Linear movement and cartesian softservo disabling
Continued

Program execution

After reaching the programmed position, the transferred position number is saved as current robot position.

If the robot is moved to the home position using the HomeRun, the position number-dependent movement routine is called up by the "MT_HomeRun" routine, and the first position to be moved to (start position) is searched for, i.e. no movement takes place until the saved robot position is identical with the position number in the movement instruction.

Since the “linear” or “axis-related” movement mode is saved in every movement command, it is assured that the movement to the start position is performed with the same movement mode as was used previously.

The stiffness of the robot is gradually increased during the movement. When the stop point is reached the robot will be stiff.

Limitations

There are the following limitations:

- Deactivation can only be done in stop points.
- The programmed TCP speed is not allowed to exceed 500 mm/s.
- Only allowed in a motion controlling task.

Syntax

```
MT_CSSDeactMoveL
[ActPos ':='] < expression (IN) of dnum> ','
[ToPoint ':='] < expression (IN) of robtarget > ','
[Speed ':=' ] < expression (IN) of speeddata > ','
[Tool ':='] < persistent (PERS) of tooldata >
['\'WObj ':=' < persistent (PERS) of wobjdata > ] ';'
```

Other information

Information about	See
`CSSDeactMoveL` – Linear robot movement.	*Application manual SoftMove* listed in the section *References on page 11*.

15.2.6 MT_EndOfCycleAck – Acknowledge the request "Halt after end of cycle"

Usage

MT_EndOfCycleAck is used to acknowledge a "Halt after end of cycle" request. Here the request can be triggered by a digital input signal or even at the RWMT user interface.

Basic example

```
PROC Production()
!If halt after end of cycle is requested
IF MT_ EndOfCycleReq() THEN
!empty the cell
RunOutCycle;
!Notification, that end of cycle has been reached
MT_EndOfCycleAck;
ELSE
!Execute normal production cycle
NormalCycle;
ENDIF
…
…
!If halt after end of cycle has been already confirmed
!in the production cycle => move to home position
IF MT_EndOfCycleOk() MoveTo 999;
ENDPROC
```

At the start of the production cycle, there is a query asking if "Halt after end of cycle" has been requested (MT_EndOfCycleReq). If this is the case, then a run-out cycle is executed so that for example, remaining parts in the cell can be outfeeded

Now the request for “Halt after end of cycle“ is confirmed through MT_EndOfCycleAck. When now leaving the production routine and returning to the RWMT engine the program will be finished.

In the further execution of the production routine, it will be checked by means of the function MT_EndOfCycleOk, if a request for “Halt after end of cycle” has been already confirmed. If this is the case, the robot moves to home position

Program execution

If, after the "Halt after end of cycle" has been requested by the application program, this request is acknowledged, then the program run will end, leaving the application programs and returning to the RWMT Engine. A fresh program start will then start from the first program instruction (program start "from main").

If the “direct halt after end of cycle” has been enabled in the system parameters (see the chapter *MT API Commands on page 154*), parameter `direct_stop_after_cycle` then the acknowledgement can be done directely without a previous request.

Syntax

```
MT_EndOfCycleAck ´;´
```

Continues on next page

15.2.6 MT_EndOfCycleAck – Acknowledge the request "Halt after end of cycle"
Continued

More information

Information about	See
Query if the request "Halt after end of cycle" is present	*MT_EndOfCycleReq – Recognizing the request "Halt after end of cycle" on page 459*
Query if the "Halt after end of cycle" request has been acknowledged already	*MT_EndOfCycleOk – Check if "Halt after end of cycle" was acknowledged on page 457*

15.2.7 MT_Execute – Execution of the RWMT Engine

Usage

MT_Execute executes the engine of the RobotWare Machine Tending. The engine is responsible for executing the actions that have been requested by the system operator (for example, production cycles and service routines).

MT_Execute must be called in the start up routine main() of every movement task. The routine main() should not contain any other instructions, functions or conditions apart from this call, and a subsequent stop instruction. Otherwise it cannot be guaranteed that RWMT will work correctely.

Basic examples

```
PROC main()
!Aufruf der RWMT engine
MT_Execute;
Stop;
ENDPROC
```

The RWMT Engine will be called. The operator can now call production cycles or service routines remotely or by means of the graphical user interface.

If the program pointer leaves the engine, then the Stop command will stop the program run.

Program execution

If the RWMT Engine is called with the help of the MT_Execute instruction, the program pointer will remain within the engine until a Halt after end of cycle or the home position is requested.

Syntax

```
MT_Execute ´;´
```

15.2.8 MT_Exit – Program processing complete

Usage

MT_EXIT is used to save the current robot position and cancel program execution. The restart of the program is then inhibited, which means that the program can only be restarted from the first instruction in the “Main” routine (unless the program pointer has been moved manually).

In order to ensure that Homepos-Running works correctly, the instruction MT_EXIT has to be used instead of EXIT if a serious errors occur or to permanently cancel program execution.

Basic example

```
ErrWrite "Fatal error","Illegal state";
MT_EXIT;
```

Program execution is stopped after the current robot position has been saved and cannot be restarted from this position.

Syntax

```
MT_Exit ';'
```

Other information

Information about	See
`EXIT` - Stop program execution	*Technical Reference Manual – Instructions, Functions and Data Types* listed in the section *References on page 11*.
`MT_ExitCycle` – Abort current cycle and start next cycle.	*MT_ExitCycle – Abort current cycle and start next cycle on page 321*

15.2.9 MT_ExitCycle – Abort current cycle and start next cycle

Usage

MT_ExitCycle is used to save the current robot position, cancel the current cycle and set the program pointer (PP) back to the first instruction of the "main" routine.

If “continuous” execution mode has been selected, program execution is continued with the next cycle.

If “cyclic” execution mode has been selected, program execution stops at the first instruction of the "main" routine.

Basic example

```
VAR intnum irHomeRun;
PROC MT_AfterHomeRun()
IDelete irHomeRun;
CONNECT irHomeRun WITH T_HomeRun;
ISignalDI\Single,diIRBgoHome,high,irHomeRun;
ENDPROC
TRAP T_HomeRun
IDelete irHomePos;
StopMove;
ClearPath;
MT_ExitCycle;
ENDTRAP
```

After reaching the home position, an interrupt is connected to the diIRBgoHome input. As soon as this input switches to high , interrupt routine T_HomeRun is called up. This stops robot movement, saves the current robot position and then continues program execution in the "main" routine.

Program execution

Execution of MT_ExitCycle leads to the following result in the current task:

- Current robot movement is stopped
- The robot position is saved through `MT_HomeRunSavePos`
- All robot movement paths that have not yet been executed are deleted in all planes (both in the normal plane and the StorePath-plane)
- All instructions that have started but not yet ended are interrupted in all execution planes (both in the normal plane and the Interrupt plane)
- The program pointer is set to the first instruction of the main routine

Program execution continues with execution of the next cycle

All other modal settings in the program and the system are not influenced by MT_ExitCycle :

The current value of variables or persistents.

Any movement settings such as the StorePath-RestoPath sequence, world zones, , and so on.

Open files, folders , and so on.

Continues on next page

15.2.9 MT_ExitCycle – Abort current cycle and start next cycle
Continued

Defined interrupts , and so on.

> **Note**
>
> If the entry routine is defined with "Move PP to Routine ..." or "Call Routine ..." when using MT_ExitCycle in routine calls, MT_ExitCycle interrupts the current cycle and moves the program pointer back to the first instruction of the entry routine (instead of the "main" routine, as specified above).

Arguments

```
MT_ExitCycle [ \GoHome ]
```

`[\GoHome]` Data type: `switch`

This switch is used to stop program execution and move the robot to the home position without an explicit request (dialogue or digital signal).

Syntax

```
MT_ExitCycle'
[ '\' GoHome ]';'
```

Other information

Information about	See
`ExitCycle` - Abort current cycle and start next cycle	*Technical Reference Manual – Instructions, Functions and Data Types* listed in the section *References on page 11*.
`MT_Exit` - Program processing complete	*MT_Exit – Program processing complete on page 320*

15.2.10 MT_GetUserProgNo – User defined program execution

Usage

MT_GetUserProgNo is used to to fit the customer needs of how to transfer a program number, if the RWMT mechanism can not be modified in a satisfying way.

Provisions for the procedure should be made by the application programmer and programmed in detail if necessary.

RWMT is delivered and installed with a template MT_MAIN.mod. This module represents a template for the application program and also contains, among other routines, the procedure MT_GetUserProgNo.

Basic examples

Example 1:

In the following example MT_GetUserProgNo assigns a constant program number 7 so that RWMT can look up a partdata declaration with this program number and can call the related production routine.

```
PROC MT_GetUserProgNo(
INOUT dnum ProgNo,
INOUT string Routine)
!
!Program number 7 assigned
ProgNo:=7;
!
ENDPROC
```

Example 2:

In this case MT_GetUserProgNo is used for an individual program number transfer through a serial port.

```
PROC MT_GetUserProgNo(
INOUT dnum ProgNo,
INOUT string Routine)
!
!Program number assigned by serial interface
ProgNo:=GetProgNoFromRS232();
!
ENDPROC
```

In this case, RWMT tries to call a production routine by ist name, if the argument Routine has been passed.

In this case, all RWMT mechanisms for reading a program number as well as for evaluating the check codes, are bypassed.

Example 3:

The routine UserProduction is called constantly.

```
PROC MT_GetUserProgNo(
INOUT dnum ProgNo,
INOUT string Routine)
!
!Production routine name assigned
```

Continues on next page

15.2.10 MT_GetUserProgNo – User defined program execution
Continued

```
    Routine:="UserProduction";
    !
    ENDPROC
```

Example 4:

It is also possible to pass a program number or a routine name, depending on the conditions, as shown below:

```
PROC MT_GetUserProgNo(
INOUT dnum ProgNo,
INOUT string Routine)
!
VAR dnum dnProgNo
!user defined hand shake
!to read the program number
IF diProgNoReady=high THEN
dnProgNo=giProgNo;
Set doProgNoAck;
...
!select the routine name which shall
!be executed
TEST dnProgNo
CASE 1,2,3,4,5:
Routine:="Production_T"+Valtostr(dnProgNo)
CASE 100,101:
!call service routines (menudata) or standard
!partdata
ProgNo:=dnProgNo;
ENDTEST
ENDIF
!
ENDPROC
```

Program run

The procedure MT_GetUserProgNo will be called by RWMT automatically if there is no program pre-selection of part type at the RWMT user interface.

The procedure MT_GetUserProgNo will only be used if this has been released in the process configuration.

Arguments

```
MT_GetUserProgNo ProgNo Routine
```

`ProgNo`	**Data type:** `dnum` The program number of the part type that is to be selected. This program number must appear in exactly one declaration of the type partdata. A program number may be specified only if the argument Routine has not been passed respectively is an empty string. When using the argument ProgNo, the evaluation of the tool code and further check codes is done as usual.

Continues on next page

`Routine`	Data type: `string` The production routine that is to be called. A routine may be specified only if the argument ProgNo has not been passed or if the value is <=0. When using the argument Routine, no evaluation of the tool code and further check codes is done.

Syntax

```
MT_GetUserProgNo
[ ProgNo ':=' < expression (INOUT) of dnum > ]
[ Routine':=' < expression (INOUT) of string > ]
';'
```

More information

Information about	See
Part types	*partdata – Part data on page 284*

15.2.11 MT_GripCheck – Check position of the control element of the gripper

Usage

MT_GripCheck is used to wait until all the control elements of the gripper have attained the desired position. Up to 6 control elements can be queried simultaneously.

Basic examples

```
MT_GripCheck gsClose,gdGRP1_Y2;
```

The system waits till the control element Y2 of the gripper 1 is closed.

```
MT_GripCheck gsOpen,gdGRP1_Y2;
```

The system waits till the control element Y2 of the gripper 1 is open.

```
MT_GripCheck gsClose,gdGRP1_Y1\Grp2:=gdGRP1_Y2;
```

The system waits till the control elements Y1 and Y2 of the gripper 1 are closed.

Arguments

```
MT_GripCheck [\CheckOpen] | [\CheckClose] Position Grp1 [\Grp2]
[\Grp3] [\Grp4] [\Grp5] [\Grp6] [\ErrorNo] [\Fault]
```

[\CheckOpen]	Data type: switch Checks the status of the "opened"-response only
[\CheckClose]	Data type: switch Checks the status of the "closed"-response only
Position	Data type: grppos Desired connection status (for example, Control element or control elements opened or closed)
Grp1	Data type: grpdata Gripper data of the first control element that is to be monitored.
[\Grp2]	Data type: grpdata Gripper data of the second control element that is to be monitored.
[\Grp3]	Data type: grpdata Gripper data of the third control element that is to be monitored.
[\Grp4]	Data type: grpdata Gripper data of the fourth control element that is to be monitored.
[\Grp5]	Data type: grpdata Gripper data of the fifth control element that is to be monitored.
[\Grp6]	Data type: grpdata Gripper data of the sixth control element that is to be monitored.
\Error No	Data type: num Combined error domains and error number as positive integer, which can be used for display in the event of errors. The last four digits represent the error number, the digits preceding this represent the error domain.
[\Fault]	Data type: switch If the switch fault is set, a gripper related message will appear as a fault message, otherwise it will appear as a warning message.

Continues on next page

Program execution

The program will be continued only if all the sensors of the control elements have attained the corresponding open or closed positions.

Only those signals that contain a valid signal name will be considered.

If the query is for 'closed', all the "Closed"-signals must be set to "high" and all the "Open"-signals must be set to "low".

If the query is for 'opened', all the "Closed"-signals must be set to "low" and all the "Open"-signals must be set to "high".

By using the `\CheckClose` or `\CheckOpen` switch, it is possible to omit the query for the opposite "opened" or "closed" case.

If a sensor fails to reach the required state within the defined waiting period, then an error message is output.

Syntax

```
MT_GripCheck
[´\CheckOpen ]|[´\CheckClose ] ’,’
[Position] ´:=´ < expression (IN) of grppos> ]
[Grp1] ´:=´ < expression (IN) of grppdata> ]
[´\Grp2 ´:=´ < expression (IN) of grpdata> ]
[´\Grp3 ´:=´ < expression (IN) of grpdata> ]
[´\Grp4 ´:=´ < expression (IN) of grpdata> ]
[´\Grp5 ´:=´ < expression (IN) of grpdata> ]
[´\Grp6 ´:=´ < expression (IN) of grpdata> ]
[´\ErrorNo ´:=´ < expression (IN) of num> ]
[´\´Fault´]
´;´
```

More information

Information about	See
Gripper data	*grpdata – Configuration of a control element of the gripper on page 257*
Gripper position	*grppos – Gripper position on page 262*

15.2.12 MT_GripCheckType – Check pos. of the control element of the gripper

Usage

`MT_GripCheckType` is used to wait until all the control elements of the gripper have attained the desired position. Up to 6 control elements can be queried simultaneously.

`MT_GripCheckType` provides mainly the same functionality as `MT_GripCheck` but considers part type specific type numbers and type prefixes as follows:

- There might be different grippers for each part type in the production cell. The grippers might work differently, thus each gripper will need its own grpdata declarations.
- Instead of assigning the grpdata directly as this is done with `MT_GripCheck`, a string is provided to `MT_GripCheckType` which represents the name of the grpdata but without part type number and part type prefix.
- `MT_GripCheckType` will internally complete the grpdata name, depending on the current settings for the type prefix and the type number. Then the instruction will execute the appropriate type-depending grpdata declaration.

Basic examples

Assuming, the current part type number is 6 and the standard part type prefix is “T”

```
MT_GripCheckType gsClose,"gdGRP_Y2";
```

The system waits till the control element Y2 of the gripper is closed (grpdata gdGRP_Y2_T6)

```
MT_ GripCheckType gsOpen,"gdGRP_Y2"\Prefix:= "P";
```

The system waits till the control element Y2 of the gripper is open (grpdata gdGRP_Y2_P6).

```
MT_ GripCheckType gsClose,"gdGRP_Y1"\Grp2:="gdGRP_Y2";
```

The system waits till the control elements Y1 and Y2 of the gripper are closed (grpdata gdGRP_Y1_P6 and gdGRP_Y2_P6).

Arguments

```
MT_GripCheckType [\CheckOpen] | [\CheckClose] Position Grp1
[\Grp2] [\Grp3] [\Grp4] [\Grp5] [\Grp6] [\Prefix] [\ErrorNo]
[\Fault]
```

`[\CheckOpen]`	Data type: `switch` Checks the status of the "opened"-response only
`[\CheckClose]`	Data type: `switch` Checks the status of the "closed"-response only
`Position`	Data type: `grppos` Desired connection status (for example, Control element or control elements opened or closed)

Continues on next page

`Grp1`	Data type: `string` Gripper data name of the first control element that is to be actuated, without part type prefix and without part type number.
`[\Grp2]`	Data type: `string` Gripper data name of the second control element must that is to be actuated, without part type prefix and without part type number.
`[\Grp3]`	Data type: `string` Gripper data name of the third control element that is to be actuated, without part type prefix and without part type number.
`[\Grp4]`	Data type: `string` Gripper data name of the fourth control element that is to be actuated, without part type prefix and without part type number.
`[\Grp5]`	Data type: `string` Gripper data name of the fifth control element that is to be actuated, without part type prefix and without part type number.
`[\Grp6]`	Data type: `string` Gripper data name of the sixth control element that is to be actuated, without part type prefix and without part type number.
`[\Prefix]`	Data type: `string` Assigns another part type prefix apart from the default prefix.
`[\ErrorNo]`	Data type: `num` Combined error domains and error number as positive integer, which can be used for display in the event of errors. The last four digits represent the error number, the digits preceding this represent the error domain.
`[\Fault]`	Data type: `switch` If the switch fault is set, a gripper related message will appear as a fault message, otherwise it will appear as a warning message.

Program execution

The program will be continued only if all the sensors of the control elements have attained the corresponding open or closed positions.

Only those signals that contain a valid signal name will be considered.

If the query is for 'closed', all the "Closed"-signals must be set to "high" and all the "Open"-signals must be set to "low".

If the query is for 'opened', all the "Closed"-signals must be set to "low" and all the "Open"-signals must be set to "high".

By using the `\CheckClose` or `\CheckOpen` switch, it is possible to omit the query for the opposite "opened" or "closed" case.

If a sensor fails to reach the required state within the defined waiting period, then an error message is output.

Syntax

```
MT_GripCheckType
[´\CheckOpen ]|[´\CheckClose ] ’,’
[Position] ´:=´ < expression (IN) of grppos> ]
[Grp1] ´:=´ < expression (IN) of string> ]
[´\Grp2 ´:=´ < expression (IN) of string > ]
```

Continues on next page

15.2.12 MT_GripCheckType – Check pos. of the control element of the gripper
Continued

```
[´\Grp3 ´:=´ < expression (IN) of string > ]
[´\Grp4 ´:=´ < expression (IN) of string > ]
[´\Grp5 ´:=´ < expression (IN) of string > ]
[´\Grp6 ´:=´ < expression (IN) of string > ]
[´\Prefix ´:=´ < expression (IN) of string > ]
[´\ErrorNo ´:=´ < expression (IN) of num> ]
[´\´Fault´]
´;´
```

More information

Information about	See
Gripper data	*grpdata – Configuration of a control element of the gripper on page 257*
Gripper position	*grppos – Gripper position on page 262*

15.2.13 MT_GripJ – Robot axis movement with gripper settings

Usage

MT_GripJ is used similary to MT_MoveJ to move the robot quickly from one point to another, if this movement does not have to be in a straight line. When the robot reaches the destination position, the numeric value that is passed is saved as the current position.

The robot and the external axes move to the destination position along a non-linear path. All axes reach the destination position simultaneously.

In addition, a gripper action in the target position is executed.

This instruction can only be used in the Main task T_ROB1 or in motion tasks in the case of a MultiMove system.

The instruction basically corresponds to a MoveJ with some additions.

Basic example

Gripping pre-position no. 10:

```
MT_GripJ 10, p10, v1000, z30, tGripper, gsOpen, gdGRP1_Y2;
```

The TCP of the tGripper tool moves to position p10 in an axle-related way at speed v1000 and zone data z30. When the position is reached, "11" is stored as the current position.

The control element Y2 of the gripper 1 will be opened when having reached position p10.

Arguments

```
MT_GripJ
[\Conc] ActPos ToPoint speed [\T] zone [\Inpos]
Tool [ \Wobj] Position Grp1 [\Grp2] [\Grp3] [\Grp4] [\Grp5] [\Grp6]
     [\PartLoad] | [\SetLoad] | [\ResetLoad] [\NoCheck] |
     [\CheckOpen] | [\CheckClose] [\Prefix] [\ErrorNo] [\Fault]
```

`[\Conc]`
Concurrent

Data type: `switch`

The following instructions are executed whilst the robot is in motion.

Further information can be obtained from the MoveJ instruction.

`ActPos`

Data type: `dnum`

Contains the position number of the position to be moved to.

`ToPoint`

Data type: `robtarget`

The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

`speed`

Data type: `speeddata`

The speed programmed for the movement. The speed data define the velocity of the TCP, of the tool reorientation and of external axes.

`[\T](Time)`

Data type: `num`

This argument is used to specify the time in seconds during which a movement of the manipulator and of the external axes should be executed. This value is then substituted for the corresponding speed data.

Continues on next page

15.2.13 MT_GripJ – Robot axis movement with gripper settings
Continued

`Zone`	Data type: `zonedata` Zone data for the movement. Zone data describe the distance in which the axes must stand from the destination point before the next instruction is executed.
`[\Inpos]`	Data type: `stoppointdata` This argument is used to specify the convergence criteria for the position of the robot's TCP in the stop point. The stop point data substitutes the zone specified in the Zone parameter.
`Tool`	Data type: `tooldata` The tool in use when the robot moves. The tool centre point is moved to the specified destination point.
`[\Wobj]`	Data type: `wobjdata` The work object (tool coordinate system) to which the robot position in the instruction is related. This argument can be omitted. In this case the position relates to the world coordinate system.
`Position`	Data type: `grppos` Desired position (for example, opening or closing the control element or the control elements).
`Grp1`	Data type: `grpdata` Gripper data of the first control element that is to be actuated.
`[\Grp2]`	Data type: `grpdata` Gripper data of the second control element must that is to be actuated.
`[\Grp3]`	Data type: `grpdata` Gripper data of the third control element that is to be actuated.
`[\Grp4]`	Data type: `grpdata` Gripper data of the fourth control element that is to be actuated.
`[\Grp5]`	Data type: `grpdata` Gripper data of the fifth control element that is to be actuated.
`[\Grp6]`	Data type: `grpdata` Gripper data of the sixth control element that is to be actuated.
`[\PartLoad]`	Data type: `loaddata` Load data that will be activated on closing or opening the gripper.
`[\SetLoad]`	Data type: `switch` Sets the load of the currently selected part (see *partdata – Part data on page 284*).
`[\ResetLoad]`	Data type: `switch` Resets the load to load0.
`[\NoCheck]`	Data type: `switch` If this switch is used, the system will not wait till the sensors have reached the required position.
`[\CheckOpen]`	Data type: `switch` If this switch is used, the system will wait till the "open" sensors have reached the required position. No check for "closed" sensors will be performed.
`[\CheckClose]`	Data type: `switch` If this switch is used, the system will wait till the "closed" sensors have reached the required position. No check for "open" sensors will be performed.

Continues on next page

[\Prefix]	Data type: string A type prefix, which is different from the prefix which has been set in the system parameters (See the chapters *MT Part Settings on page 162* and *Use of type-related movement routines on page 137*).
[\ErrorNo]	Data type: num Combined error domains and error number as positive integer, which can be used for display in the event of errors. The last four digits represent the error number, the digits preceding this represent the error domain.
[\Fault]	Data type: switch If the switch fault is set, a gripper related message will appear as a fault message, otherwise it will appear as a warning message.

Program execution

After reaching the programmed position, the transferred position number is saved as current robot position.

If the robot is moved to the home position using HomeRun, the position number-dependent movement routine is called up by the "MT_HomeRun" routine, and the first position to be moved to (start position) is searched for, that is, no movement takes place until the saved robot position is identical with the position number in the movement instruction.

Since the “linear” or “axis-related” movement mode is saved whenever a movement command is executed, it is ensured that the movement to the start position is performed with the same movement mode that was used previously.

In the target position, the specified valves of the gripper will be actuated. This is followed by a waiting time which is defined by the longest one of all available actuators.

If a sensor fails to reach the required state within the defined waiting period, then an error message will be output on the programming device and a corresponding error code will be sent for example,to an external PLC. To be able to send an error code, this must be process configuration first (See the chapter *MT API Commands on page 154*).

Tip

In the "Ghost mode", the "NoGhostSet" flag is evaluated in every control element declaration. If the flag has been set to "TRUE" then the sensors will not be checked. If the flag has been set to "FALSE" then the sensors will be checked even in the ghost mode.

Tip

In the "Ghost mode" the "NoGhostCheck" flag is evaluated in every control element declaration. If the flag has been set to "TRUE", the valve will not be actuated. If the flag has been set to "FALSE", then the actuation is done even in the ghost mode.

Continues on next page

15.2.13 MT_GripJ – Robot axis movement with gripper settings
Continued

Restrictions

The optional parameters \PartLoad, \SetLoad and \ResetLoad cannot be used in common.

The optional parameters \NoCheck, \CheckOpen and \CheckClose cannot be used in common as well.

Syntax

```
MT_GripJ
['\'Conc ',']
[ActPos ':='] < expression (IN) of dnum> ','
[ToPoint ':='] < expression (IN) of robtarget > ','
[speed ':='] < expression (IN) of speeddata >
['\'T ':=' < expression (IN) of num > ',']
[zone ':='] < expression (IN) of zonedata >
['\'Inpos':='< expression (IN) of stoppointdata > ',']
[Tool ':='] < persistent (PERS) of tooldata >
['\'WObj ':=' < persistent (PERS) of wobjdata > ]
[Position] ´:=´ < expression (IN) of grppos> ]
[Grp1 ´:=´ < expression (IN) of grpdata> ]
[´\´Grp2 ´:=´ < expression (IN) of grpdata> ]
[´\´Grp3 ´:=´ < expression (IN) of grpdata> ]
[´\´Grp4 ´:=´ < expression (IN) of grpdata> ]
[´\´Grp5 ´:=´ < expression (IN) of grpdata> ]
[´\´Grp6 ´:=´ < expression (IN) of grpdata> ]
[´\´PartLoad ´:=´ < expression (IN) of loaddata>]
| [´\´SetLoad´]
| [´\´ResetLoad´]
[´\´NoCheck´]
| [´\´CheckOpen´]
| [´\´CheckClosed´]
[´\´Prefix ´:=´ < expression (IN) of string> ]
[´\´ErrorNo]´:=´ < expression (IN) of num> ]
[´\´Fault] ';'
```

Other information

Information about	See
`MoveJ` Robot axis movement	*Technical Reference Manual – Instructions, Functions and Data Types* listed in the section *References on page 11*.
`MT_MoveJ` Robot axis movement	*MT_MoveJ – Robot axis movement on page 360*
`MT_GripSet` Controlling the gripper	*MT_GripSet – Controlling the gripper on page 349*

15.2.14 MT_GripL – Robot linear movement with gripper settings

Usage

MT_GripL is used similary to MT_MoveL to move the robot quickly from one point to another, if this movement has to be in a straight line. When the robot reaches the destination position, the numeric value that is passed is saved as the current position.

The robot and the external axes move to the destination position along a linear path. All axes reach the destination position simultaneously.

In addition, a gripper action in the target position is executed.

This instruction can only be used in the Main task T_ROB1 or in motion tasks in the case of a MultiMove system.

The instruction basically corresponds to a MoveL with some additions.

Basic example

Gripping position no. 11:

```
MT_GripL 11, p11, v1000, fine, tGripper, gsClose, gdGRP1_Y2;
```

The TCP of the tGripper tool moves to position p11 in an linear way at speed v1000 and zone data fine. When the position is reached, "11" is stored as the current position.

The control element Y2 of the gripper 1 will be closed when having reached position p11.

Arguments

```
MT_GripJ
[\Conc] ActPos ToPoint speed [\T] zone [\Inpos]
Tool [ \Wobj] Position Grp1 [\Grp2] [\Grp3] [\Grp4] [\Grp5] [\Grp6]
     [\PartLoad] | [\SetLoad] | [\ResetLoad] [\NoCheck] |
     [\CheckOpen] | [\CheckClose] [\Prefix] [\ErrorNo] [\Fault]
```

`[\Conc]`
`Concurrent`

Data type: `switch`

The following instructions are executed whilst the robot is in motion.

Further information can be obtained from the MoveJ instruction.

`ActPos`

Data type: `dnum`

Contains the position number of the position to be moved to.

`ToPoint`

Data type: `robtarget`

The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

`speed`

Data type: `speeddata`

The speed programmed for the movement. The speed data define the velocity of the TCP, of the tool reorientation and of external axes.

`[\T](Time)`

Data type: `num`

This argument is used to specify the time in seconds during which a movement of the manipulator and of the external axes should be executed. This value is then substituted for the corresponding speed data.

Continues on next page

15.2.14 MT_GripL – Robot linear movement with gripper settings
Continued

`Zone`	Data type: `zonedata` Zone data for the movement. Zone data describe the distance in which the axes must stand from the destination point before the next instruction is executed.
`[\Inpos]`	Data type: `stoppointdata` This argument is used to specify the convergence criteria for the position of the robot's TCP in the stop point. The stop point data substitutes the zone specified in the Zone parameter.
`Tool`	Data type: `tooldata` The tool in use when the robot moves. The tool centre point is moved to the specified destination point.
`[\Wobj]`	Data type: `wobjdata` The work object (tool coordinate system) to which the robot position in the instruction is related. This argument can be omitted. In this case the position relates to the world coordinate system.
`Position`	Data type: `grppos` Desired position (for example, opening or closing the control element or the control elements)
`Grp1`	Data type: `grpdata` Gripper data of the first control element that is to be actuated
`[\Grp2]`	Data type: `grpdata` Gripper data of the second control element must that is to be actuated
`[\Grp3]`	Data type: `grpdata` Gripper data of the third control element that is to be actuated
`[\Grp4]`	Data type: `grpdata` Gripper data of the fourth control element that is to be actuated
`[\Grp5]`	Data type: `grpdata` Gripper data of the fifth control element that is to be actuated
`[\Grp6]`	Data type: `grpdata` Gripper data of the sixth control element that is to be actuated
`[\PartLoad]`	Data type: `loaddata` Load data that will be activated on closing or opening the gripper.
`[\SetLoad]`	Data type: `switch` Sets the load of the currently selected part (see *partdata – Part data on page 284*).
`[\ResetLoad]`	Data type: `switch` Resets the load to load0.
`[\NoCheck]`	Data type: `switch` If this switch is used, the system will not wait till the sensors have reached the required position.
`[\CheckOpen]`	Data type: `switch` If this switch is used, the system will wait till the "open" sensors have reached the required position. No check for "closed" sensors will be performed.
`[\CheckClose]`	Data type: `switch` If this switch is used, the system will wait till the "closed" sensors have reached the required position. No check for "open" sensors will be performed.

Continues on next page

[\Prefix]	Data type: `string` A type prefix, which is different from the prefix which has been set in the system parameters (See the chapters *MT Part Settings on page 162* and *Use of type-related movement routines on page 137*).
[\ErrorNo]	Data type: `num` Combined error domains and error number as positive integer, which can be used for display in the event of errors. The last four digits represent the error number, the digits preceding this represent the error domain.
[\Fault]	Data type: `switch` If the switch fault is set, a gripper related message will appear as a fault message, otherwise it will appear as a warning message.

Program execution

After reaching the programmed position, the transferred position number is saved as current robot position.

If the robot is moved to the home position using the HomeRun, the position number-dependent movement routine is called up by the "MT_HomeRun" routine, and the first position to be moved to (start position) is searched for, i.e. no movement takes place until the saved robot position is identical with the position number in the movement instruction.

Since the “linear” or “axis-related” movement mode is saved whenever a movement command is executed, it is ensured that the movement to the start position is performed with the same movement mode that was used previously.

In the target position, the specified valves of the gripper will be actuated. This is followed by a waiting time which is defined by the longest one of all available actuators.

If a sensor fails to reach the required state within the defined waiting period, then an error message will be output on the programming device and a corresponding error code will be sent for example,to an external PLC. To be able to send an error code, this must be configured in the process configuration first (See the chapter *MT API Commands on page 154*).

Tip

In the "Ghost mode", the "NoGhostSet" flag is evaluated in every control element declaration. If the flag has been set to "TRUE" then the sensors will not be checked. If the flag has been set to "FALSE" then the sensors will be checked even in the ghost mode.

Tip

In the "Ghost mode" the "NoGhostCheck" flag is evaluated in every control element declaration. If the flag has been set to "TRUE", the valve will not be actuated. If the flag has been set to "FALSE", then the actuation is done even in the ghost mode

Continues on next page

15.2.14 MT_GripL – Robot linear movement with gripper settings
Continued

Restrictions

The optional parameters \PartLoad, \SetLoad and \ResetLoad cannot be used in common.

The optional parameters \NoCheck, \CheckOpen and \CheckClose cannot be used in common as well.

Syntax

```
MT_GripL
['\'Conc ',']
[ActPos ':='] < expression (IN) of dnum> ','
[ToPoint ':='] < expression (IN) of robtarget > ','
[speed ':='] < expression (IN) of speeddata >
['\'T ':=' < expression (IN) of num > ',']
[zone ':='] < expression (IN) of zonedata >
['\'Inpos':='< expression (IN) of stoppointdata > ',']
[Tool ':='] < persistent (PERS) of tooldata >
['\'WObj ':=' < persistent (PERS) of wobjdata > ]
[Position] ´:=´ < expression (IN) of grppos> ]
[Grp1 ´:=´ < expression (IN) of grpdata> ]
[´\´Grp2 ´:=´ < expression (IN) of grpdata> ]
[´\´Grp3 ´:=´ < expression (IN) of grpdata> ]
[´\´Grp4 ´:=´ < expression (IN) of grpdata> ]
[´\´Grp5 ´:=´ < expression (IN) of grpdata> ]
[´\´Grp6 ´:=´ < expression (IN) of grpdata> ]
[´\´PartLoad ´:=´ < expression (IN) of loaddata>]
| [´\´SetLoad´]
| [´\´ResetLoad´]
[´\´NoCheck´]
| [´\´CheckOpen´]
| [´\´CheckClosed´]
[´\´Prefix ´:=´ < expression (IN) of string> ]
[´\´ErrorNo]´:=´ < expression (IN) of num> ]
[´\´Fault] ';'
```

Other information

Information about	See
`MoveL` Linear robot movement	*Technical Reference Manual – Instructions, Functions and Data Types* listed in the section *References on page 11*.
`MT_MoveL` Linear robot movement	*MT_MoveL – Linear robot movement. on page 374*
`MT_GripSet` Controlling the gripper	*MT_GripSet – Controlling the gripper on page 349*

15.2.15 MT_GripSeqJ – Robot axis movement with gripper sequence

Usage

MT_GripJ is used similary to MT_MoveJ to move the robot quickly from one point to another, if this movement does not have to be in a straight line. When the robot reaches the destination position, the numeric value that is passed is saved as the current position.

The robot and the external axes move to the destination position along a non-linear path. All axes reach the destination position simultaneously.

In addition, a gripper sequence is executed in the target position.

This instruction can only be used in the Main task T_ROB1 or in motion tasks in the case of a MultiMove system.

The instruction basically corresponds to a MoveJ with some additions.

Basic example

```
!gripper data
const grpdata gdY1_T127:=[];
const grpdata gdY2_T127:=[];
const grpdata gdY3_T127:=[];
const grpdata gdY4_T127:=[];

!gripper sequence for opening the gripper
const grpseq gsOpenGripper{3}:=
[[gsClose,"gdY1_T127","","","","",""],
[[gsOpen,"gdY2_T127","gdY3_T127","", "","","",""],
[[gsClose,"gdY4_T127","","", "","",""]];

MT_GripSeqJ 10, p10, v1000, z30, tGripper
\Sequence:= gsOpenGripper;
```

In the sequence declaration gsOpenGripper, the control element Y1 is first closed, then the control elements Y2 and Y3 are opened and then the control element Y4 is closed.

The TCP of the tGripper tool moves to position p10 in an axle-related way at speed v1000 and zone data z30. When the position is reached, "10" is stored as the current position.

When having reached the position, the gripper sequence is called to open the gripper.

Arguments

```
MT_GripSeqJ
[\Conc] ActPos ToPoint speed [\T] zone [\Inpos]
Tool [ \Wobj] [\Sequence] | [SeqName] [\Prefix] [\PartLoad] |
     [SetLoad] | [ResetLoad] [\ErrorNo] [\Fault]
```

[\Conc] Concurrent

Data type: switch

The following instructions are executed whilst the robot is in motion. Further information can be obtained from the MoveJ instruction.

Continues on next page

15.2.15 MT_GripSeqJ – Robot axis movement with gripper sequence
Continued

`ActPos`	Data type: `dnum` Contains the position number of the position to be moved to.
`ToPoint`	Data type: `robtarget` The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).
`speed`	Data type: `speeddata` The speed programmed for the movement. The speed data define the velocity of the TCP, of the tool reorientation and of external axes.
`[\T] (Time)`	Data type: `num` This argument is used to specify the time in seconds during which a movement of the manipulator and of the external axes should be executed. This value is then substituted for the corresponding speed data.
`Zone`	Data type: `zonedata` Zone data for the movement. Zone data describe the distance in which the axes must stand from the destination point before the next instruction is executed.
`[\Inpos]`	Data type: `stoppointdata` This argument is used to specify the convergence criteria for the position of the robot's TCP in the stop point. The stop point data substitutes the zone specified in the Zone parameter.
`Tool`	Data type: `tooldata` The tool in use when the robot moves. The tool centre point is moved to the specified destination point.
`[\Wobj]`	Data type: `wobjdata` The work object (tool coordinate system) to which the robot position in the instruction is related. This argument can be omitted. In this case the position relates to the world coordinate system.
`[\Sequence]`	Data type: `grpseq` Array with the gripper sequence. Only one of the arguments \Sequence or \SeqName should be used.
`[\SeqName]`	Data type: `string` Name of the gripper sequence that is to be executed. Only one of the arguments \Sequence or \SeqName should be used.
`[\Prefix]`	Data type: `string` A type prefix, which is different from the prefix which has been set in the system parameters (See the chapters *MT Part Settings on page 162* and *Use of type-related movement routines on page 137*).
`[\PartLoad]`	Data type: `loaddata` New gripper load after the sequence is executed.
`[\SetLoad]`	Data type: `switch` Sets the load of the currently selected part (see *partdata – Part data on page 284*).
`[\ResetLoad]`	Data type: `switch` Resets the load to load0.
`[\ErrorNo]`	Data type: `num` Combined error domains and error number as positive integer, which can be used for display in the event of errors. The last four digits represent the error number, the digits preceding this represent the error domain.

Continues on next page

[\Fault]	Data type: switch If the switch fault is set, a gripper related message will appear as a fault message, otherwise it will appear as a warning message.

Program execution

After reaching the programmed position, the transferred position number is saved as current robot position.

If the robot is moved to the home position using HomeRun, the position number-dependent movement routine is called up by the "MT_HomeRun" routine, and the first position to be moved to (start position) is searched for, i.e. no movement takes place until the saved robot position is identical with the position number in the movement instruction.

Since the “linear” or “axis-related” movement mode is saved whenever a movement command is executed, it is ensured that the movement to the start position is performed with the same movement mode that was used previously.

Whe having reached the target position, the valves of the control elements that have been passed will be opened or closed one after the other, depending on the selection.

If all the sensors fail to reach the required state within the defined waiting period, then, an error message will be output.

Limitations

When using SeqName the array size for the gripper sequence is limited to 20 elements.

Restrictions

The optional parameters \PartLoad, \SetLoad and \ResetLoad cannot be used in common.

Syntax

```
MT_GripSeqJ
['\'Conc ',']
[ActPos ':='] < expression (IN) of dnum> ','
[ToPoint ':='] < expression (IN) of robtarget > ','
[speed ':='] < expression (IN) of speeddata >
['\'T ':=' < expression (IN) of num > ',']
[zone ':='] < expression (IN) of zonedata >
['\'Inpos':='< expression (IN) of stoppointdata > ',']
[Tool ':='] < persistent (PERS) of tooldata >
['\'WObj ':=' < persistent (PERS) of wobjdata > ]
[´\´Sequence ´:=´ < expression {IN} of grpseq > ] |
[´\´SeqName ´:=´ < expression (IN) of string > ]
[´\´Prefix ´:=´ < expression (IN) of string > ]
[´\´PartLoad ´:=´ < expression (PERS) of loaddata > ] |
[´\´SetLoad] |
[´\´ResetLoad]
[´\´ErrorNo] ´:=´ < expression (IN) of num> ]
[´\´Fault] ';'
```

Continues on next page

15.2.15 MT_GripSeqJ – Robot axis movement with gripper sequence
Continued

Other information

Information about	See
`MoveJ` Robot axis movement	*Technical Reference Manual – Instructions, Functions and Data Types* listed in the section *References on page 11*
`MT_MoveJ` Robot axis movement	*MT_MoveJ – Robot axis movement on page 360*
`MT_GripSequence` Sequential gripper actuation	*MT_GripSequence – Sequential actuation of gripper actuators on page 347*

15.2.16 MT_GripSeqL – Linear robot movement with gripper sequence

Usage

MT_GripL is used similary to MT_MoveJ to move the robot quickly from one point to another, if this movement has to be in a straight line. When the robot reaches the destination position, the numeric value that is passed is saved as the current position.

The robot and the external axes move to the destination position along a linear path. All axes reach the destination position simultaneously.

In addition, a gripper sequence is executed in the target position.

This instruction can only be used in the Main task T_ROB1 or in motion tasks in the case of a MultiMove system.

The instruction basically corresponds to a MoveL with some additions.

Basic example

```
!gripper data
const grpdata gdY1_T127:=[];
const grpdata gdY2_T127:=[];
const grpdata gdY3_T127:=[];
const grpdata gdY4_T127:=[];
!gripper sequence for opening the gripper
const grpseq gsOpenGripper{3}:=
[[gsClose,"gdY1_T127","","","","",""],
[[gsOpen,"gdY2_T127","gdY3_T127","", "","",""],
[[gsClose,"gdY4_T127","","", "","",""]];
MT_GripSeqL 16, p16, v1000, fine, tGripper
\Sequence:= gsOpenGripper;
```

In the sequence declaration gsOpenGripper, the control element Y1 is first closed, then the control elements Y2 and Y3 are opened and then the control element Y4 is closed.

The TCP of the tGripper tool moves to position p16 in an axle-related way at speed v1000 and zone data fine. When the position is reached, "16" is stored as the current position.

When having reached the position, the gripper sequence is called to open the gripper.

Arguments

```
MT_GripSeqL
[\Conc] ActPos ToPoint speed [\T] zone [\Inpos]
Tool [ \Wobj] [\Sequence] | [SeqName] [\Prefix] [\PartLoad] |
     [SetLoad] | [ResetLoad] [\ErrorNo] [\Fault]
```

`[\Conc]`
`Concurrent`

Data type: `switch`
The following instructions are executed whilst the robot is in motion. Further information can be obtained from the MoveJ instruction.

`ActPos`

Data type: `dnum`
Contains the position number of the position to be moved to.

Continues on next page

15.2.16 MT_GripSeqL – Linear robot movement with gripper sequence
Continued

`ToPoint`	Data type: `robtarget` The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).
`speed`	Data type: `speeddata` The speed programmed for the movement. The speed data define the velocity of the TCP, of the tool reorientation and of external axes.
`[\T] (Time)`	Data type: `num` This argument is used to specify the time in seconds during which a movement of the manipulator and of the external axes should be executed. This value is then substituted for the corresponding speed data.
`Zone`	Data type: `zonedata` Zone data for the movement. Zone data describe the distance in which the axes must stand from the destination point before the next instruction is executed.
`[\Inpos]`	Data type: `stoppointdata` This argument is used to specify the convergence criteria for the position of the robot's TCP in the stop point. The stop point data substitutes the zone specified in the Zone parameter.
`Tool`	Data type: `tooldata` The tool in use when the robot moves. The tool centre point is moved to the specified destination point.
`[\Wobj]`	Data type: `wobjdata` The work object (tool coordinate system) to which the robot position in the instruction is related. This argument can be omitted. In this case the position relates to the world coordinate system.
`\Sequence`	Data type: `grpseq` Array with the gripper sequence. Only one of the arguments \Sequence or \SeqName should be used.
`\SeqName`	Data type: `string` Name of the gripper sequence that is to be executed. Only one of the arguments \Sequence or \SeqName should be used.
`[\Prefix]`	Data type: `string` A type prefix, which is different from the prefix which has been set in the system parameters (See the chapters *MT Part Settings on page 162* and *Use of type-related movement routines on page 137*).
`\PartLoad`	Data type: `loaddata` New gripper load after the sequence is executed.
`[\SetLoad]`	Data type: `switch` Sets the load of the currently selected part (see *partdata – Part data on page 284*).
`[\ResetLoad]`	Data type: `switch` Resets the load to load0.
`[\ErrorNo]`	Data type: `num` Combined error domains and error number as positive integer, which can be used for display in the event of errors. The last four digits represent the error number, the digits preceding this represent the error domain.

Continues on next page

[\Fault]	Data type: switch If the switch fault is set, a gripper related message will appear as a fault message, otherwise it will appear as a warning message.

Program execution

After reaching the programmed position, the transferred position number is saved as current robot position.

If the robot is moved to the home position using the HomeRun, the position number-dependent movement routine is called up by the "MT_HomeRun" routine, and the first position to be moved to (start position) is searched for, i.e. no movement takes place until the saved robot position is identical with the position number in the movement instruction.

Since the “linear” or “axis-related” movement mode is saved whenever a movement command is executed, it is ensured that the movement to the start position is performed with the same movement mode that was used previously.

Whe having reached the target position, the valves of the control elements that have been passed will be opened or closed one after the other, depending on the selection.

If all the sensors fail to reach the required state within the defined waiting period, then, an error message will be output.

Limitations

When using SeqName the array size for the gripper sequence is limited to 20 elements.

Restrictions

The optional parameters \PartLoad, \SetLoad and \ResetLoad cannot be used in common.

Syntax

```
MT_GripSeqL
[’\’Conc ’,’]
[ActPos ’:=’] < expression (IN) of dnum> ’,’
[ToPoint ’:=’] < expression (IN) of robtarget > ’,’
[speed ’:=’] < expression (IN) of speeddata >
[’\’T ’:=’ < expression (IN) of num > ’,’]
[zone ’:=’] < expression (IN) of zonedata >
[’\’Inpos’:=’< expression (IN) of stoppointdata > ’,’]
[Tool ’:=’] < persistent (PERS) of tooldata >
[’\’WObj ’:=’ < persistent (PERS) of wobjdata > ]
[´\´Sequence ´:=´ < expression {IN} of grpseq > ] |
[´\´SeqName ´:=´ < expression (IN) of string > ]
[´\´Prefix ´:=´ < expression (IN) of string > ]
[´\´PartLoad ´:=´ < expression (PERS) of loaddata > ] |
[´\´SetLoad] |
[´\´ResetLoad]
[´\´ErrorNo] ´:=´ < expression (IN) of num> ]
[´\´Fault] ’;’
```

Continues on next page

15.2.16 MT_GripSeqL – Linear robot movement with gripper sequence
Continued

Other information

Information about	See
`MoveL` Linear robot movement	*Technical Reference Manual – Instructions, Functions and Data Types* listed in the section *References on page 11*.
`MT_MoveL` Robot linear movement	*MT_MoveJ – Robot axis movement on page 360*
`MT_GripSequence` Sequential gripper actuation	*MT_GripSequence – Sequential actuation of gripper actuators on page 347*

15.2.17 MT_GripSequence – Sequential actuation of gripper actuators

Usage

`MT_GripSequence` is used to actuate several control elements on the gripper, one after the other, using just one instruction. This makes it easier to actuate more complex grippers.

Basic examples

```
!gripper data
const grpdata gdY1_T127:=[];
const grpdata gdY2_T127:=[];
const grpdata gdY3_T127:=[];
const grpdata gdY4_T127:=[];
!gripper sequence
const grpseq gsOpen_T127{3}:=
[[gsClose,"gdY1_T127","","","","",""],
[[gsOpen,"gdY2_T127","gdY3_T127","", "","",""],
[[gsClose,"gdY4_T127","","", "","",""]];
```

In the sequence declaration `gsOpen_T127` for opening the gripper for part 127, the control element Y1 is first closed, then the control elements Y2 and Y3 are opened and then the control element Y4 is closed.

This sequence is called as follows in the robot program:

```
GripSequence\Sequence:=gsOpen_T127;
```

Arguments

```
MT_GripSequence [\Sequence] | [SeqName] [\Prefix] [\PartLoad] |
     [SetLoad] | [ResetLoad] [\ErrorNo] [\Fault]
```

`[\Sequence]` Data type: `grpseq`

Array with the gripper sequence. Only one of the arguments \Sequence or \SeqName should be used.

`[\SeqName]` Data type: `string`

Name of the gripper sequence that is to be executed. Only one of the arguments \Sequence or \SeqName should be used.

`[\Prefix]` Data type: `string`

Type prefix. Only necessary, if a type-depending gripper sequence is declared and only if the standard type prefix “T” is not used. Example of a type related gripper sequence name: gsOpenGripper_T137, if the prefix is “T” and the part type number of the current partdata declaration is 137.

`[\PartLoad]` Data type: `loaddata`

New gripper load after the sequence is executed.

`[\SetLoad]` Data type: `switch`

Sets the load of the currently selected part (see partdata).

`[\ResetLoad]` Data type: `switch`

Resets the load to load0.

Continues on next page

15.2.17 MT_GripSequence – Sequential actuation of gripper actuators
Continued

[\ErrorNo]	Data type: num Combined error domains and error number as positive integer, which can be used for display in the event of errors. The last four digits represent the error number, the digits preceding this represent the error domain.
[\Fault]	Data type: switch If the switch fault is set, a gripper related message will appear as a fault message, otherwise it will appear as a warning message.

Program execution

The valves of the control element that has been passed will be opened or closed one after the other, de[pending on the selection.

If all the sensors fail to reach the required state within the defined waiting period, then, an error message will be output.

Limitations

When using SeqName the array size for the gripper sequence is limited to 20 elements.

Restrictions

The optional parameters \PartLoad, \SetLoad and \ResetLoad cannot be used in common.

Syntax

```
MT_GripSequence
[´\´Sequence ´:=´ < expression {IN} of grpseq > ] |
[´\´SeqName ´:=´ < expression (IN) of string > ]
[´\´Prefix ´:=´ < expression (IN) of string > ]
[´\´PartLoad ´:=´ < expression (PERS) of loaddata > ] |
[´\´SetLoad] |
[´\´ResetLoad]

[´\´ErrorNo] ´:=´ < expression (IN) of num> ]
[´\´Fault]´;´
```

More information

Information about	See
Gripper data	*grpdata – Configuration of a control element of the gripper on page 257*
Gripper sequences	*grpseq – Gripper sequence for actuating several control elements on page 264*

15.2.18 MT_GripSet – Controlling the gripper

Usage

MT_GripSet is used for actuating control elements at the grippers. Up to 6 control elements can be actuated simultaneously.

Basic examples

```
MT_GripSet gsClose,gdGRP1_Y2;
```

The control element Y2 of the gripper 1 will be closed.

```
MT_GripSet gsOpen,gdGRP1_Y2\NoCheck;
```

The control element Y2 of the gripper 1 will be opened, without the sensors being monitored while doing so

```
MT_GripSet gsClose,gdGRP1_Y1\Grp2:=gdGRP1_Y2\PartLoad:=loPart;
```

The control elements Y1 and Y2 of the gripper 1 will be closed. As soon as both the control elements have attained the Closed position, the load data of the part will be activated.

```
GripSet gsReset,gdGRP1_Y2;
```

Both valve outputs for the control element Y2 of the gripper 1 will be reset (for example, while the gripper is undocking).

Arguments

```
MT_GripSet Position Grp1 [\Grp2] [\Grp3] [\Grp4] [\Grp5] [\Grp6]
     [\PartLoad] | [\SetLoad] | [\ResetLoad] [\NoCheck] |
     [\CheckOpen] | [\CheckClose] [\ErrorNo] [\Fault]
```

`Position`	Data type: `grppos` Desired position (for example, opening or closing the control element or the control elements)
`Grp1`	Data type: `grpdata` Gripper data of the first control element that is to be actuated
`[\Grp2]`	Data type: `grpdata` Gripper data of the second control element must that is to be actuated
`[\Grp3]`	Data type: `grpdata` Gripper data of the third control element that is to be actuated
`[\Grp4]`	Data type: `grpdata` Gripper data of the fourth control element that is to be actuated
`[\Grp5]`	Data type: `grpdata` Gripper data of the fifth control element that is to be actuated
`[\Grp6]`	Data type: `grpdata` Gripper data of the sixth control element that is to be actuated
`[\PartLoad]`	Data type: `loaddata` Load data that will be activated on closing or opening the gripper.
`[\SetLoad]`	Data type: `switch` Sets the load of the currently selected part (see *partdata – Part data on page 284*).

Continues on next page

15.2.18 MT_GripSet – Controlling the gripper
Continued

[\ResetLoad]	Data type: switch Resets the load to load0.
[\NoCheck]	Data type: switch If this switch is used, the system will not wait till the sensors have reached the required position.
[\CheckOpen]	Data type: switch If this switch is used, the system will wait till the “open” sensors have reached the required position. No check for “closed” sensors will be performed.
[\CheckClose]	Data type: switch If this switch is used, the system will wait till the “closed” sensors have reached the required position. No check for “open” sensors will be performed.
[\ErrorNo]	Data type: num Combined error domains and error number as positive integer, which can be used for display in the event of errors. The last four digits represent the error number, the digits preceding this represent the error domain.
[\Fault]	Data type: switch If the switch fault is set, a gripper related message will appear as a fault message, otherwise it will appear as a warning message.

Program execution

The valves of the control elements that have been passed will be opened or closed depending on the selection. Only those signals that contain a valid signal name will be considered.

To begin with, all the valves will be actuated. This is followed by a waiting time which is defined by the longest one of all available actuators.

If a sensor fails to reach the required state within the defined waiting period, then an error message will be output on the programming device and a corresponding error code will be sent for example,to an external PLC. To be able to send an error code, this must be configured in the process configuration first (see chapter System Parameters => MT API Commands).

Tip

In the "Ghost mode", the "NoGhostSet" flag is evaluated in every control element declaration. If the flag has been set to "TRUE" then the sensors will not be checked. If the flag has been set to "FALSE" then the sensors will be checked even in the ghost mode.

Tip

In the "Ghost mode" the "NoGhostCheck" flag is evaluated in every control element declaration. If the flag has been set to "TRUE", the valve will not be actuated. If the flag has been set to "FALSE", then the actuation is done even in the ghost mode.

Continues on next page

Restrictions

The optional parameters \PartLoad, \SetLoad and \ResetLoad cannot be used in common.

The optional parameters \NoCheck, \CheckOpen and \CheckClose cannot be used in common as well.

Syntax

```
MT_GripSet
[Position] ´:=´ < expression (IN) of grppos> ]
[Grp1 ´:=´ < expression (IN) of grpdata> ]
[´\´Grp2 ´:=´ < expression (IN) of grpdata> ]
[´\´Grp3 ´:=´ < expression (IN) of grpdata> ]
[´\´Grp4 ´:=´ < expression (IN) of grpdata> ]
[´\´Grp5 ´:=´ < expression (IN) of grpdata> ]
[´\´Grp6 ´:=´ < expression (IN) of grpdata> ]
[´\´PartLoad ´:=´ < expression (IN) of loaddata>]
| [´\´SetLoad´]
| [´\´ResetLoad´]
[´\´NoCheck´]
| [´\´CheckOpen´]
| [´\´CheckClose´]
[´\´ErrorNo] ´:=´ < expression (IN) of num> ]
| [´\´Fault´]´;
```

More information

Information about	See
Gripper data	*grpdata – Configuration of a control element of the gripper on page 257*
Gripper position	*grppos – Gripper position on page 262*

15.2.19 MT_GripSetType – Controlling the gripper

Usage

`MT_GripSetType` is used for actuating control elements at the grippers. Up to 6 control elements can be actuated simultaneously.

`MT_GripSetType` provides mainly the same functionality as `MT_GripSet` but considers part type specific type numbers and type prefixes as follows:

There might be different grippers for each part type in the production cell. The grippers might work differently, thus each gripper will need its own grpdata declarations.

Instead of assigning the grpdata directly as this is done with `MT_GripSet`, a string is provided to `MT_GripSetType` which represents the name of the grpdata but without part type number and part type prefix.

`MT_GripSetType` will internally complete the grpdata name, depending on the current settings for the type prefix and the type number. Then the instruction will execute the appropriate type-depending grpdata declaration.

Basic examples

Assuming, the current part type number is 6 and the standard part type prefix is “T”

```
MT_GripSetType gsClose,"gdGRP_Y2";
```

The control element Y2 of the gripper for part type 6 (gripdata gdGRP_Y2_T6) will be closed.

```
MT_GripSetType gsOpen,"gdGRP_Y2"\NoCheck\Prefix:="P";
```

The control element Y2 of the gripper for part type 6 (gripdata gdGRP_Y2_P6) will be opened, without the sensors being monitored while doing so.

```
MT_GripSetType gsClose,"gdGRP_Y1"\Grp2:="gdGRP_Y2"\PartLoad:=loPart;
```

The control elements Y1 and Y2 of the gripper for part type 6 (gripdata gdGRP_Y1_T6 and gdGRP_Y2_T6) will be closed. As soon as both the control elements have attained the closed position, the load data of the part will be activated.

```
MT_GripSetType gsReset,"gdGRP_Y2";
```

Both valve outputs for the control element Y2 of the gripper for part type 6 (gripdata gdGRP_Y2_T6) will be reset, for example, while the gripper is undocking.

Arguments

```
MT_GripSetType Position Grp1 [\Grp2] [\Grp3] [\Grp4] [\Grp5] [\Grp6]
     [\PartLoad] | [\SetLoad] | [\ResetLoad] [\NoCheck] |
     [\CheckOpen] | [\CheckClose] [\Prefix][\ErrorNo] [\Fault]
```

`Position` Data type: `grppos`

Desired position (for example, opening or closing the control element or the control elements)

`Grp1` Data type: `string`

Gripper data name of the first control element that is to be actuated, without part type prefix and without part type number.

Continues on next page

[\Grp2]	Data type: string Gripper data name of the second control element must that is to be actuated, without part type prefix and without part type number.
[\Grp3]	Data type: string Gripper data name of the third control element that is to be actuated, without part type prefix and without part type number.
[\Grp4]	Data type: string Gripper data name of the fourth control element that is to be actuated, without part type prefix and without part type number.
[\Grp5]	Data type: string Gripper data name of the fifth control element that is to be actuated, without part type prefix and without part type number.
[\Grp6]	Data type: string Gripper data name of the sixth control element that is to be actuated, without part type prefix and without part type number.
[\PartLoad]	Data type: loaddata New gripper load after the sequence is executed.
[\SetLoad]	Data type: switch Sets the load of the currently selected part (see *partdata – Part data on page 284*).
[\ResetLoad]	Data type: switch Resets the load to load0.
[\NoCheck]	Data type: switch If this switch is used, the system will not wait till the sensors have reached the required position.
[\CheckOpen]	Data type: switch If this switch is used, the system will wait till the “open” sensors have reached the required position. No check for “closed” sensors will be performed.
[\CheckClose]	Data type: switch If this switch is used, the system will wait till the “closed” sensors have reached the required position. No check for “open” sensors will be performed.
[\Prefix]	Data type: string Assigns another part type prefix apart from the default prefix.
[\ErrorNo]	Data type: num Combined error domains and error number as positive integer, which can be used for display in the event of errors. The last four digits represent the error number, the digits preceding this represent the error domain.
[\Fault]	Data type: switch If the switch fault is set, a gripper related message will appear as a fault message, otherwise it will appear as a warning message.

Program execution

The valves of the control elements that have been passed will be opened or closed depending on the selection. Only those signals that contain a valid signal name will be considered.

Continues on next page

To begin with, all the valves will be actuated. This is followed by a waiting time which is defined by the longest one of all available actuators.

If a sensor fails to reach the required state within the defined waiting period, then an error message will be output on the programming device and a corresponding error code will be sent for example,to an external PLC. To be able to send an error code, this must be configured in the process configuration first (see the chapter *MT API Commands on page 154*) .

Tip

In the "Ghost mode", the "NoGhostSet" flag is evaluated in every control element declaration. If the flag has been set to "TRUE" then the sensors will not be checked. If the flag has been set to "FALSE" then the sensors will be checked even in the ghost mode.

Tip

In the "Ghost mode" the "NoGhostCheck" flag is evaluated in every control element declaration. If the flag has been set to "TRUE", the valve will not be actuated. If the flag has been set to "FALSE", then the actuation is done even in the ghost mode.

Restrictions

The optional parameters \PartLoad, \SetLoad and \ResetLoad cannot be used in common.

The optional parameters \NoCheck, \CheckOpen and \CheckClose cannot be used in common as well.

Syntax

```
MT_GripSet
[Position] ´:=´ < expression (IN) of grppos> ]
[Grp1 ´:=´ < expression (IN) of string> ]
[´\´Grp2 ´:=´ < expression (IN) of string > ]
[´\´Grp3 ´:=´ < expression (IN) of string > ]
[´\´Grp4 ´:=´ < expression (IN) of string > ]
[´\´Grp5 ´:=´ < expression (IN) of string > ]
[´\´Grp6 ´:=´ < expression (IN) of string > ]
[´\´PartLoad ´:=´ < expression (IN) of loaddata>]
| [´\´SetLoad´]
| [´\´ResetLoad´]
[´\´NoCheck´]
| [´\´CheckOpen´]
| [´\´CheckClose´]
[´\´Prefix ´:=´ < expression (IN) of string > ]
[´\´ErrorNo] ´:=´ < expression (IN) of num> ]
| [´\´Fault´]´;´
```

Continues on next page

More information

Information about	See
Gripper data	*grpdata – Configuration of a control element of the gripper on page 257*
Gripper position	*grppos – Gripper position on page 262*

15.2.20 MT_HomeDirect – Movement directly to the home position

Usage

`MT_HomeDirect` is used in the application program to move the robot directly to the home position.

This routine has to be implemented by the integrator

Basic example

```
PROC MT_HomeDirect()
MoveJ p999,v200,fine,tGripper;
ENDPROC
```

The robot moves directly to the home position at slow speed.

Program execution

If the robot has been manually moved out of the start area of the last position that was moved to automatically (max. 150 mm) or an undefined condition exists, the `HomeRun` (see *HomeRun on page 105*) strategy cannot be used. In this case, the robot can only be moved to the home position directly or manually using the joystick.

In this case the robot must be moved manually into a free area in the vicinity of the home position. The robot is then moved to the home position directly after operator entry on the programming unit. The "MT_HomeDirect" routine is called up, in which the direct movement to the home position is stored.

Syntax

```
MT_HomeDirect ';'
```

15.2.21 MT_HomeRun – HomeRun Strategy

Usage

`MT_HomeRun` is used in the application program to define the strategy for moving the robot to the home position on the basis of the current position number.

In order to do this it must be defined for each start, end or intermediate position which signals are to be set or interrogated, and which movement routines have to be called up in order to move the robot to the next position without colliding.

The `MT_HomeRun` routine is called with the respective current position until the robot has reached the home position. This routine has to be provided by the programmer / integrator.

Basic example

```
PROC MT_HomeRun(num Position)
TEST Position
CASE 10:
mv10_999;
CASE ...
DEFAULT:
MT_ContHomeRun …;
ENDTEST
ENDPROC
```

If the robot is at position 10, then the movement routine mv10_999 is selected.

Arguments

```
MT_HomeRun Position
```

`Position` — Data type: `num`

Two to six digit position number that is used to decide where the robot is going next.

Program execution

HomeRun calls the MT_HomeRun routine and passes the current robot position as a two to six-digit value. Within MT_HomeRun the decision is made how to continue to the next position from this position and what conditions (signals, open/close gripper) must be observed.

The call-up of MT_HomeRun is repeated until the robot is in the home position.

Tip

If the robot is back at the starting position (for example, position 30 → 31 → 30) after a program part repetition (for example, loading the conveyor), the variable bMT_HomeRunCheckPos must be set to FALSE to avoid an error message (position check deactivation).

Example:

```
TEST Position
```

Continues on next page

15.2.21 MT_HomeRun – HomeRun Strategy
Continued

```
CASE 30:
IF diPartinGripper=high THEN
Load_Belt;
bMT_HomeRunCheckPos:=FALSE;
ELSE
mv30_999;
ENDIF
CASE ...
ENDTEST
```

If the robot is in position 30 (belt pre-position) and has a part in the gripper, the routine for loading the belt is called up. Since the robot has subsequently returned to position 30, variable `bMT_HomeRunCheckPos` is set to FALSE.

Syntax

```
MT_HomeRun
[Position ':=' ] < expression (IN) of num> ';'
```

More information

Information about	See
Creating the HomeRun strategy	*Strategy for automatic movement into the home position on page 134*
Continuation of robot movement	*MT_ContHomeRun – Continue a movement routine on page 310*
Backwards processing of a movement routine	*MT_MoveRoutine – Execute a movement routine at HomeRun on page 389*

15.2.22 MT_HomeRunSavePos – Saving the stop position

Usage

MT_HomeRunSavePos is used to save the current robot position (robtarget) as soon as the robot stops moving.

With the aid of this position a check is made in HomeRun whether the robot has been moved manually.

Basic example

```
PROC main()
...
IF ... RAISE 1;
ERROR
IF ERRNO=1 THEN
MT_HomeRunSavePos;
RETURN;
ENDIF
ENDPROC
```

The current robot position is saved in the error handler and the program is restarted from the beginning (start from "main“).

Program execution

If the move to the home position is to take place without a program stop from the program sequence or the error handling or triggered via an interrupt, the current robot position must first be saved by directly calling the MT_HomeRunSavePos instruction.

Example:

```
TRAP T_HomePos
IDelete irHomePos;
StopMove;
MT_HomeRunSavePos;
ExitCycle;
ENDTRAP
```

Syntax

```
MT_HomeRunSavePos ';'
```

More information

Information about	See
`MT_Exit` - Program processing complete	*MT_Exit – Program processing complete on page 320*
`MT_ExitCycle` – Abort current cycle and start next cycle.	*MT_ExitCycle – Abort current cycle and start next cycle on page 321*

15.2.23 MT_MoveJ – Robot axis movement

Usage

MT_MoveJ is used to move the robot quickly from one point to another, if this movement does not have to be in a straight line. When the robot reaches the destination position, the numeric value that is passed is saved as the current position.

The robot and the external axes move to the destination position along a non-linear path. All axes reach the destination position simultaneously.

This instruction can only be used in the Main task T_ROB1 or in motion tasks in the case of a MultiMove system.

The instruction basically corresponds to a MoveJ with some additions.

Basic example

Position No. 11:

```
MT_MoveJ 11, p11, v1000, z30, tGripper;
```

The TCP of the tGripper tool moves to position p11 in an axle-related way at speed v1000 and zone data z30. When the position is reached, "11" is stored as the current position.

Intermediate position of movement from 10 to 11:

```
MT_MoveJ 111001,* ,vmax, z30, tGripper;
```

The TCP of the tGripper tool moves in an axis-related way to the position programmed in the instruction (marked with an *) and then saves “111001” as current position.

Arguments

```
MT_MoveJ [\Conc] ActPos ToPoint speed [\T] zone [\Inpos]
Tool [ \Wobj] [\NoMove]
```

`[\Conc] Concurrent`
Data type: `switch`
The following instructions are executed whilst the robot is in motion.
Further information can be obtained from the MoveJ instruction.

`ActPos`
Data type: `dnum`
Contains the position number of the position to be moved to.

`ToPoint`
Data type: `robtarget`
The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

`speed`
Data type: `speeddata`
The speed programmed for the movement. The speed data define the velocity of the TCP, of the tool reorientation and of external axes.

`[\T] (Time)`
Data type: `num`
This argument is used to specify the time in seconds during which a movement of the manipulator and of the external axes should be executed. This value is then substituted for the corresponding speed data.

Continues on next page

`Zone`	Data type: `zonedata` Zone data for the movement. Zone data describe the distance in which the axes must stand from the destination point before the next instruction is executed.
`[\Inpos]`	Data type: `stoppointdata` This argument is used to specify the convergence criteria for the position of the robot's TCP in the stop point. The stop point data substitutes the zone specified in the Zone parameter.
`Tool`	Data type: `tooldata` The tool in use when the robot moves. The tool centre point is moved to the specified destination point.
`[\Wobj]`	Data type: `wobjdata` The work object (tool coordinate system) to which the robot position in the instruction is related. This argument can be omitted. In this case the position relates to the world coordinate system.
`[\NoMove]`	Data type: `switch` This switch is used in the first position of a movement routine and serves to avoid stop functions by moving to a position twice.

Program execution

After reaching the programmed position, the transferred position number is saved as current robot position.

If the robot is moved to the home position using HomeRun, the position number-dependent movement routine is called up by the "MT_HomeRun" routine, and the first position to be moved to (start position) is searched for, i.e. no movement takes place until the saved robot position is identical with the position number in the movement instruction.

Since the “linear” or “axis-related” movement mode is saved whenever a movement command is executed, it is ensured that the movement to the start position is performed with the same movement mode that was used previously.

To find the start position of a movement, the two-digit respectively the three-digit start point of the movement must always be contained as first movement command in the movement routine. The \NoMove switch is set for this instruction to avoid stop points.

Note

In programming mode, this position (\NoMove) is only moved to if the program line is executed forwards one instruction at a time.

Syntax

```
MT_MoveJ
['\'Conc ',']
[ActPos ':='] < expression (IN) of dnum> ','
[ToPoint ':='] < expression (IN) of robtarget > ','
[speed ':='] < expression (IN) of speeddata >
['\'T ':=' < expression (IN) of num > ',']
[zone ':='] < expression (IN) of zonedata >
```

Continues on next page

15.2.23 MT_MoveJ – Robot axis movement
Continued

```
['\'Inpos':='< expression (IN) of stoppointdata > ',']
[Tool ':='] < persistent (PERS) of tooldata >
['\'WObj ':=' < persistent (PERS) of wobjdata > ]
['\'NoMove ] ';'
```

Other information

Information about	See
`MoveJ` - Robot axis movement	*Technical Reference Manual – Instructions, Functions and Data Types* listed in the section *References on page 11*.

15.2.24 MT_MoveJDO – Robot axis movement and setting of a digital output

Usage

MT_MoveJDO is used to move the robot quickly from one point to another, if this movement does not have to be in a straight line.

The position number assignment and the setting/resetting of the digital output signal take place in the middle of the corner path.

The robot and the external axes move to the destination position along a non-linear path. All axes reach the destination position simultaneously.

This instruction can only be used in the Main task T_ROB1 or in motion tasks in the case of a MultiMove system.

The instruction basically corresponds to a MoveJDO with additions extensions.

Basic example

```
MT_MoveJDO 11, p11, v1000, z30, tGripper, doIRBoutArea, 1;
```

The TCP of the tGripper tool moves in an axis-related way to position p11 at speed v1000 and with zone data z30. In the middle of path of the zone of p11 the position number "11" is saved as the current position and output doIRBoutArea is set to "1".

Arguments

```
MT_MoveJDO [\Conc] ActPos ToPoint speed [\T] zone [\Inpos] Tool
[\Wobj] Signal Value [\DODelay]
```

[\Conc] Concurrent	Data type: switch The following instructions are executed whilst the robot is in motion. Further information can be obtained from the MoveJ instruction.
ActPos	Data type: dnum Contains the position number of the position to be moved to
ToPoint	Data type: robtarget The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).
Speed	Data type: speeddata The speed programmed for the movement. The speed data define the velocity of the TCP, of the tool reorientation and of external axes.
[\T](Time)	Data type: num This argument is used to specify the time in seconds during which a movement of the manipulator and of the external axes should be executed. This value is then substituted for the corresponding speed data.
Zone	Data type: zonedata Zone data for the movement. Zone data describe the distance in which the axes must stand from the destination point before the next instruction is executed.

Continues on next page

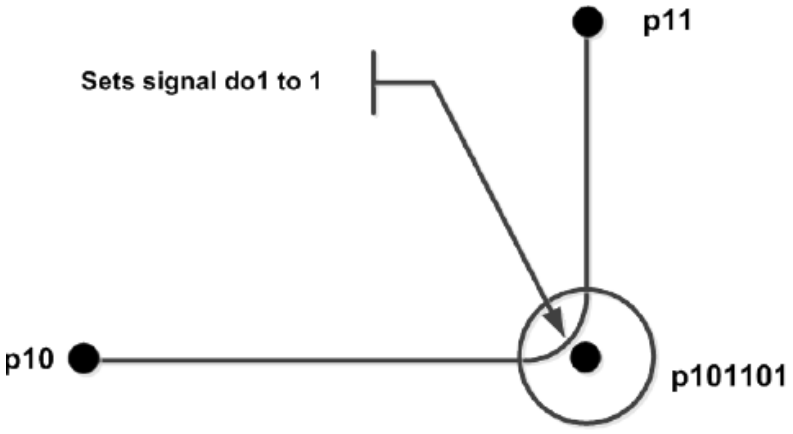
15.2.24 MT_MoveJDO – Robot axis movement and setting of a digital output
Continued

`[\Inpos]`	Data type: `stoppointdata` This argument is used to specify the convergence criteria for the position of the robot's TCP in the stop point. The stop point data substitutes the zone specified in the Zone parameter.
`Tool`	Data type: `tooldata` The tool in use when the robot moves. The tool centre point is moved to the specified destination point.
`[\Wobj]`	Data type: `wobjdata` The work object (tool coordinate system) to which the robot position in the instruction is related. This argument can be omitted. In this case the position relates to the world coordinate system.
`Signal`	Data type: `signaldo` The name of the digital output.
`Value`	Data type: `dionum` The required value of the signal (0 or 1 or high or low).
`[\DODelay] Digital Output Delay`	Data type: `num` Used to delay the setting of output signals after the robot has reached the specified position. There is no delay if this argument is omitted. The delay is not synchronised with the movement.

Program execution

More information about axis-specific movements can be found in the explanations for instruction MoveJDO or MoveJ.

The digital output signal and the passed position number are set/reset in the middle of the corner path for fly-by points, as shown in the figure below.




```
SMoveJDO p101101,v1000,z30,tGripper,do1,1;
```

en1200000791

The specified I/O signal is set/reset in continuous and "forwards one step at a time" execution mode, but not in "backwards one step at a time" execution mode.

If the robot is moved to the home position using the HomeRun routine, the position number-dependent movement routine is called up by the "MT_HomeRun" routine, and the first position to be moved to (start position) is searched for, i.e. no

Continues on next page

movement takes place until the saved robot position is identical with the position number in the movement instruction.

Note

The MT_MoveJDO instruction may never be used as the first movement instruction in a movement routine. MT_MoveL or MT_MoveJ with the \NoMove argument must always be used for this purpose.

Note

No digital output processing takes place during backwards instruction by instruction execution or when searching for the first position, nor during backwards movement with "MT_MoveRoutine".

Tip

The output is not set or reset during movement to the home position if the execution of trigger events (`ExecTriggEvt`) has been disabled in the system parameters (FALSE).

Syntax

```
MT_MoveJDO
['\'Conc ',']
[ActPos ':='] < expression (IN) of dnum> ','
[ToPoint ':='] < expression (IN) of robtarget > ','
[speed ':='] < expression (IN) of speeddata >
['\'T':=' < expression (IN) of num > ] ','
[zone ':='] < expression (IN) of zonedata >
['\'Inpos ':=' < expression (IN) of stoppointdata >] ','
[Tool ':='] < persistent (PERS) of tooldata >
['\'WObj ':=' < persistent (PERS) of wobjdata > ]
[Signal':=' ] < variable (VAR) of signaldo >
[Value':=' ] < expression (IN) of dionum >
['\'DODelay':=' < variable (IN) of num > ]';'
```

Other information

Information about	See
`MoveJ` - Robot axis movement	*Technical Reference Manual – Instructions, Functions and Data Types* listed in the section *References on page 11*.
`MoveJDO` - Robot axis movement and setting of a digital output in the corner path	*Technical Reference Manual – Instructions, Functions and Data Types* listed in the section *References on page 11*.

15.2.25 MT_MoveJGO – Robot axis movement and setting of a group output

Usage

MT_MoveJGO is used to move the robot quickly from one point to another, if this movement does not have to be in a straight line.

The position number assignment and the setting of the group output signal take place in the middle of the corner path.

The robot and the external axes move to the destination position along a non-linear path. All axes reach the destination position simultaneously.

This instruction can only be used in the Main task T_ROB1 or in motion tasks in the case of a MultiMove system.

Basic example

```
MT_MoveJGO 11, p11, v1000, z30, tGripper, goArea, 11;
```

The TCP of the tGripper tool moves in an axis-related way to position p11 at speed v1000 and with zone data z30, and then saves "11" as the current position. Group output goArea is set to 11 in the middle of the path in the zone of p11.

```
MT_MoveJGO 11, p11, v1000, z30, tGripper, goArea,0xFFFFFF;
```

The 32-bit group output goArea is set to the hexadecimal value "FFFFFF" or 4294967295 in the middle of the path.

Arguments

```
MT_MoveJGO [\Conc] ActPos ToPoint speed [\T] zone [\Inpos] Tool
[\Wobj] [Signal Value [\DODelay]
```

`[\Conc]`
`Concurrent`

Data type: `switch`

The following instructions are executed whilst the robot is in motion.

Further information can be obtained from the MoveL instruction.

`ActPos`

Data type: `dnum`

Contains the position number of the position to be moved to

`ToPoint`

Data type: `robtarget`

The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

`speed`

Data type: `speeddata`

The speed programmed for the movement. The speed data define the velocity of the TCP, of the tool reorientation and of external axes.

`[\T](Time)`

Data type: `num`

This argument is used to specify the time in seconds during which a movement of the manipulator and of the external axes should be executed. This value is then substituted for the corresponding speed data.

`zone`

Data type: `zonedata`

Zone data for the movement. Zone data describe the distance in which the axes must stand from the destination point before the next instruction is executed.

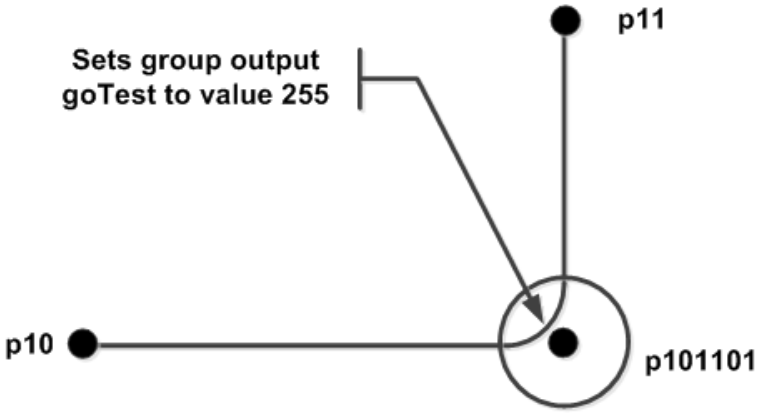
Continues on next page

[\Inpos]	Data type: stoppointdata This argument is used to specify the convergence criteria for the position of the robot's TCP in the stop point. The stop point data substitutes the zone specified in the Zone parameter.
Tool	Data type: tooldata The tool in use when the robot moves. The tool centre point is moved to the specified destination point.
[\Wobj]	Data type: wobjdata The work object (tool coordinate system) to which the robot position in the instruction is related. This argument can be omitted. In this case the position relates to the world coordinate system.
Signal	Data type: signalgo The name of the digital group output
Value	Data type: dnum The required integer value of the signal between 0 and 4294967295 (max. group output width 32 bits)
[\DODelay] Digital Output Delay	Data type: num Used to delay the setting of the group output after the robot has reached the specified position. There is no delay if this argument is omitted. The delay is not synchronised with the movement.

Program execution

More information about axis-specific movements can be found in the explanations for instruction MoveJDO or MoveJ.

The digital group output and the passed position number are set/reset in the middle of the corner path for fly-by points, as shown in the figure below.




```
SMoveJGO p101101,v1000,z30,tGripper,goTest,255;
```

en1300000155

Continues on next page

15.2.25 MT_MoveJGO – Robot axis movement and setting of a group output
Continued

The specified I/O signal is set/reset in continuous and "forwards one step at a time" execution mode, but not in "backwards one step at a time" execution mode.

Note

The MT_MoveJGO instruction may never be used as the first movement instruction in a movement routine. MT_MoveL or MT_MoveJ with the \NoMove argument must always be used for this purpose.

Note

No digital group output processing takes place during backwards instruction by instruction execution or when searching for the first position, nor during backwards movement with "MT_MoveRoutine".

Tip

The digital group output is not set or reset during movement to the home position if the execution of trigger events (ExecTriggEvt) has been disabled in the system parameters (FALSE).

Syntax

```
MT_MoveJGO
['\'Conc ',']
[ActPos ':='] < expression (IN) of dnum> ','
[ToPoint ':='] < expression (IN) of robtarget > ','
[speed ':='] < expression (IN) of speeddata >
['\'T':=' < expression (IN) of num > ] ','
[zone ':='] < expression (IN) of zonedata >
['\'Inpos ':=' < expression (IN) of stoppointdata >] ','
[Tool ':='] < persistent (PERS) of tooldata >
['\'WObj ':=' < persistent (PERS) of wobjdata > ]
[Signal':=' ] < variable (VAR) of signalgo >
[Value':=' ] < expression (IN) of dionum >
['\'DODelay':=' < variable (IN) of num > ]';'
```

Other information

Information about	See
MoveJ - Robot axis movement	*Technical Reference Manual – Instructions, Functions and Data Types*
MoveJDO - Robot axis movement and setting of a digital output in the corner path	*Technical Reference Manual – Instructions, Functions and Data Types*

15.2.26 MT_MoveJSync – Axis-wise movement and processing a procedure.

Usage

MT_MoveJSync (Move Joint Synchronously) is used to move the robot quickly from one point to another, if this movement does not have to be in a straight line.

The position number assignment and the execution of the specified RAPID procedure takes place in the centre of the corner path.

The robot and the external axes move to the destination position along a non-linear path. All axes reach the destination position simultaneously.

This instruction can only be used in the Main task T_ROB1 or in motion tasks in the case of a MultiMove system.

The instruction basically corresponds to a MoveJSync with the addition of the position number.

Basic example

Example 1:

```
MT_MoveJSync 111001,p111001,vmax, z30, tGripper,"OpenGripper";
```

The TCP of the tGripper tool moves in an axis-related way to the position programmed in the instruction p111001 and then saves 111001 as the current position. The OpenGripper procedure OpenGripper is processed in the middle of the corner path at p111001 .

Example 2:

```
MT_MoveJSync 111002,*,vmax,z30,tGripper,"MySync"\PNum:=2;
```

The TCP moves in an axis-related way to the position programmed in the instruction and calls up the procedure "MySync“ with the numeric passing parameter "2".

Arguments

```
MT_MoveJSync [\Conc] ActPos ToPoint speed [\T] zone [\Inpos] Tool
     [\Wobj] ProcName [\PNum] [\PDnum] [\PStr]
```

`[\Conc]` `Concurrent`
Data type: `switch`
The following instructions are executed whilst the robot is in motion.
Further information can be obtained from the MoveL instruction.

`ActPos`
Data type: `dnum`
Contains the position number of the position to be moved to

`ToPoint`
Data type: `robtarget`
The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

`speed`
Data type: `speeddata`
The speed programmed for the movement. The speed data define the velocity of the TCP, of the tool reorientation and of external axes.

`[\T] (Time)`
Data type: `num`
This argument is used to specify the time in seconds during which a movement of the manipulator and of the external axes should be executed. This value is then substituted for the corresponding speed data.

Continues on next page

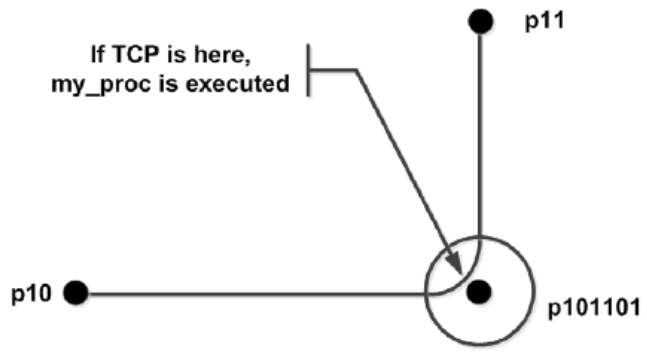
15.2.26 MT_MoveJSync – Axis-wise movement and processing a procedure.
Continued

`Zone`	Data type: `zonedata` Zone data for the movement. Zone data describe the distance in which the axes must stand from the destination point before the next instruction is executed.
`[\Inpos]`	Data type: `stoppointdata` This argument is used to specify the convergence criteria for the position of the robot's TCP in the stop point. The stop point data substitutes the zone specified in the Zone parameter.
`Tool`	Data type: `tooldata` The tool in use when the robot moves. The tool centre point is moved to the specified destination point.
`[\Wobj]`	Data type: `wobjdata` The work object (tool coordinate system) to which the robot position in the instruction is related. This argument can be omitted. In this case the position relates to the world coordinate system.
`ProcName`	Data type: `string` Name of the RAPID procedure to be executed at the middle of the corner path in the destination point.
`[\PNum ]`	Data type: `num` Numeric passing parameter to be passed to the calling RAPID procedure.
`[\PDnum ]`	Data type: `dnum` Double numeric passing parameter to be passed to the calling RAPID procedure.
`[\PStr ]`	Data type: `string` String to be passed to the calling RAPID procedure.

Program execution

More information about axis-specific movements can be found in the explanations for instruction MoveJSync or MoveJ.

The specified RAPID procedure and the position number are processed when the TCP reaches the middle of the corner path at the destination point of the MT_MoveJSync instruction.




```
SMoveJSync p101101,v1000,z30,tGripper,"my_proc";
```

en1200000793

Continues on next page

Execution of the specified RAPID procedure in different execution modes:

Execution mode	Execution of RAPID procedure
Continuously or Cycle	According to this description
Forward step	In the stop point
Backward step	Not at all
During backwards movement with MT_MoveRoutine	Not at all
Use of "fine" points	At the stop point

`MT_MoveJSync` is an encapsulation of the `TriggInt` and `TriggJ` instructions. The procedure call is executed at TRAP level.

If the centre of the corner path at the destination point is reached during the slowdown after a program stop, the procedure is not called up (program execution is stopped). The procedure call is executed when the next program start takes place.

Note

The MT_MoveJSync instruction may never be used as the first movement instruction in a movement routine. MT_MoveL or MT_MoveJ with the \NoMove argument must always be used for this purpose.

Note

The specified RAPID procedure is not processed during backwards instruction by instruction execution or when searching for the first position, nor during backwards movement with "MT_MoveRoutine".

Tip

The RAPID procedure is not executed during movement to the home position if the execution of trigger events (ExecTriggEvt) has been disabled in the system parameters (FALSE).

Use of the RAPID routine

Local routines

A routine that is declared locally in the module can be called up by specifying the module name, for example, "Module:Routine"

Example:

```
MT_MoveJSync 10,p10,v1000,z10,tGripper,"MyModule:MySync
MODULE MyModule
LOCAL PROC MySync()
…
ENDPROC
ENDMODULE
```

The procedure that is declared locally in the "MyModule" module ("MySync") will call up the centre of the corner path p10.

Continues on next page

15.2.26 MT_MoveJSync – Axis-wise movement and processing a procedure.
Continued

Use of parameters

A num, a dnum and/or a string parameter can also be passed to a RAPID routine.

During the creation of the calling RAPID routine it must be ensured that it contains the relevant passing parameters.

If several parameters are used, the order (num, dnum, string) must be taken into consideration.

Example:

```
PROC mv10_11()
MT_MoveJSync 101101,*,v1000,z10,tGripper,"MySync1"\PNum:=1;
MT_MoveJSync 101102,*,v1000,z10,tGripper,"MySync2"
\PStr:="Test";
MT_MoveJSync 101103,*,v1000,z10,tGripper,"MySync3"
\PNum:=5\PDnum:=10\PStr:="Test";
ENDPROC…
PROC MySync1(num Value)
Tpwrite "Sync 1: "\Num:=Value;
ENDPROC
PROC MySync2(string Value)
Tpwrite "Sync 2: "+Value;
ENDPROC
PROC MySync3(num Value1, dnum Value2, string Value3)
Tpwrite "Sync 3: "+Valtostr(Value1)+ "/"+ Value3;
TEST Value2
CASE 10:
…
CASE 11:
…
ENDTEST
ENDPROC
```

Limitations

The MT_MoveJSync instruction cannot be used at interrupt level.

The specified RAPID procedure cannot be tested with stepwise execution.

When the robot reaches the centre of the corner path, the RAPID routine is usually executed after a delay of 20-30s, depending on the type pf movement being made at the time.

A change of execution mode from continuous or cyclic to forwards or backwards after a program stop results in an error. This error tells the user that the mode change can result in the RAPID procedure being omitted from the queue for execution on the path.

Syntax

```
MT_MoveJSync
['\'Conc ',']
[ActPos ':='] < expression (IN) of dnum> ','
[ToPoint ':='] < expression (IN) of robtarget > ','
[Speed ':=' ] < expression (IN) of speeddata >
['\'T':=' < expression (IN) of num > ] ','
```

Continues on next page

```
[Zone ':=' ] < expression (IN) of zonedata >
['\'Inpos ':=' < expression (IN) of stoppointdata > ]',''
[Tool ':='] < persistent (PERS) of tooldata >
['\'WObj ':=' < persistent (PERS) of wobjdata > ]
[ProcName ':=' ] < variable (IN) of string >
['\'PNum:='< expression (IN) of num >]',''
['\'PDnum':='< expression (IN) of dnum >]
['\'PStr':='< expression (IN) of string >]';'
```

Other information

Information about	See
`MoveJSync` - Move robot by means of axis movement and execution of a RAPID procedure	*Technical Reference Manual – Instructions, Functions and Data Types* listed in the section *References on page 11*.
`TriggInt` - Definition of a position-dependent interrupt	*Technical Reference Manual – Instructions, Functions and Data Types* listed in the section*References on page 11*.
`TriggJ` - Axis-wise robot movement with events.	*Technical Reference Manual – Instructions, Functions and Data Types* listed in the section *References on page 11*.

15.2.27 MT_MoveL – Linear robot movement.

Usage

MoveL is used to move the tool centre point (TCP) linearly to a specified destination. If the TCP is to remain stationary, this instruction can also be used to reorient the tool.

When the robot reaches the destination position, the numeric value that is passed is saved as the current position.

This instruction can only be used in the Main task T_ROB1 or in motion tasks in the case of a MultiMove system.

The instruction basically corresponds to a MoveL with some additions.

Basic example

Position No. 11:

MT_MoveL 11, p11, v1000, z30, tGripper;

The TCP of the tGripper tool moves to position p11 linearly at speed v1000 and with zone data z30. When the position is reached, "11" is stored as the current position.

Intermediate position of movement from 10 to 11:

MT_MoveL 111001,* ,vmax, z30, tGripper;

The TCP of the tGripper tool moves linearly to the position programmed in the instruction (marked with an *) and then saves “111001” as current position.

Arguments

```
MT_MoveL [\Conc] ActPos ToPoint speed [\T] zone [\Inpos]
Tool [ \Wobj] [\NoMove] [\Corr]
```

`[\Conc]` `Concurrent`
Data type: `switch`
The following instructions are executed whilst the robot is in motion.
Further information can be obtained from the MoveL instruction.

`ActPos`
Data type: `dnum`
Contains the position number of the position to be moved to

`ToPoint`
Data type: `robtarget`
The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

`speed`
Data type: `speeddata`
The speed programmed for the movement. The speed data define the velocity of the TCP, of the tool reorientation and of external axes.

`[\T] (Time)`
Data type: `num`
This argument is used to specify the time in seconds during which a movement of the manipulator and of the external axes should be executed. This value is then substituted for the corresponding speed data.

Continues on next page

`Zone`	Data type: `zonedata` Zone data for the movement. Zone data describe the distance in which the axes must stand from the destination point before the next instruction is executed.
`[\Inpos]`	Data type: `stoppointdata` This argument is used to specify the convergence criteria for the position of the robot's TCP in the stop point. The stop point data substitutes the zone specified in the Zone parameter.
`Tool`	Data type: `tooldata` The tool in use when the robot moves. The tool centre point is moved to the specified destination point.
`[\Wobj]`	Data type: `wobjdata` The work object (tool coordinate system) to which the robot position in the instruction is related. This argument can be omitted. In this case the position relates to the world coordinate system.
`[\Corr]`	Data type: `switch` Correction data that has been written in a correction entry using the CorrWrite instruction is added to the path and the destination position if this argument is present.
`[\NoMove]`	Data type: `switch` This switch is used in the first position of a movement routine and serves to avoid stop functions by moving to a position twice.

Program execution

After reaching the programmed position, the transferred position number is saved as current robot position.

If the robot is moved to the home position using the HomeRun routine, the position number-dependent movement routine is called up by the "MT_HomeRun" routine, and the first position to be moved to (start position) is searched for, i.e. no movement takes place until the saved robot position is identical with the position number in the movement instruction.

Since the “linear” or “axis-related” movement mode is saved in every movement command, it is assured that the movement to the start position is performed with the same movement mode as was used previously.

To find the start position of a movement, the two-digit respectively the three-digit start point of the movement must always be contained as first movement command in the movement routine. The \NoMove switch is set for this instruction to avoid stop points.

Note

In programming mode, this position (\NoMove) is only moved to if the program line is executed forwards one instruction at a time.

Syntax

```
MT_MoveL
['\'Conc ',']
[ActPos ':='] < expression (IN) of dnum> ','
[ToPoint ':='] < expression (IN) of robtarget > ','
```

Continues on next page

15.2.27 MT_MoveL – Linear robot movement.
Continued

```
[Speed ':=' ] < expression (IN) of speeddata >
['\'T ':=' < expression (IN) of num > ',']
[Zone ':=' ] < expression (IN) of zonedata >
['\'Inpos':='< expression (IN) of stoppointdata > ',']
[Tool ':='] < persistent (PERS) of tooldata >
['\'WObj ':=' < persistent (PERS) of wobjdata > ]
['\'NoMove ]
['\'Corr ]';
```

Other information

Information about	See
`MoveJSync` - Move robot by means of axis movement and execution of a RAPID procedure	*Technical Reference Manual – Instructions, Functions and Data Types* listed in the section *References on page 11*.

15.2.28 MT_MoveLDO – Linear movement and setting a digital output in the zone

Usage

MT_MoveLDO is used to move the tool centre point (TCP) linearly to a specified destination. The position number assignment and the setting/resetting of the digital output signal take place in the middle of the corner path.

If the TCP is to remain stationary, this instruction can also be used to reorient the tool.

This instruction can only be used in the Main task T_ROB1 or in motion tasks in the case of a MultiMove system

The instruction basically corresponds to a MoveLDO with some additions.

Basic example

```
MT_MoveLDO 11, p11, v1000, z30, tGripper, doIRBoutArea, 1;
```

The TCP of the tGripper tool moves in a linear way to position p11 at speed v1000 and with zone data z30. In the middle of path of the zone of p11 the position number "11" is saved as the current position and output doIRBoutArea is set to "1".

Arguments

```
MT_MoveLDO
[\Conc] ActPos ToPoint speed [\T] zone [\Inpos] Tool [\Wobj] Signal
     Value [\DODelay] [\Corr]
```

`[\Conc]` `Concurrent`
Data type: `switch`
The following instructions are executed whilst the robot is in motion.
Further information can be obtained from the MoveL instruction.

`ActPos`
Data type: `dnum`
Contains the position number of the position to be moved to

`ToPoint`
Data type: `robtarget`
The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

`speed`
Data type: `speeddata`
The speed programmed for the movement. The speed data define the velocity of the TCP, of the tool reorientation and of external axes.

`[\T] (Time)`
Data type: `num`
This argument is used to specify the time in seconds during which a movement of the manipulator and of the external axes should be executed. This value is then substituted for the corresponding speed data.

`Zone`
Data type: `zonedata`
Zone data for the movement. Zone data describe the distance in which the axes must stand from the destination point before the next instruction is executed.

`[\Inpos]`
Data type: `stoppointdata`
This argument is used to specify the convergence criteria for the position of the robot's TCP in the stop point. The stop point data substitutes the zone specified in the Zone parameter.

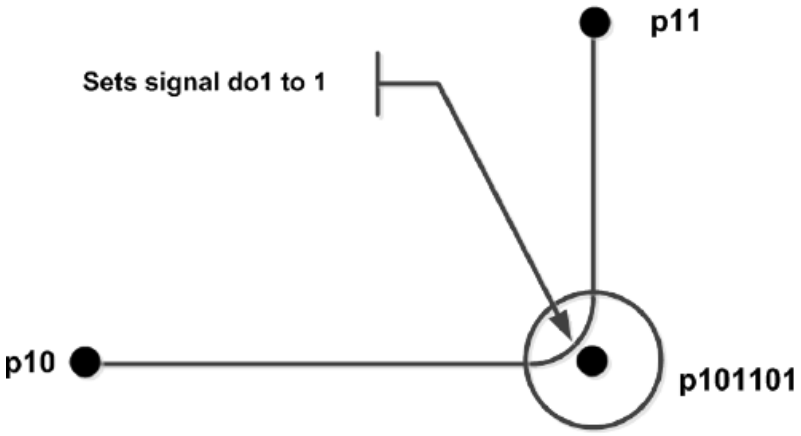
Continues on next page

15.2.28 MT_MoveLDO – Linear movement and setting a digital output in the zone
Continued

`Tool`	Data type: `tooldata` The tool in use when the robot moves. The tool centre point is moved to the specified destination point.
`[\Wobj]`	Data type: `wobjdata` The work object (tool coordinate system) to which the robot position in the instruction is related. This argument can be omitted. In this case the position relates to the world coordinate system.
`Signal`	Data type: `signaldo` The name of the digital output.
`Value`	Data type: `dionum` The required value of the signal (0 or 1 or high or low).
`[\DODelay]` `Digital Output Delay`	Data type: `num` Used to delay the setting of output signals after the robot has reached the specified position. There is no delay if this argument is omitted. The delay is not synchronised with the movement.
`[\Corr]`	Data type: `switch` Correction data that has been written in a correction entry using the CorrWrite instruction is added to the path and the destination position if this argument is present.

Program execution

More information about linear movements can be found in the explanations for instruction MoveLDO or MoveL.




```
SMoveLDO p101101,v1000,z30,tGripper,do1,1;
```

en1200000794

The digital output signal and the passed position number are set/reset in the middle of the corner path for fly-by points, as shown in the figure below.

The specified I/O signal is set/reset in continuous and "forwards one step at a time" execution mode, but not in "backwards one step at a time" execution mode.

If the robot is moved to the home position using the `HomeRun` routine, the position number-dependent movement routine is called up by the "MT_HomeRun" routine, and the first position to be moved to (start position) is searched for, i.e. no

Continues on next page

movement takes place until the saved robot position is identical with the position number in the movement instruction.

Note

The MT_MoveLDO instruction may never be used as the first movement instruction in a movement routine. MT_MoveL or MT_MoveJ with the \NoMove argument must always be used for this purpose.

Note

No digital output processing takes place during backwards instruction by instruction execution or when searching for the first position, nor during backwards movement with "MT_MoveRoutine".

Tip

The output is not set or reset during movement to the home position if the execution of trigger events (ExecTriggEvt) has been disabled in the system parameters (FALSE).

Syntax

```
MT_MoveLDO
['\'Conc ',']
[ActPos ':='] < expression (IN) of dnum> ','
[ToPoint ':='] < expression (IN) of robtarget > ','
[speed ':='] < expression (IN) of speeddata >
['\'T':=' < expression (IN) of num > ] ','
[zone ':='] < expression (IN) of zonedata >
['\'Inpos ':=' < expression (IN) of stoppointdata > ]','
[Tool ':='] < persistent (PERS) of tooldata >
['\'WObj ':=' < persistent (PERS) of wobjdata > ]
[Signal':=' ] < variable (VAR) of signaldo >
[Value':=' ] < expression (IN) of dionum >
['\'DODelay':=' < variable (IN) of num > ]
['\'Corr ]';'
```

Other information

Information about	See
`MoveL` – Linear robot movement.	*Technical Reference Manual – Instructions, Functions and Data Types* listed in the section *References on page 11*.
`MoveLDO` - Linear robot movement and setting of a digital output in the zone	*Technical Reference Manual – Instructions, Functions and Data Types* listed in the section *References on page 11*.

15.2.29 MT_MoveLGO – Linear robot movement and set group output in zone.

Usage

MT_MoveLGO is used to move the tool centre point (TCP) linearly to a specified destination. The position number assignment and the setting of the digital group output take place in the middle of the corner path.

If the TCP is to remain stationary, this instruction can also be used to reorient the tool.

This instruction can only be used in the Main task T_ROB1 or in motion tasks in the case of a MultiMove system.

Basic example

MT_MoveLGO 11, p11, v1000, z30, tGripper, goArea, 11;

The TCP of the tGripper tool moves in a linear way to position p11 at speed v1000 and with zone data z30, and then saves "11" as the current position. Group output goArea is set to 11 in the middle of the path in the zone of p11.

MT_MoveJGO 11, p11, v1000, z30, tGripper, goArea,0xFFFFFF;

The 32-bit group output goArea is set to the hexadecimal value "FFFFFF" or 4294967295 in the middle of the path.

Arguments

```
MT_MoveLGO
[\Conc] ActPos ToPoint speed [\T] zone [\Inpos]
Tool [\Wobj] Signal Value [\DODelay] [\Corr]
```

`[\Conc]`
`Concurrent`

Data type: `switch`

The following instructions are executed whilst the robot is in motion.

Further information can be obtained from the `MoveL` instruction.

`ActPos`

Data type: `dnum`

Contains the position number of the position to be moved to

`ToPoint`

Data type: `robtarget`

The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

`speed`

Data type: `speeddata`

The speed programmed for the movement. The speed data define the velocity of the TCP, of the tool reorientation and of external axes.

`[\T] (Time)`

Data type: `num`

This argument is used to specify the time in seconds during which a movement of the manipulator and of the external axes should be executed. This value is then substituted for the corresponding speed data.

`Zone`

Data type: `zonedata`

Zone data for the movement. Zone data describe the distance in which the axes must stand from the destination point before the next instruction is executed.

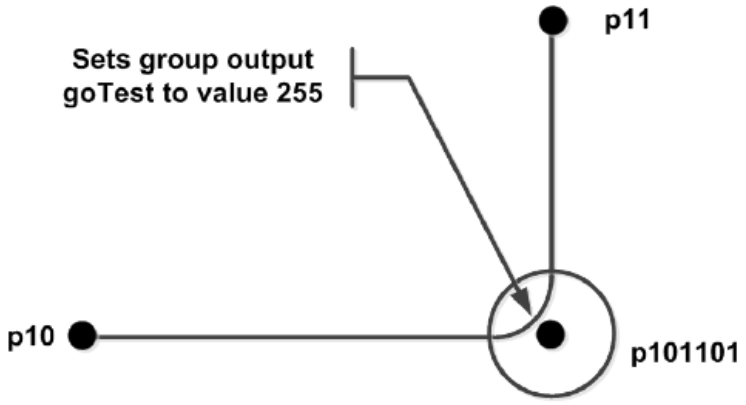
Continues on next page

[\Inpos]	Data type: stoppointdata This argument is used to specify the convergence criteria for the position of the robot's TCP in the stop point. The stop point data substitutes the zone specified in the Zone parameter.
Tool	Data type: tooldata The tool in use when the robot moves. The tool centre point is moved to the specified destination point.
[\Wobj]	Data type: wobjdata The work object (tool coordinate system) to which the robot position in the instruction is related. This argument can be omitted. In this case the position relates to the world coordinate system.
Signal	Data type: signaldo The name of the digital output.
Value	Data type: dionum The required value of the signal (0 or 1 or high or low).
[\DODelay] Digital Output Delay	Data type: num Used to delay the setting of output signals after the robot has reached the specified position. There is no delay if this argument is omitted. The delay is not synchronised with the movement.
[\Corr]	Data type: switch Correction data that has been written in a correction entry using the CorrWrite instruction is added to the path and the destination position if this argument is present.

Program execution

More information about axis-specific movements can be found in the explanations for instruction MoveLDO or MoveL.

The digital group output and the passed position number are set/reset in the middle of the corner path for fly-by points, as shown in the figure below.




```
SMoveLGO p101101,v1000,z30,tGripper,goTest,255;
```

en1200000795

Continues on next page

15.2.29 MT_MoveLGO – Linear robot movement and set group output in zone.
Continued

The specified I/O signal is set/reset in continuous and "forwards one step at a time" execution mode, but not in "backwards one step at a time" execution mode.

Note

The MT_MoveLGO instruction may never be used as the first movement instruction in a movement routine. MT_MoveL or MT_MoveJ with the \NoMove argument must always be used for this purpose.

Note

No digital group output processing takes place during backwards instruction by instruction execution or when searching for the first position, nor during backwards movement with "MT_MoveRoutine".

Tip

The digital group output is not set or reset during movement to the home position if the execution of trigger events (ExecTriggEvt) has been disabled in the system parameters (FALSE).

Syntax

```
MT_MoveJGO
['\'Conc ',']
[ActPos ':='] < expression (IN) of dnum> ','
[ToPoint ':='] < expression (IN) of robtarget > ','
[speed ':='] < expression (IN) of speeddata >
['\'T':=' < expression (IN) of num > ] ','
[zone ':='] < expression (IN) of zonedata >
['\'Inpos ':=' < expression (IN) of stoppointdata > ]','
[Tool ':='] < persistent (PERS) of tooldata >
['\'WObj ':=' < persistent (PERS) of wobjdata > ]
[Signal':=' ] < variable (VAR) of signalgo >
[Value':=' ] < expression (IN) of dionum >
['\'DODelay':=' < variable (IN) of num > ]
['\'Conc ]';'
```

Other information

Information about	See
`MoveL` – Linear robot movement.	*Technical Reference Manual – Instructions, Functions and Data Types* listed in the section *References on page 11*.
`MoveLDO` - Linear robot movement and setting of a digital output in the zone	*Technical Reference Manual – Instructions, Functions and Data Types* listed in the section *References on page 11*.

15.2.30 MT_MoveLSync – Linear movement and execution of a RAPID procedure

Usage

MT_MoveLSync (Move Linearly Synchronously) is used to move the tool centre point (TCP) linearly to a specified destination.

The position number assignment and the execution of the specified RAPID procedure takes place in the centre of the corner path.

If the TCP is to remain stationary, this instruction can also be used to reorient the tool.

This instruction can only be used in the Main task T_ROB1 or in motion tasks in the case of a MultiMove system.

The instruction basically corresponds to a MoveLSync with the addition of the position number.

Basic example

Example 1:

```
MT_MoveLSync 111001,p111001,vmax, z30, tGripper,"OpenGripper";
```

The TCP of the tGripper tool moves in a linear to the position programmed in the instruction p111001 and then saves 111001 as the current position. The OpenGripper procedure OpenGripper is processed in the middle of the corner path at p111001 .

Example 2:

```
MT_MoveLSync 111002,*,vmax,z30,tGripper,"MySync"\PStr:= "Hello";
```

The TCP moves linear to the position programmed in the instruction and calls up the procedure "MySync“ with the passing parameter "Hello".

Arguments

```
MT_MoveLSync
[\Conc] ActPos ToPoint speed [\T] zone [\Inpos]
Tool [\Wobj] ProcName [\PNum] [\PDnum] [\PStr] [\Corr]
```

`[\Conc]`
`Concurrent`

Data type: `switch`

The following instructions are executed whilst the robot is in motion. Further information can be obtained from the MoveL instruction.

`ActPos`

Data type: `dnum`

Contains the position number of the position to be moved to

`ToPoint`

Data type: `robtarget`

The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

`speed`

Data type: `speeddata`

The speed programmed for the movement. The speed data define the velocity of the TCP, of the tool reorientation and of external axes.

Continues on next page

15.2.30 MT_MoveLSync – Linear movement and execution of a RAPID procedure
Continued

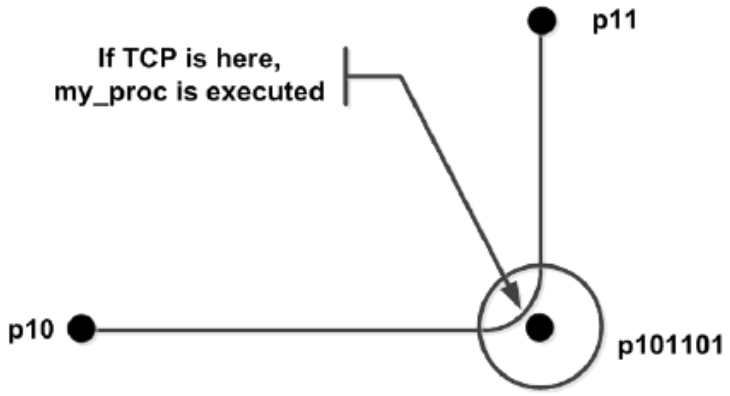
[\T] (Time)	Data type: num This argument is used to specify the time in seconds during which a movement of the manipulator and of the external axes should be executed. This value is then substituted for the corresponding speed data.
Zone	Data type: zonedata Zone data for the movement. Zone data describe the distance in which the axes must stand from the destination point before the next instruction is executed.
[\Inpos]	Data type: stoppointdata This argument is used to specify the convergence criteria for the position of the robot's TCP in the stop point. The stop point data substitutes the zone specified in the Zone parameter.
Tool	Data type: tooldata The tool in use when the robot moves. The tool centre point is moved to the specified destination point.
[\Wobj]	Data type: wobjdata The work object (tool coordinate system) to which the robot position in the instruction is related. This argument can be omitted. In this case the position relates to the world coordinate system.
ProcName	Data type: string Name of the RAPID procedure to be executed at the middle of the corner path in the destination point.
[\PNum]	Data type: num Numeric passing parameter to be passed to the calling RAPID procedure.
[\PDnum]	Data type: dnum Double numeric passing parameter to be passed to the calling RAPID procedure.
[\PStr]	Data type: string String to be passed to the calling RAPID procedure.
[\Corr]	Data type: switch Correction data that has been written in a correction entry using the CorrWrite instruction is added to the path and the destination position if this argument is present.

Continues on next page

Program execution

More information about linear movements can be found in the explanations for instruction `MoveLSync` or `MoveL`.

The specified RAPID procedure and the position number are processed when the TCP reaches the middle of the corner path at the destination point of the MT_MoveLSync instruction.




```
SMoveLSync p101101,v1000,z30,tGripper,"my_proc";
```

en1200000796

Execution of the specified RAPID procedure in different execution modes:

Execution mode	Execution of RAPID procedure
Continuously or Cycle	According to this description
Forward step	In the stop point
Backward step	Not at all
During backwards movement with MT_MoveRoutine	Not at all
Use of "fine" points	At the stop point

`MT_MoveLSync` is an encapsulation of the `TriggInt` and `TriggL` instructions. The procedure call is executed at TRAP level.

If the centre of the corner path at the destination point is reached during the slowdown after a program stop, the procedure is not called up (program execution is stopped). The procedure call is executed when the next program start takes place.

Note

The `MT_MoveLSync` instruction may never be used as the first movement instruction in a movement routine. `MT_MoveL` or `MT_MoveJ` with the `\NoMove` argument must always be used for this purpose.

Continues on next page

15.2.30 MT_MoveLSync – Linear movement and execution of a RAPID procedure
Continued

Note

The specified RAPID procedure is not processed during backwards instruction by instruction execution or when searching for the first position, nor during backwards movement with "MT_MoveRoutine".

Tip

The RAPID procedure is not executed during movement to the home position if the execution of trigger events (`ExecTriggEvt`) has been disabled in the system parameters (FALSE).

Use of the RAPID routine

Local routines

A routine that is declared locally in the module can be called up by specifying the module name, for example, "Module:Routine"

Example:

```
MT_MoveLSync 10,p10,v1000,z10,tGripper,"MyModule:MySync
MODULE MyModule
LOCAL PROC MySync()
…
ENDPROC
ENDMODULE
```

The procedure that is declared locally in the "MyModule" module ("MySync") will call up the centre of the corner path p10.

Use of parameters

A `num`, a `dnum` and/or a `string` parameter can also be passed to a RAPID routine.

During the creation of the calling RAPID procedure it must be ensured that it contains the required passing parameters.

If several parameters are used, the order (`num, dnum, string`) must be taken into consideration.

Example:

```
PROC mv10_11()
MT_MoveLSync 101101,*,v1000,z10,tGripper,"MySync1"\PNum:=1;
MT_MoveLSync 101102,*,v1000,z10,tGripper,"MySync2"
\PStr:="Test";
MT_MoveLSync 101103,*,v1000,z10,tGripper,"MySync3"
\PNum:=5\PDnum:=10\PStr:="Test";
ENDPROC
PROC MySync1(num Value)
Tpwrite "Sync 1: "\Num:=Value;
ENDPROC
PROC MySync2(string Value)
Tpwrite "Sync 2: "+Value;
ENDPROC
PROC MySync3(num Value1, dnum Value2, string Value3)
```

Continues on next page

```
Tpwrite "Sync 3: "+Valtostr(Value1)+ "/"+ Value3;
TEST Value2
CASE 10:
…
CASE 11:
…
ENDTEST
ENDPROC
```

Limitations

The MT_MoveLSync instruction cannot be used at interrupt level.

The specified RAPID procedure cannot be tested with stepwise execution.

When the robot reaches the centre of the corner path, the RAPID routine is usually executed after a delay of 20-30s, depending on the type pf movement being made at the time.

A change of execution mode from continuous or cyclic to forwards or backwards after a program stop results in an error. This error tells the user that the mode change can result in the RAPID procedure being omitted from the queue for execution on the path.

Syntax

```
MT_MoveLSync
['\'Conc ',']
[ActPos ':='] < expression (IN) of dnum> ','
[ToPoint ':='] < expression (IN) of robtarget > ','
[Speed ':=' ] < expression (IN) of speeddata >
['\'T':=' < expression (IN) of num > ] ','
[Zone ':=' ] < expression (IN) of zonedata >
['\'Inpos ':=' < expression (IN) of stoppointdata > ]','
[Tool ':='] < persistent (PERS) of tooldata >
['\'WObj ':=' < persistent (PERS) of wobjdata > ]
['\'PNum:='< expression (IN) of num >]','
['\'PDnum':='< expression (IN) of dnum >]
[ '\'PStr':='< expression (IN) of string >]
['\'Corr ]';'
```

Other information

Information about	See
MoveL – Linear robot movement.	*Technical Reference Manual – Instructions, Functions and Data Types* listed in the section *References on page 11*.
MoveLSync - Linear robot movement and execution of a RAPID procedure	*Technical Reference Manual – Instructions, Functions and Data Types* listed in the section *References on page 11*.
TriggInt - Definition of a position-dependent interrupt	*Technical Reference Manual – Instructions, Functions and Data Types* listed in the section *References on page 11*.

Continues on next page

15.2.30 MT_MoveLSync – Linear movement and execution of a RAPID procedure
Continued

Information about	See
`TriggL` - Linear robot movements with events.	*Technical Reference Manual – Instructions, Functions and Data Types* listed in the section *References on page 11*.

15.2.31 MT_MoveRoutine – Execute a movement routine at HomeRun

Usage

`MT_MoveRoutine` is used within the HomeRun strategy to execute the specified movement routine in forward or backward direction.

Basic example

```
TEST Position
CASE 100:
MT_MoveRoutine "mv100_999";
CASE 101:
MT_MoveRoutine "mv100_101"\Back;
CASE 401:
MT_MoveRoutine "mv401_405";
CASE 401400:
MT_MoveRoutine "mv400_401"\Back;
```

If robot has reached the position 100 the routine mv100_999 will be executed.

If robot was stopped in position 101, the routine mv100_101 will be executed in backward direction, so that the robot will move from position 101 to position 100.

If robot has reached the position 401, the routine mv401_405 will be tried to execute. If this routine does not exist, the routine mv401_405 or the routine mv401_405 will be executed.

If robot was stopped at an intermediate position between position 400 and 401, the routine mv400_401 will be tried to execute in backward direction.

Limitations

Note

When `MT_MoveRoutine` is being used, no move instructions other than the `MT_Move instructions` may be used within a movement routine.

Arguments

```
MT_MoveRoutine Routine [\ModName] [\Prefix] [\Index] | [\DIndex]
    [\Backw]
```

`Routine`
Data type: `string`
Name of the movement routine which should be executed.

`ModName`
Data type: `num`
Name of the module where the required movement routines are located. This is only needed if the movement routines are declared as LOCAL.

`\Prefix`
Data type: `string`
Type prefix. Only necessary for type-depending movement routines and only if the standard type prefix “T” is not used. Example of a type related declaration name: mv10_30_T137, if the prefix is “T” and the part type number of the current partdata declaration is 137.

Continues on next page

15.2.31 MT_MoveRoutine – Execute a movement routine at HomeRun
Continued

[\Index]	Data type: num Index to select movement routines type-dependently. The index is appended to the routine name without an underscore, using the standard type prefix as specified in the process parameters or the explicitly specified prefix. Only one of the parameters \Index or \DIndex may be used.
[\DIndex]	Data type: dnum Index to select movement routines type-dependently. The index is appended to the routine name with an underscore, using the standard type prefix or the explicitly specified prefix. Only one of the parameters \Index or \DIndex may be used.
[\Backw]	Data type: switch Execute the movement in backward direction.

Program execution

The specified routine name and the type code of the currently used partdata declaration will be used to build the name of the corresponding moving routine and call it dynamically.

The movement routines have the folling naming structure:

```
"mv"<START>"_"<TARGET> ["_"< [<PREFIX>] [<INDEX>]>]
```

Structure examples:

Routine name	Description
mv100_101 _T1	Moves the robot from position 100 to position 101. This routine depends on a part type with part type prefix "T" and the part type index "1"
mv100_101	Moves the robot from position 100 to position 101. This routine does not depend on a part type.

Syntax

```
HR_MoveRoutine

[Routine ´:=´ ] < expression (IN) of string >
[´\´ ModName ´:=´ < expression (IN) of string> ]
[´\´ Prefix ´:=´ < expression (IN) of string> ]
[´\´ Index ´:=´ < expression (IN) of num> ]
[´\´ DIndex ´:=´ < expression (IN) of dnum> ]
[´\´ Backw ]´;´
```

Related information

Information about	See
MT_MoveTo – Execute a robot movement routine	*MT_MoveTo – Dynamic execution of a movement routine on page 391*
Move backward while HomeRun	*MT_HomeRun – HomeRun Strategy on page 357*

15.2.32 MT_MoveTo – Dynamic execution of a movement routine

Usage

The procedure `MT_MoveTo` is used to move from the current robot position to the desired target position. For this, the robot forms a string using the saved start position and the target position that has been passed. This string represents the name of the movement routine that is to be called. The movement routine will be called dynamically and the target position will be saved as the new start position.

Programming

The way how `MT_MoveTo` tries to call a movement routine, depends on the availability of information as follows:

- Common module name in the process configuration in the section MT PART SETTINGS (please refer to *MT Part Settings on page 162*).
- Type prefix in the process configuration also in section “MT PART SETTINGS” (please refer to*MT Part Settings on page 162*).
- Type Code in currently used partdata declaration (please refer to *partdata – Part data on page 284*).

The following examples clarify the proceeding:

If a common module name is set in the process configuration under MT API positions, then MT_MoveTo will try to call a movement routine in the following order, until one has been found for execution:

1) Local type-depending routine in type-depending module

Example :

- Common module name in the proc.cfg is “Movement”
- Type prefix in the proc.cfg is “T”
- Type code of the currently executed partdata is “13”

Routine, which MT_MoveTo will try to call:

```
MODULE Movement_T13
…
LOCAL PROC mv10_20_T13()
…
ENDPROC
…
ENDMODULE
```

2) Local type-depending routine in type-independent module

Example:

- Common module name in the proc.cfg is “Movement”
- Type prefix in the proc.cfg is “T”
- Type code of the currently executed partdata is “13”

Routine, which MT_MoveTo will try to call:

```
MODULE Movement
…
LOCAL PROC mv10_20_T13()
```

Continues on next page

```
…
ENDPROC
…
ENDMODULE
```

3) Local type-independent routine in type-depending module

Example:

- Common module name in the proc.cfg is “Movement”
- Type prefix in the proc.cfg is “T”
- Type code of the currently executed partdata is “13”

Routine, which MT_MoveTo will try to call:

```
MODULE Movement_T13
…
LOCAL PROC mv10_20()
…
ENDPROC
…
ENDMODULE
```

4) Global type-depending routine somewhere in the program

Example:

- Type prefix in the proc.cfg is “T”
- Type code of the currently executed partdata is “13”

Routine, which MT_MoveTo will try to call:

```
PROC mv10_20_T13()
…
ENDPROC
```

5) Global type-independent routine somewhere in the program

Example:

Routine, which MT_MoveTo will try to call:

```
PROC mv10_20()
…
ENDPROC
```

If a common module name is not set in the process configuration under MT API positions, then MT_MoveTo will try to call a movement routine in the following order:

1) Global type-depending routine somewhere in the program

Example:

- Type prefix in the proc.cfg is “T”
- Type code of the currently executed partdata is “13”

Routine, which MT_MoveTo will try to call:

```
PROC mv10_20_T13()
…
ENDPROC
```

2) Global type-independent routine somewhere in the program

Example:

Continues on next page

Routine, which MT_MoveTo will try to call:

```
PROC mv10_20()
…
ENDPROC
```

If none of the above metionned conditions is fulfilled, then an error message will be output.

Basic example

```
!Startposition is 10 (where the robot has moved before)
!Type prefix in proc.cfg is default (= “T“)
!Common module name in the proc.cfg is “Movement”
!Type index of currently selected part is 8
…
!Call the movement routine mv10_20_T8 in module Movement_T8
MT_MoveTo 20;
…
…
MODULE Movement_T8
…
PROC mv10_20_T8()
…
ENDPROC
…
ENDMODULE
```

Arguments

```
MT_MoveTo
Target [\ModName] [\Prefix] [\Index] [\Dindex]
```

`Traget`	**Data type:** `num` The position number of the target position
`ModName`	**Data type:** `string` Name of the module, in which the movement routine is contained. This argument should be used only for routines that have been declared locally.
`Prefix`	**Data type:** `string` Possible type prefix for the Index or DIndex.
`Index`	**Data type:** `num` Type number that is to be appended to the movement routine as index, when using data type num.
`DIndex`	**Data type:** `dnum` Type number that is to be appended to the movement routine as index, when using data type dnum.

Syntax

```
MT_MoveTo
[ Target ’:=’ ] < expression (IN) of num >
[ ’\’ ModName ’:=’ < expression (IN) of string > ]
[ ’\’ Prefix ’:=’ < expression (IN) of string > ]
[ ’\’ Index ’:=’ < expression (IN) of num > ]
```

Continues on next page

15.2.32 MT_MoveTo – Dynamic execution of a movement routine
Continued

```
[ '\' DIndex ':=' < expression (IN) of dnum > ] ';'
```

More information

For information about	See
Setting the current position for using `MT_MoveTo`	*MT_SetActualPosition – Setting the current position for MT_MoveTo on page 410*
Reading the current position for using `MT_MoveTo`	*MT_GetActualPosition – Reading the start position for MT_MoveTo on page 462*

15.2.33 MT_PartCheck – Part controls on the gripper

Usage

`MT_PartCheck` is used to check the component control sensors on a gripper. If the sensor status does not correspond to the state which is to be checked, then, an error message is output after a waiting period.

The check can be queried, optionally, for "part controls busy" or "part controls free".

Upto 5 different component configurations can be assigned at the same time.

Basic examples

Example 1:

```
MT_PartCheck low, gpGRP1_BT2;
```

A check is conducted to see if all the part sensors of the part controls 2 of the gripper 1 are busy.

Example 2:

```
MT_PartCheck high, gpGRP1_BT1\Part2:= gpGRP1_BT2;
```

A check is conducted to see if all the part sensors of the part controls 1 and 2 of the gripper 1 are busy.

Arguments

```
MT_PartCheck Value Part1 [\Part2] [\Part3] [\Part4]
[\Part5] [\ErrorNo] [\Fault]
```

`Value`	Data type: `dionum` The desired value of the component control sensors (high or low or 1 or 0)
`Part1`	Data type: `grppart` Part control configuration of the first part which is to be monitored.
`[\Part2]`	Data type: `grppart` Part control configuration of the second part which is to be monitored.
`[\Part3]`	Data type: `grppart` Part control configuration of the third part which is to be monitored.
`[\Part4]`	Data type: `grppart` Part control configuration of the fourth part which is to be monitored.
`[\Part5]`	Data type: `grppart` Part control configuration of the fifth part which is to be monitored.
`[\ErrorNo]`	Data type: `num` Combined error domains and error number as positive integer, which can be used for display in the event of errors. The last four digits represent the error number, the digits preceding this represent the error domain.
`[\Fault]`	Data type: `switch` If the switch fault is set, a gripper related message will appear as a fault message, otherwise it will appear as a warning message.

Continues on next page

15.2.33 MT_PartCheck – Part controls on the gripper
Continued

Program execution

All the part sensors that are contained in the part configurations will be checked and an error message is issued if a sensor does not report the desired state.

The program will be continued only if all the part sensors give the expected response.

Only those signals that contain a valid signal name will be considered.

Restrictions

In the "Ghost mode", the component control sensors are not checked if the requestedalue for the sensors is high (1).

Syntax

```
MT_PartCheck
[Value] ´:=´ < expression (IN) of dionum> ]
[Part1] ´:=´ < expression (IN) of grpppart> ]
[´\Part2] ´:=´ < expression (IN) of grpppart > ]
[´\Part3] ´:=´ < expression (IN) of grpppart > ]
[´\Part4] ´:=´ < expression (IN) of grpppart > ]
[´\Part5] ´:=´ < expression (IN) of grpppart > ]
[´\ErrorNo] ´:=´ < expression (IN) of num> ]
[´\Fault]
´;´
```

More information

Information about	See
grppart	*grppart – Part control configuration on page 260*

15.2.34 MT_PartCheckType – Part controls on the gripper

Usage

`MT_PartCheckType` is used to check the component control sensors on a gripper. If the sensor status does not correspond to the state which is to be checked, then, an error message is output after a waiting period.

The check can be queried, optionally, for "part controls busy" or "part controls free". Up to 5 different component configurations can be assigned at the same time.

`MT_PartCheckType` provides mainly the same functionality as MT_PartCheck but considers part type specific type numbers and type prefixes as follows:

There might be different grippers for each part type in the production cell. The grippers might work differently, thus each gripper will need its own grppart declarations.

Instead of assigning the grppart directly as this is done with `MT_PartCheck`, a string is provided to `MT_PartCheckType` which represents the name of the grppart but without part type number and part type prefix.

`MT_PartCheckType` will internally complete the grppart name, depending on the current settings for the type prefix and the type number. Then the instruction will execute the appropriate type-depending grppart declaration.

Basic examples

Assuming, the current part type number is 6 and the standard part type prefix is “T”:

Example 1:

```
MT_PartCheck low, ”gpGRP_BT2”;
```

A check is conducted to see if all the part sensors of the part controls 2 of the gripper are busy (grppart gpGRP_BT2_T6).

Example 2:

```
MT_PartCheck
high,”gpGRP_BT1”\Part2:=”gpGRP_BT2”\Prefix:=”P”;
```

A check is conducted to see if all the part sensors of the part controls 1 and 2 of the gripper are busy (grppart gpGRP_BT1_P6 and gpGRP_BT2_P6).

Arguments

```
MT_PartCheckType Value Part1 [\Part2] [\Part3] [\Part4]
[\Part5] \ [Prefix] \[ErrorNo] \[Fault]
```

`Value`	Data type: `dionum` The desired value of the component control sensors (high or low or 1 or 0)
`Part1`	Data type: `string` Name of the part control configuration of the first part which is to be monitored, without part type prefix and without part type number.

Continues on next page

15.2.34 MT_PartCheckType – Part controls on the gripper
Continued

[\Part2]	Data type: string Name of the part control control configuration of the second part which is to be monitored, without part type prefix and without part type number.
[\Part3]	Data type: string Name of the part control control configuration of the third part which is to be monitored, without part type prefix and without part type number.
[\Part4]	Data type: string Name of the part control control configuration of the fourth part which is to be monitored, without part type prefix and without part type number.
[\Part5]	Data type: string Name of the part control control configuration of the fifth part which is to be monitored, without part type prefix and without part type number.
[\Prefix]	Data type: string Assigns another part type prefix apart from the default prefix.
[\ErrorNo]	Data type: num Combined error domains and error number as positive integer, which can be used for display in the event of errors. The last four digits represent the error number, the digits preceding this represent the error domain.
[\Fault]	Data type: switch If the switch fault is set, a gripper related message will appear as a fault message, otherwise it will appear as a warning message.

Program execution

All the part sensors that are contained in the part configurations will be checked and an error message is issued if a sensor does not report the desired state.

The program will be continued only if all the part sensors give the expected response. Only those signals that contain a valid signal name will be considered.

Restrictions

In the "Ghost mode", the component control sensors are not checked if the requested value for the sensors is high (1).

Syntax

```
MT_PartCheck
[Value] ´:=´ < expression (IN) of dionum> ]
[Part1] ´:=´ < expression (IN) of string> ]
[´\Part2] ´:=´ < expression (IN) of string > ]
[´\Part3] ´:=´ < expression (IN) of string > ]
[´\Part4] ´:=´ < expression (IN) of string > ]
[´\Part5] ´:=´ < expression (IN) of string > ]
[´\Prefix] ´:=´ < expression (IN) of string > ]
[´\ErrorNo] ´:=´ < expression (IN) of num> ]
[´\Fault]
´;´
```

Continues on next page

More information

Information about	See
`grppart`	*grppart – Part control configuration on page 260*

15.2.35 MT_ResetActiveStation – Set station symbol to "inactive"

Usage

`MT_ResetActiveStation` should be used to represent on the RWMT user interface that the robot is not serving any station at the moment.

A station here could mean a specific machine, a conveyor belt or a slide within the production cell.

The operator can thus recognize on the RWMT user interface where the robot is working at the moment:

Station ready, not served by robot: 

Station ready, robot serves the station: 

Basic example

```
LOCAL PERS stationdata DCM_Station:=
["DCM","DCM","Die casting machine","station-DCM.png",
"diDCMReady","diMouldClosed","","",TRUE,FALSE,1,1];

PROC UnloadDCM()
!Wait until DCM is ready for unloading
MT_WaitMsgDI diUnloadDCM,high,msgUnloadDCM;
!Robot serves the machine now
MT_SetActiveStation DCM_Station;
!Machine is being unloaded
…
…
!The robot has finished serving the machine
MT_ResetActiveStation;
…
ENDPROC
```

The robot waits for the release to move into the die casting machine. Once it gets this release, the program will set the symbol of this station to active on the RWMT user interface with the help of the instruction `MT_SetActiveStation`: 

After the pressure casting machine has been unloaded, the robot program will set the symbol to inactive, with the help of the instruction `MT_ResetActiveStation`:

Continues on next page

Program execution

The use of the instructions `MT_SetActiveStation` and `MT_ResetActiveStation` will influence only the station symbols on the RWMT user interface (GUI), but not the program run itself.

> **Note**
>
> If after serving a machine there is a next station which is marked as “served by robot“ by using `MT_SetActiveStation`, then it is not necessary to use `ResetActiveStation`.

Syntax

```
MT_ResetActiveStation ';'
```

More information

Information about	See
Data type `stationdata`	*stationdata – Definition of a station on page 297*
Setting up of stations	*Stations on page 44*
Instruction `MT_SetActiveStation`	*MT_SetActiveStation – Set station symbol to "active" on page 408*

15.2.36 MT_ResetFirstCycle – Declare first cycle as finished

Usage

`MT_ResetFirstCycle` resets the internal RWMT “first cycle” flag. This flag is automatically set to TRUE, when starting RWMT from the beginning (from “main”). It is reset automatically by RWMT after the first production loop ends and the program pointer leaves the first production loop and returns to the RWMT engine.

The purpose of `MT_ResetFirstCycle` is to reset the “first cycle” flag before having finished the first production loop.

This can be necessary for example, if the first cycle is left by using the error handler and through the error number `ERR_MT_ABORT`. If this cycle shall nevertheless be defined as finished normally, then `MT_ResetFirstCycle` should be used before leaving the first production loop.

Basic example

```
PROC MyProduction()
…
RAISE ERR_MT_ABORT;
…
ERROR
IF ERRNO = ERR_MT_ABORT MT_ResetFirstCycle;
RAISE;
ENDPROC
```

Program execution

`MT_ResetFirstCycle` resets the internal RWMT “first cycle flag”.

Arguments

```
MT_ResetFirstCycle ;
```

No arguments.

Syntax

```
MT_ResetFirstCycle ';'
```

More information

Information about	See
Requesting “first cycle” status	*MT_FirstCycle – Requesting first cycle status on page 461*

15.2.37 MT_SearchL – Linear search movement of robot

Usage

`MT_SearchL` (Search Linear) is used to search for a position with a linear movement of the Tool Centre Point (TCP). The robot monitors a digital input signal during the movement. If the signal adopts the required value, the robot reads off the current position immediately.

The position number is only assigned if a position has been found.

This instruction is typically suitable for the use of a surface detection sensor. The `MT_SearchL` instruction can be used to determine the outline of a workpiece, for example..

This instruction can only be used in the Main task T_ROB1 or in motion tasks in the case of a MultiMove system.

When search instructions are being used it is important to configure the I/O system so that there is an extremely short time between the setting of the physical signal and the arrival of the information about the setting in the system.

The instruction basically corresponds to the `SearchL` instruction with the addition of the position number.

Basic example

```
VAR robtarget pSearch;
MT_SearchL \Stop,11, diSensor, pSearch, p10, v100, tSensor;
```

The tSensor TCP is moved linearly to position `p10` . If the value of the `diSensor` signal changes to high, the position is saved in `pSearch`, the robot stops moving and position number 11 is saved as the current position

Arguments

```
SearchL [\Stop] | [\PStop] | [\SStop] | [\Sup]
ActPos Signal [\Flanks] [ \PosFlank ] [ \NegFlank ]
[ \HighLevel ] [ \LowLevel ]SearchPoint ToPoint speed
[\V] | [\T] Tool [\WObj] [\Corr]
```

`[\Stop] Stiff Stop`

Data type: `switch`

The robot movement stops as quickly as possible without keeping the TCP on the path (hard stop) if the value of the search signal changes to active.

However, before the stop the robot travels back a short way and is not moved back to the position that was searched for, i.e. the position where the signal change took place.

`[\PStop] Path Stop`

Data type: `switch`

The robot movement stops as quickly as possible and keeps the TCP on the path (soft stop) if the value of the search signal changes to active. However, before the stop the robot travels back a certain way and is not moved back to the position that was searched for, i.e. the position where the signal change took place.

Continues on next page

15.2.37 MT_SearchL – Linear search movement of robot
Continued

[\SStop] Soft Stop	Data type: switch The robot movement stops as quickly as possible and keeps the TCP close to or on the path (soft stop) if the value of the search signal changes to active. However, before the stop the robot only travels back a short way and is not moved back to the position that was searched for, i.e. the position where the signal change took place. SStop is faster than PStop. However, if the robot exceeds a speed of 100 mm/s it stops tangentially in the direction of movement, which results in less deviation from the path.
[\Sup] Supervision	Data type: switch The search instruction reacts sensitively to signal activation during a complete movement (flying search), i.e. even after the first signal change has been indicated. If several correspondences are found during a search, a rectifiable error is generated in ToPoint with the robot. If the \Stop, \PStop, \SStop or \Sup argument is omitted (no switch used): • The movement (flying search) at the position specified in the ToPoint argument is continued (as with the \Sup argument) • An error is signalled for no hits, but no error is signalled for several hits (the first hit is returned as the SearchPoint).
ActPos	Data type: dnum Contains the position number of the position to be moved to.
Signal	Data type: signaldi Name of signal to be monitored.
[\Flanks]	Data type: switch The positive and negative flank of the signal applies for a search result.
[\PosFlank]	Data type:switch The positive edge of the signal is valid for a search hit.
[\NegFlank]	Data type:switch The negative edge of the signal is valid for a search hit.
[\HighLevel]	Data type:switch The same functionality as if not using \Flanks switch. The positive edge of the signal is valid for a search hit and a signal supervision will be activated at the beginning of a search process. This means that if the signal has the positive value already at the beginning of a search process or the communication with the signal is lost then the robot movement is stopped as quickly as possible, while keeping the TCP on the path (soft stop). A user recovery error (ERR_SIGSUPSEARCH) will be generated and can be handled in the error handler.
[\LowLevel]	The negative edge of the signal is valid for a search hit and a signal supervision will be activated at the beginning of a search process. This means that if the signal has value 0 already at the beginning of a search process or the communication with the signal is lost then the robot movement is stopped as quickly as possible, while keeping the TCP on the path (soft stop). A user recovery error (ERR_SIGSUPSEARCH) will be generated and can be handled in the error handler.

Continues on next page

`SearchPoint`	Data type: `robtarget` The position of the TCP and the external axes if the search signal has been triggered. The position is specified in the outermost coordinate system, taking the specified tool, workpiece and active ProgDisp/ExtOffs coordinate system into consideration.
`ToPoint`	Data type: `robtarget` The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an *). SearchL always uses as stop point as zone data for the destination.
`speed`	Data type: `speeddata` The speed programmed for the movement. The speed data define the velocity of the TCP, of the tool reorientation and of external axes.
`[\V] Speed`	Data type: `num` This argument is used to specify the velocity of the TCP in mm/s directly in the instruction. It is then substituted for the corresponding velocity specified in the speed data.
`[\T] Time`	Data type: `num` This argument specifies the total time in seconds for which the robot moves. It then replaces the corresponding speed data.
`Tool`	Data type: `tooldata` The tool in use when the robot moves. The tool centre point is moved to the specified destination point.
`[\WObj] Work Object`	Data type: `wobjdata` The work object (coordinate system) to which the robot position in the instruction relates. This argument can be omitted; in this case the position relates to the world coordinate system. However, if a stationary TCP or coordinated external axes are used, this argument must be specified so that a linear path in relation to the workpiece is possible.
`[\Corr] Correction`	Data type: `switch` Correction data that has been written in a correction entry using the CorrWrite instruction is added to the path and the destination position if this argument is present.

Program execution

More information about linear search movements can be found in the explanations for instruction `MT_SearchL`.

The movement always ends with a stop point, i.e. the robot stops at the destination point. With a flying search, i.e. the `\Sup` argument has been used or no switch has been specified, the robot always continues moving to the programmed destination point. If a search is carried out using the `\Stop, \PStop` or `\SStop` switch, the robot movement stops with the first search result.

Continues on next page

15.2.37 MT_SearchL – Linear search movement of robot
Continued

The SearchL instruction saves the position of the TCP when the value of the digital signal changes, as required.

> **Note**
>
> The MT_SearchL instruction may never be used as the first movement instruction in a movement routine. MT_MoveX must always be used with the \NoMove argument for this purpose.

Limitation

If backwards instruction by instruction execution is taking place, the robot moves directly to the programmed destination position.

> **Note**
>
> If a virtual controller (for example, RobotStudio) is being used, the search movement is not performed. The robot moves directly to the programmed destination position instead.

Backwards execution of a movement routine (see *MT_MoveRoutine – Execute a movement routine at HomeRun on page 389*) that uses the MT_SearchL instruction is not possible.

Syntax

```
MT_SearchL
['\'Stop',']|['\'PStop',']|['\'SStop',']|['\'Sup',']
[ActPos ':='] < expression (IN) of dnum> ','
[Signal ':=' ] < variable (VAR) of signaldi >
['\'Flanks]',’
['\'PosFlank]',’
['\'NegFlank]',’
['\'HighLevel]',’
['\'LowLevel]',’
[SearchPoint':='] < var or pers (INOUT) of robtarget>',’
[ToPoint' :=' ] < expression (IN) of robtarget >
[speed ':=' ] < expression (IN) of speeddata >
['\'V ':=' < expression (IN) of num > ] |
['\'T ':=' < expression (IN) of num > ] ',’
[Tool ´:=´ ] < persistent (PERS) of tooldata >
['\'WObj' :=' < persistent (PERS) of wobjdata > ]
['\'Corr ]';
```

Other information

Information about	See
SearchL - Linear search movement of robot	*Technical Reference Manual – Instructions, Functions and Data Types* listed in the section *References on page 11*.

Continues on next page

Information about	See
`MoveL` – Linear robot movement.	*Technical Reference Manual – Instructions, Functions and Data Types* listed in the section *References on page 11*.

15.2.38 MT_SetActiveStation – Set station symbol to "active"

Usage

`MT_SetActiveStation` should be used at appropriate places in the application program to show on the RWMT user interface which station the robot is currently serving.

A station here could mean a specific machine, a conveyor belt or a slide within the production cell.

Station ready, not served by robot: 

Station ready, robot serves the station: 

Basic example

```
LOCAL PERS stationdata DCGM_Station:=
["DCM","DCM","Die casting machine","station-DCM.png",
"diDCMReady","diMouldClosed","","",TRUE,FALSE,1,.1];

PROC UnloadDCM()
!Wait until DCM is ready for unloading
MT_WaitMsgDI diUnloadDCM,high,msgUnloadDCM;
!Robot serves the machine now
MT_SetActiveStation DCGM_Station;
!Machine is being unloaded
…
…
!The robot has finished serving the machine
MT_ResetActiveStation;
…
ENDPROC
```

The robot waits for the release to move into the die casting machine. Once it gets this release, the program will set the symbol of this station to active on the RWMT user interface with the help of the instruction `MT_SetActiveStation`: 

After the pressure casting machine has been unloaded, the robot program will set the symbol to inactive, with the help of the instruction `MT_ResetActiveStation`:

Continues on next page

Program execution

The use of the instructions `MT_SetActiveStation` and `MT_ResetActiveStation` will influence only the station symbols on the RWMT user interface (GUI), but not the program run itself.

> **Note**
>
> If after serving a machine there is a next station which is marked as “served by robot“ by using `MT_SetActiveStation`, then it is not necessary to use `ResetActiveStation`.

Arguments

```
MT_SetActiveStation station
```

`station`	**Data type:** `stationdata` The station whose station pictogram is to be set to "active".

Program execution

The use of instructions `MT_SetActiveStation` and `MT_ResetActiveStation` will only influence the station symbols on the RWMT user interface (GUI), but not the program run itself.

Syntax

```
MT_SetActiveStation
[ station ':=' ] < expression (IN) of stationdata > ';'
```

More information

Information about	See
Data type stationdata	*stationdata – Definition of a station on page 297*
Setting up of stations	*Stations on page 44*
Instruction `MT_ResetActiveStation`	*MT_ResetActiveStation – Set station symbol to "inactive" on page 400*

15.2.39 MT_SetActualPosition – Setting the current position for MT_MoveTo

Usage

With the help of the instruction `MT_SetActualPosition`, the start position, which is used by the instruction `MT_MoveTo`, can be initialized.

It might be necessary to define the start position once again, for example, after having moved the robot without the `MT_Move` instruction.

Basic example

```
!Set 999 (Home) as the start position
MT_SetActualPosition 999;
…
!Call the global movement routine mv999_20
MT_MoveTo 20;
!Call the global movement routine mv20_30
MT_MoveTo 30;
…
```

Arguments

```
MT_SetActualPosition ActPos
```

`ActPos`	**Data type:** `num` The new start position for using the `MT_MoveTo` instruction.

Syntax

```
MT_SetActualPosition
[ ActPos ':=' ] < expression (IN) of num > ';'
```

More information

Information about	See
Call movement routines with `MT_MoveTo` dynamically	*MT_MoveTo – Dynamic execution of a movement routine on page 391*
Reading the current position for using `MT_MoveTo`	*MT_GetActualPosition – Reading the start position for MT_MoveTo on page 462*

15.2.40 MT_SetEndOfCycle – Set the "Halt after end of cycle" state

Usage

`MT_SetEndOfCycle` allows the application program to induce the end of cycle by setting the request or by setting the end of cycle directely.

Normally the “halt after end of cycle” is induced either through the graphic user interface or by means of external signals. If the application program shall also be able to induce the halt after end of cycle, the instruction `MT_SetEndOfCycle` can be used.

Basic example

```
PROC Production()
!If the nmachine is not ready for production
IF diMachineReady=0 THEN
!force end of cycle
MT_SetEndOfCycle\Direct;
…
ENDPROC
```

If a machine is not ready for production, a direct halt after end of cycle can be forced.

Arguments

```
MT_SetEndOfCycle [\Direct]
```

`[\Direct]` Data type: `switch`
The new start position for using the MT_MoveTo-instruction.

Directly set the halt after end of cycle without the need to acknowledge it through the instruction `MT_EndOfCycleAck`.

Program execution

`MT_SetEndOfCycle` without the switch `\Direct` sets the request for “halt after end of cycle”. This request must be confirmed by using the instruction `MT_EndOfCycleAck`.

If the switch `\Direct` is used, the “halt after end of cycle” is set directly and no confirmation is needed.

Syntax

```
MT_SetEndOfCycle
['\'Direct] ';'
```

More information

Information about	See
Query if the request "Halt after end of cycle" is present	*MT_EndOfCycleReq – Recognizing the request "Halt after end of cycle" on page 459*
Confirming the “Halt after end of cycle”	*MT_EndOfCycleAck – Acknowledge the request "Halt after end of cycle" on page 317*

Continues on next page

15.2.40 MT_SetEndOfCycle – Set the "Halt after end of cycle" state
Continued

Information about	See
Query if the "Halt after end of cycle" request has been acknowledged already	*MT_EndOfCycleOk – Check if "Halt after end of cycle" was acknowledged on page 457*

15.2.41 MT_ShowMessage – Show message on the RWMT user interface

Usage

With the help of `MT_ShowMessage`, a program message can be output on the RWMT user interface.

This is done with the help of the data type msgdata, which has other additional information apart from the actual message text.

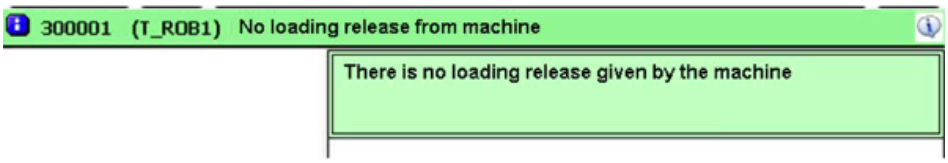
By outputting this message on the user interface (GUI), it is possible to prevent it from being overlapped by other message windows. Hence, this instruction is recommended for all those cases where an acknowledgement is not necessary at the programming device.

The instruction `MT_ShowMessage` is also available for background tasks.

Basic example

```
const msgdata msgLoadMachine:=
[30,1,0,"No loading release from machine","There is no loading
     release given","by the machine","","","",1,""];
!Show message that the machine is not ready for loading
MT_ShowMessage msgLoadMachine;
…
…
!Delete message
MT_ClearMessage;
```

The program will output on the RWMT user interface the message to the effect that the machine is not ready for loading.



en1300000272

The message will be deleted by executing the MT_ClearMessage instruction.

Arguments

```
MT_ShowMessage [msg] [\Ack]
```

`[msg]`	Data type: `msgdata` The message that has to be output, along with the message attributes.
`[\Ack]`	Data type: `switch` Setting this switch will cause the graphical user interface to display an OK button when showing a message. This OK button can be used by the operator to confirm and delete the message.

Continues on next page

15.2.41 MT_ShowMessage – Show message on the RWMT user interface
Continued

Program execution

When using `MT_ShowMessage`, a message is shown in the graphical user interface, until a new message is shown by `MT_ShowMessage` or the current message is deleted through the instruction `MT_ClearMessage`.

If the argument of type msgdata contains an error icon then an error number is sent to an external PLC, if this has been parametrized in the process configuration (please refer to *MT API Commands on page 154*).

Syntax

```
MT_ShowMessage
[ msg ':=' ] < expression (IN) of msgdata >
[ '\' Ack ]
';'
```

More information

Information about	See
Data type msgdata	*msgdata – Message declaration on page 280*
Deleting a message in the RWMT GUI	*MT_ClearMessage – Delete message on the RWMT user interface on page 309*
Showing a simple text message	*MT_ShowText – Delete single line message on the RWMT user interface on page 415*

15.2.42 MT_ShowText – Delete single line message on the RWMT user interface

Usage

With the help of `MT_ShowText`, a single line program message can be output on the RWMT user interface.

This is done with the help of the data type string.

By outputting a message on the user interface (GUI), it is possible to prevent this from being overlapped by other message windows. Hence, this instruction is recommended for all those cases where an acknowledgement is not necessary at the programming device.

Using `MT_ShowText erfolgt` instead of `MT_ShowMessage` for simple, single row messages. Unlike the `MT_ShowMessage`, there is no need to declare any message of the data type msgdata in advance.

The instruction `MT_ShowText` is also available for background tasks.

Basic example

```
!Show message that machine is not ready for loading
IF diLoadRelease=low THEN
MT_ShowText “No loading release from machine.“;
ENDIF
WaitUntil diLoadingRelease=high;
!Delete message
MT_ClearMessage;
```

The program will output on the RWMT user interface the message to the effect that the machine is not ready for loading. The message is cleared as soon as the release is given.

en1300000273

Arguments

```
MT_ShowText [\Warning] [|Fault] [\Number] str [\Ack]
```

`[\Warning]` — Data type: `switch`

If this switch is not set, then the text that is to be output appears as a warning message which is highlighted in yellow. If none of the switches `[\Warning]` or `[\Fault]` is set, then the text that is to be output will appear as information that is highlighted green.

`[\Fault]` — Data type: `switch`

If this switch is set, then the text that is to be output appears as an error message which is highlighted in red. If none of the switches `[\Warning]` or `[|Fault]` is set, then the text that is to be output will appear as information that is highlighted green.

Continues on next page

15.2.42 MT_ShowText – Delete single line message on the RWMT user interface
Continued

[\Number]	Data type: num A positive integer can be passed as a combined error domain and error number; in this case, this will be displayed along with the message. Here, the last 4 digits represent the error number and the digits preceding it represent the error domain. Example: 112345 => Error domain 11, Error number 2345 If the switch [\Fault] is set, then an error number is sent to an external PLC, if this has been parametrized in the process configuration (please refer to *MT API Commands on page 154*).
str	Data type: msgdata The message text that is to be output.
[\Ack]	Data type: switch Setting this switch will cause the graphical user interface to display an OK button when showing a message. This OK button can be used by the operator to confirm and delete the message.

Program execution

By using the instruction MT_ShowText a message will be displayed on the RWMT user interface until a new message is output, or the message will be deleted again with the help of the instruction MT_ClearMessage).

Syntax

```
MT_ShowText
[ '\' Warning]
[ '\' Fault ]
[ '\' Number ':=' ] < expression (IN) of num > ','
[ str ':=' ] < expression (IN) of text> ';'
[ '\' Ack ]
```

More information

Information about	See
Displaying a message on the RWMT user interface	*MT_ShowMessage – Show message on the RWMT user interface on page 413*
Deleting a message on the RWMT user interface	*MT_ClearMessage – Delete message on the RWMT user interface on page 309*

15.2.43 MT_ShowTPSViewRWMT – Open the RWMT graphic user interface

Usage

`MT_ShowTPSViewRWMT` opens the graphic user interface of RWMT.It can be used to ensure, that the user interface is opened automatically through the RAPID program, if it has been closed by the operator before.

Basic example

```
!Show message that machine is not ready for loading
MT_ShowTPSViewRWMT;
```

Arguments

`MT_ShowTPSViewRWMT`

No arguments

Program execution

If the RWMT user interface is not open, it will be started when executing the instruction `MT_ShowTPSViewRWMT`.

Syntax

```
MT_ShowTPSViewRWMT ';'
```

15.2.44 MT_SpeedUpdate – Adapting the speed

Usage

`MT_SpeedUpdate` is used in the application program to modify the programmed robot speed before executing a movement instruction, for example, in the event of `SafeMove` applications.

The `MT_SpeedUpdate` routine is only used if the "SpeedUpdate" parameter has been set to TRUE in the system parameters.

This routine has to be provided by the programmer/integrator.

Basic example

```
PROC MT_SpeedUpdate(INOUT speeddata speed)
const speeddata vSafety := [ 200, 15, 200, 15 ];

!SafeMove is not synchronised
IF diPSC1CSS=0 and speed.v_tcp>200 speed:=vSafety;
!
ENDPROC
```

If SafeMove™ is not synchronised and the programmed speed is greater than 200 mm/s, the TCP speed for the next movement is set to 200 mm/s and the reorientation speed is limited to 15°/s.

Arguments

`MT_SpeedUpdate speed`

`speed`	Data type: `speeddata` Speed for next movement instruction.

Program execution

If the speed update has been set in the system parameters, the `MT_SpeedUpdate` routine is always called with the programmed speed as the parameter before executing a movement instruction.

This speed information can be overwritten depending on external conditions, meaning that the robot does not perform the next movement with the programmed speed, but with the overwritten speed.

Tip

For example, this functionality can be used to reduce the speed if a non-synchronised SafeMove robot is being used.

Syntax

```
MT_SpeedUpdate
[speed ':=' ] < expression (INOUT) of speeddata> ';'
```

Continues on next page

Other information

Information about	See
`speeddata` - speed information	*Technical Reference Manual – Instructions, Functions and Data Types* listed in the section *References on page 11*.

15.2.45 MT_StartCycleTimer – Start recording the cycle time

Usage

`MT_StartCycleTimer` is used to start the recording of the cycle time in the production and to visualize this on the RWMT user interface as a progressive cycle time.

In this way, the time required for the current cycle can be recorded.

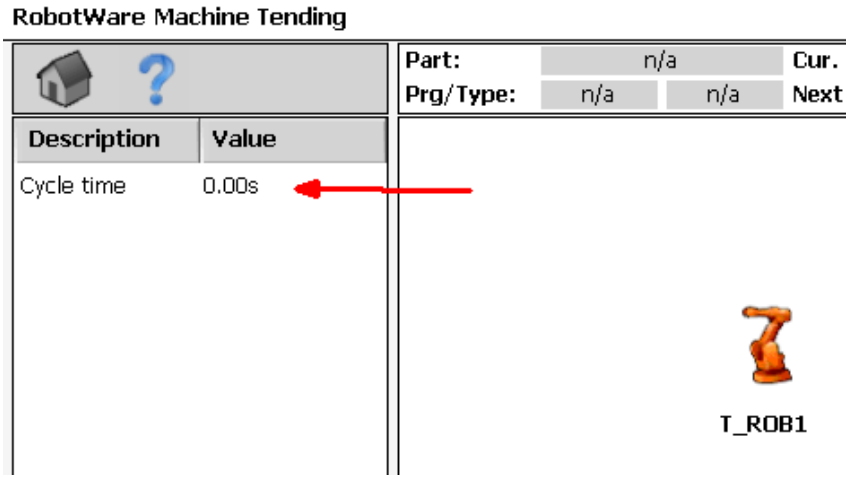
Basic example

```
…
!Start cycle timer
MT_StartCycleTimer;
!Unload machine
UnloadMachine;
!Feed out the part
LoadConveyor;
!Stop cycle timer
MT_StopCycleTimer;
…
```

At the start of the current production cycle, the measurement of the cycle time is started.

After this, the cycle is executed (unloading the machine, ejection).

Finally, the cycle time recording is stopped and the cycle time that has been determined will be displayed in seconds on the RWMT screen.



en1300000156

Program execution

By using the instruction `MT_StartCycleTimer`, the measurement of the cycle time is started.

Syntax

```
MT_StartCycleTimer ';'
```

Continues on next page

More information

Information about	See
Stopping the cycle time recording	*MT_StopCycleTimer – Stop recording the cycle time on page 422*

15.2.46 MT_StopCycleTimer – Stop recording the cycle time

Usage

With the help of `MT_StopCycleTimer`, the recording of the cycle time in the production will be stopped and this will be visualized on the RWMT user interface as the time that has been determined.

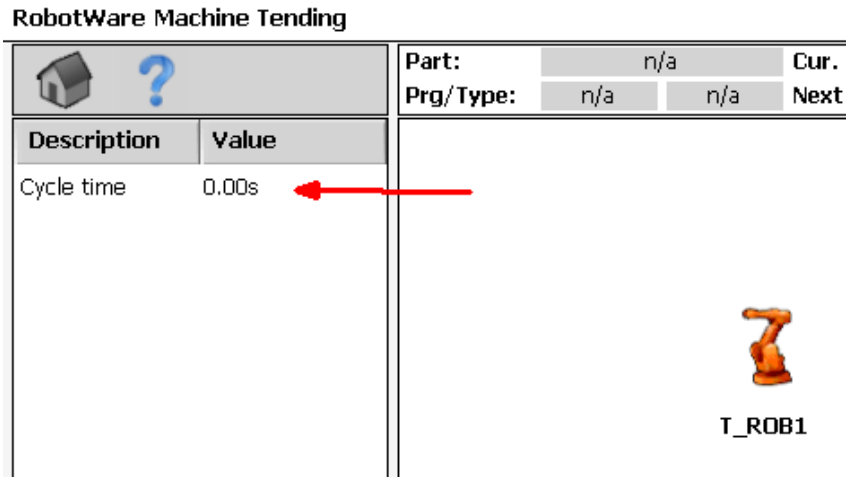
Basic example

```
…
!Start cycle timer
MT_StartCycleTimer;
!Unload machine
UnloadMachine;
!Feed out the part
LoadConveyor;
!Stop cycle timer
MT_StopCycleTimer;
…
```

At the start of the current production cycle, the measurement of the cycle time is started.

After this, the cycle is executed (unloading the machine, ejection).

Finally, the cycle time recording is stopped and the cycle time that has been determined will be displayed in seconds on the RWMT screen.



en1300000156

Program execution

By using the instruction `MT_StopCycleTimer`, the measurement of the cycle time is stopped.

Syntax

```
MT_StopCycleTimer ';'
```

Continues on next page

More information

Information about	See
Starting the cycle time recording	*MT_StartCycleTimer – Start recording the cycle time on page 420*

15.2.47 MT_ToolCheckL – Checking a tool

Usage

`MT_ToolCheckL` is used to move to a tool testing position in a linear manner and to determine byvisual inspection if the tool is present at the test point.

If the robot is not at the test point, then the system operator ca move the robot to the test position with the help of the controlling lever and get the TCP computed afresh with the help of the shifted position.

Basic example

```
PROC CheckTool1()
MoveJ p999, v500, z50,tTool1;
MoveJ […],v500,z50, tTool1;
MoveL […], v100,z10, tTool1;
ToolCheckL pTool1CheckPos, v50, tTool1;
MoveL […], v50,fine, tTool1;
MoveL […], v100,z10, tTool1;
MoveJ […],v500,z50, tTool1;
MoveJ p999, v500, z50, tTool1;
ENDPROC
```

The tool test position of the tool which belongs to the tooldata `tTool1` is approached and the dialog for the visual inspection will be displayed.

Arguments

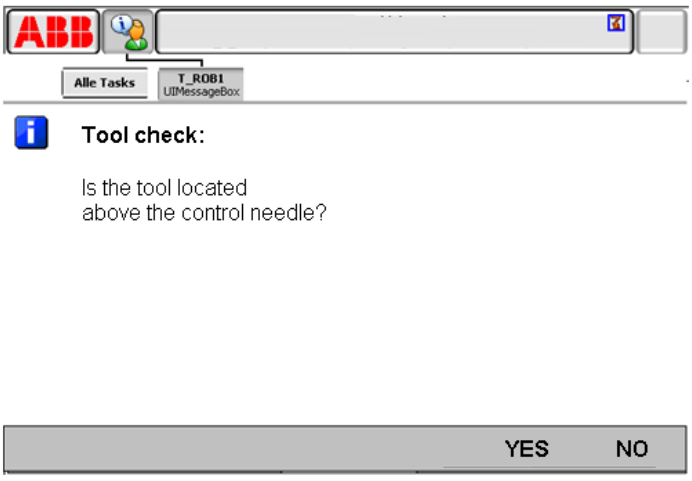
```
MT_ToolCheckL ToPoint speed Tool [\Limit]
```

`ToPoint`	Data type: `robtarget` The target position of the robot and the external axis.
`Speed`	Data type: `speeddata` The speed data that are applicable for movements. Speed data define the speed of the working point of the tool, the reorientation of the tool and the external axes.
`Tool`	Data type: `tooldata` The tool that is used during the movement. The tool working point (TCP) is the point moves at the specified target position.
`[\Limit]`	Data type: `num` Permitted deviation from the original tool data. If this parameter not used, then the max. deviation is 10 mm.

Continues on next page

Program execution

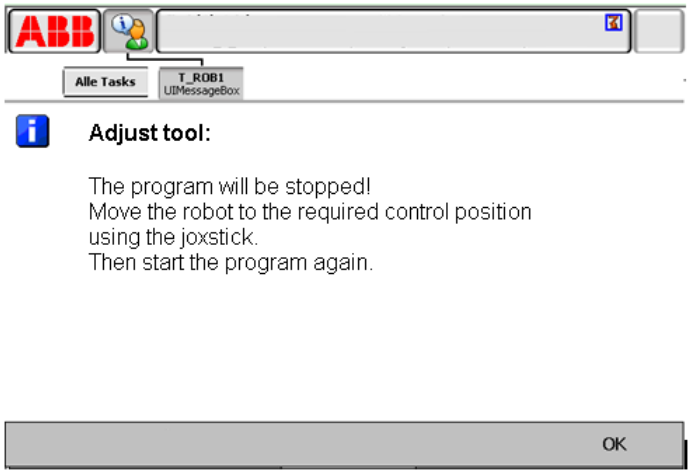
The robot moves to the tool test position in a linear manner, as a "fine" point and the following dialog will be displayed:



en1300000157

If the "Yes" key is pressed then the robot will move back to the basic setting.

If the "No" key is pressed, then the following dialog will be output, and then the program will be stopped.



en1300000158

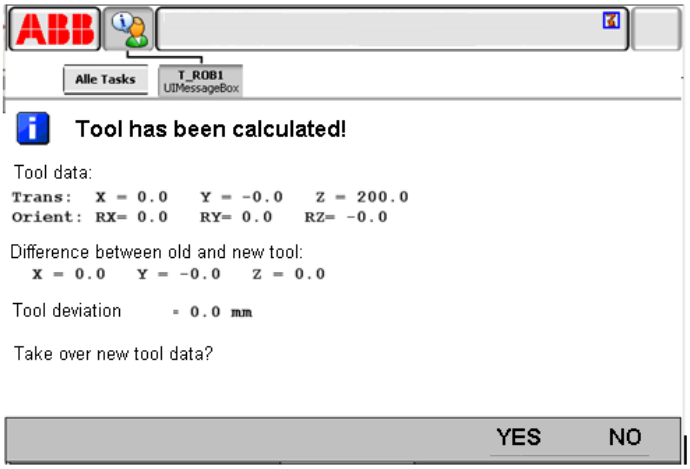
As soon as the program is stopped, robot must be moved with the help of the controlling lever to the desired test position and the program should be started again.

The tool declaration that has been passed will be computed afresh with the help of the shift in position and the test position will be approached with the corrected tool data.

Continues on next page

15.2.47 MT_ToolCheckL – Checking a tool
Continued

After this, the dialog with the result of the calculation is displayed:



en1300000159

If the "Yes" button is pressed, then the tool data that has been calculated will be imported for the tool.

If the "No" button is pressed, then the correction can be repeated or the original tool data can be retained as it is and the robot will move back to the basic setting.

Note

If the instruction `MT_ToolCheckL` is called for the first time with a tool, then its tool data will be saved in a file ("*HOME:/<tool>.wz*).

This data will be used as the basis for the maximum permitted displacement. By deleting the file, the basic tool definition can be replaced.

Syntax

```
MT_ToolCheckL
[ ToPoint ':=' ] < expression (IN) of robtarget > ','
[ Speed ':=' ] < expression (IN) of speeddata >
[ Tool ':=' ] < persistent (PERS) of tooldata >
[ '\' Limit ':=' ] < expression (IN) of num > ]';'
```

15.2.48 MT_TriggJ – Axis-wise robot movements with events

Usage

`MT_TriggJ (TriggJoint)` is used to set output signals and/or interrupt routines at approximately defined positions whilst the robot moves quickly from one point to another, whereby this movement does not have to be in a straight line. This position number assignment takes place in the middle of the corner path.

One or more (maximum of 7) events can be defined using the `TriggIO, TriggEquip, TriggInt, TriggCheckIO, TriggSpeed or TriggRampAO` instruction. Then reference can be made to these definitions with the TriggJ instruction.

This instruction can only be used in the Main task `T_ROB1` or in motion tasks in the case of a MultiMove system.

The instruction basically corresponds to a `TriggJ` with some additions.

Basic example

```
VAR triggdata trGlueing_On;
VAR triggdata trGlueing_Off;

PROC InitTrigger()
TriggIO trGlueing_On, 10 \Start \DOp:=doGlueing, 1;
TriggIO trGlueing_Off, 10 \Start \DOp:= doGlueing, 0;
ENDPROC


PROC mv10_11()
MT_MoveJ 10, p10, v1000, z30, tGripper\NoMove;
MT_TriggJ 101101,p101101,v500,trGluing_On,z10,tGripper;
...
MT_TriggL 101105,p101105,v500,trGluing_Off,z10,tGripper;
…
ENDPROC
```

The digital output signal `doGlueing` is set if the TCP of the robot is 10 mm in front of point p101101 . 101101 is then saved as the current position.

The digital output signal `doGlueing` is reset if the TCP of the robot is 10 mm in front of point p101105 . 101105 is then saved as the current position.

Arguments

```
MT_TriggJ [\Conc] ActPos ToPoint speed [\T] Trigg1 [\T2] [\T3]
[\T4] [\T5] [\T6] [\T7] zone [\Inpos] Tool [\WObj]
```

`[\Conc]` `Concurrent`	Data type: `switch` The following instructions are executed whilst the robot is in motion. Further information can be obtained from the `MoveL` instruction.
`Actpos`	Data type: `dnum` Contains the position designation of the position to be moved to.

Continues on next page

15.2.48 MT_TriggJ – Axis-wise robot movements with events
Continued

`ToPoint`	Data type: `robtarget` The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).
`Speed`	Data type: `speeddata` The speed programmed for the movement. The speed data define the velocity of the TCP, of the tool reorientation and of external axes.
`[\T] (Time)`	Data type: `num` This argument is used to specify the time in seconds during which a movement of the manipulator and of the external axes should be executed. This value is then substituted for the corresponding speed data.
`Trigg1`	Data type: `triggdata` A trigger condition-related and trigger activity-related variable that has been defined previously in the program using the `TriggIO`, `TriggEquip`, `TriggInt`, `TriggSpeed`, `TriggCheckIO` or `TriggRampAO` instructions.
`[\T2] (Trigg2)`	Data type: `triggdata` A trigger condition-related and trigger activity-related variable that has been defined previously in the program using the `TriggIO`, `TriggEquip`, `TriggInt`, `TriggSpeed`, `TriggCheckIO` or `TriggRampAO` instructions.
`[\T3 ] (Trigg3)`	Data type: `triggdata` A trigger condition-related and trigger activity-related variable that has been defined previously in the program using the `TriggIO`, `TriggEquip`, `TriggInt`, `TriggSpeed`, `TriggCheckIO` or `TriggRampAO` instructions.
`[\T4 ] (Trigg4)`	Data type: `triggdata` A trigger condition-related and trigger activity-related variable that has been defined previously in the program using the `TriggIO`, `TriggEquip`, `TriggInt`, `TriggSpeed`, `TriggCheckIO` or `TriggRampAO` instructions.
`[\T5 ] (Trigg5)`	Data type: `triggdata` A trigger condition-related and trigger activity-related variable that has been defined previously in the program using the `TriggIO`, `TriggEquip`, `TriggInt`, `TriggSpeed`, `TriggCheckIO` or `TriggRampAO` instructions.
`[\T6 ] (Trigg6)`	Data type: `triggdata` A trigger condition-related and trigger activity-related variable that has been defined previously in the program using the `TriggIO`, `TriggEquip`, `TriggInt`, `TriggSpeed`, `TriggCheckIO` or `TriggRampAO` instructions.
`[\T7 ] (Trigg7)`	Data type: `triggdata` A trigger condition-related and trigger activity-related variable that has been defined previously in the program using the `TriggIO`, `TriggEquip`, `TriggInt`, `TriggSpeed`, `TriggCheckIO` or `TriggRampAO` instructions.
`Zone`	Data type: `zonedata` Zone data for the movement. Zone data describe the distance in which the axes must stand from the destination point before the next instruction is executed.

Continues on next page

[\Inpos]	Data type: stoppointdata This argument is used to specify the convergence criteria for the position of the robot's TCP in the stop point. The stop point data substitutes the zone specified in the Zone parameter.
Tool	Data type: tooldata The tool in use when the robot moves. The tool centre point is moved to the programmed destination point.
[\Wobj]	Data type: wobjdata The work object (tool coordinate system) to which the robot position in the instruction is related. This argument can be omitted. In this case the position relates to the world coordinate system.

Program execution

More information about axis-specific movements and trigger functions can be found in the explanations for instruction MoveJ and TriggJ.

The passed position number is saved as the current robot position in the middle of the corner path of the destination position.

Note

The MT_TriggJ instruction may never be used as the first movement instruction in a movement routine. MT_MoveL or MT_MoveJ with the \NoMove argument must always be used for this purpose.

Note

The specified RAPID procedure is not processed during backwards instruction by instruction execution or when searching for the first position, nor during backwards movement with MT_MoveRoutine.

Tip

The trigger events are not executed during movement to the home position if the execution of trigger events (ExecTriggEvt) has been disabled in the system parameters (FALSE).

Syntax

```
MT_TriggJ
['\'Conc ',']
[ActPos ':='] < expression (IN) of dnum> ','
[ToPoint ':='] < expression (IN) of robtarget > ','
[speed ':='] < expression (IN) of speeddata >
['\'T':=' < expression (IN) of num > ] ','
[Trigg1 ':=' ] < Variable (VAR) as triggdata >
['\'T2 ':=' < Variable (VAR) as triggdata > ]
['\'T3 ':=' < Variable (VAR) as triggdata > ]
['\'T4 ':=' < Variable (VAR) as triggdata > ]
['\'T5 ':=' < Variable (VAR) as triggdata > ]
['\'T6 ':=' < Variable (VAR) as triggdata > ]
```

Continues on next page

15.2.48 MT_TriggJ – Axis-wise robot movements with events
Continued

```
['\'T7 ':=' < Variable (VAR) as triggdata > ]
[Zone ':=' ] < expression (IN) of zonedata >
['\'Inpos ':=' < expression (IN) of stoppointdata > ]','
[Tool ':='] < persistent (PERS) of tooldata >
['\' WObj ':=' < persistent (PERS) of wobjdata > ]';'
```

15.2.49 MT_TriggL – Linear robot movements with events

Usage

MT_TriggL (Trigg Linear) is used to set output signals and/or interrupt routines in fixed positions whilst the robot is making linear movements. This position number assignment takes place in the middle of the corner path.

One or more (maximum of 7) events can be defined using the TriggIO, TriggEquip, TriggInt, TriggCheckIO, TriggSpeed or TriggRampAO instruction. Then reference can be made to these definitions with the TriggL instruction.

This instruction can only be used in the Main task T_ROB1 or in motion tasks in the case of a MultiMove system.

The instruction basically corresponds to a TriggL with some additions.

Basic example

```
VAR triggdata trGlueing_On;
VAR triggdata trGlueing_Off;

PROC InitTrigger()
TriggIO trGlueing_On, 10 \Start \DOp:=doGlueing, 1;
TriggIO trGlueing_Off, 10 \Start \DOp:= doGlueing, 0;
ENDPROC


PROC mv10_11()
MT_MoveJ 10, p10, v1000, z30, tGripper\NoMove;
MT_TriggJ 101101,p101101,v500,trGluing_On,z10,tGripper;
...
MT_TriggL 101105,p101105,v500,trGluing_Off,z10,tGripper;
…
ENDPROC
```

The digital output signal doGlueing is set if the TCP of the robot is 10 mm in front of point p101101 . 101101 is then saved as the current position.

The digital output signal doGlueing is reset if the TCP of the robot is 10 mm in front of point p101105 . 101105 is then saved as the current position.

Arguments

```
MT_TriggL [\Conc] ActPos ToPoint speed [\T] Trigg1 [\T2]
[\T3] [\T4] [\T5] [\T6] [\T7] zone [\Inpos]
Tool [\WObj] [\Corr]
```

[\Conc] Concurrent	Data type: switch The following instructions are executed whilst the robot is in motion. Further information can be obtained from the MoveL instruction.
Actpos	Data type: dnum Contains the position designation of the position to be moved to.

Continues on next page

15.2.49 MT_TriggL – Linear robot movements with events
Continued

`ToPoint`	Data type: `robtarget` The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).
`Speed`	Data type: `speeddata` The speed programmed for the movement. The speed data define the velocity of the TCP, of the tool reorientation and of external axes.
`[\T] (Time)`	Data type: `num` This argument is used to specify the time in seconds during which a movement of the manipulator and of the external axes should be executed. This value is then substituted for the corresponding speed data.
`Trigg1`	Data type: `Data type: triggdata` A trigger condition-related and trigger activity-related variable that has been defined previously in the program using the `TriggIO`, `TriggEquip`, `TriggInt`, `TriggSpeed`, `TriggCheckIO` or `TriggRampAO` instructions.
`[\T2] (Trigg2)`	Data type: `Data type: triggdata` A trigger condition-related and trigger activity-related variable that has been defined previously in the program using the `TriggIO`, `TriggEquip`, `TriggInt`, `TriggSpeed`, `TriggCheckIO` or `TriggRampAO` instructions.
`[\T3 ] (Trigg3)`	Data type: `triggdata` A trigger condition-related and trigger activity-related variable that has been defined previously in the program using the `TriggIO`, `TriggEquip`, `TriggInt`, `TriggSpeed`, `TriggCheckIO` or `TriggRampAO` instructions.
`[\T4 ] (Trigg4)`	Data type: `triggdata` A trigger condition-related and trigger activity-related variable that has been defined previously in the program using the `TriggIO`, `TriggEquip`, `TriggInt`, `TriggSpeed`, `TriggCheckIO` or `TriggRampAO` instructions.
`[\T5 ] (Trigg5)`	Data type: `triggdata` A trigger condition-related and trigger activity-related variable that has been defined previously in the program using the `TriggIO`, `TriggEquip`, `TriggInt`, `TriggSpeed`, `TriggCheckIO` or `TriggRampAO` instructions.
`[\T6 ] (Trigg6)`	Data type: `triggdata` A trigger condition-related and trigger activity-related variable that has been defined previously in the program using the `TriggIO`, `TriggEquip`, `TriggInt`, `TriggSpeed`, `TriggCheckIO` or `TriggRampAO` instructions.
`[\T7 ] (Trigg7)`	Data type: `triggdata` A trigger condition-related and trigger activity-related variable that has been defined previously in the program using the `TriggIO`, `TriggEquip`, `TriggInt`, `TriggSpeed`, `TriggCheckIO` or `TriggRampAO` instructions.
`Zone`	Data type: `zonedata` Zone data for the movement. Zone data describe the distance in which the axes must stand from the destination point before the next instruction is executed.

Continues on next page

[\Inpos]	Data type: stoppointdata This argument is used to specify the convergence criteria for the position of the robot's TCP in the stop point. The stop point data substitutes the zone specified in the Zone parameter.
Tool	Data type: tooldata The tool in use when the robot moves. The tool centre point is moved to the programmed destination point.
[\Wobj]	Data type: wobjdata The work object (tool coordinate system) to which the robot position in the instruction is related. This argument can be omitted. In this case the position relates to the world coordinate system.
[\Corr]	Data type: switch Correction data that has been written in a correction entry using the CorrWrite instruction is added to the path and the destination position if this argument is present.

Program execution

More information about linear movements and trigger functions can be found in the explanations for instruction MoveL and TriggL.

The passed position number is saved as the current robot position in the middle of the corner path of the destination position.

Note

The MT_TriggL instruction may never be used as the first movement instruction in a movement routine. MT_MoveL or MT_MoveJ with the \NoMove argument must always be used for this purpose.

Note

The specified RAPID procedure is not processed during backwards instruction by instruction execution or when searching for the first position, nor during backwards movement with MT_MoveRoutine.

Tip

The trigger events are not executed during movement to the home position if the execution of trigger events (ExecTriggEvt) has been disabled in the system parameters (FALSE).

Syntax

```
MT_TriggL
['\'Conc ',']
[ActPos ':='] < expression (IN) of dnum> ','
[ToPoint ':='] < expression (IN) of robtarget > ','
[speed ':='] < expression (IN) of speeddata >
['\'T':=' < expression (IN) of num > ] ','
[Trigg1 ':=' ] < Variable (VAR) as triggdata >
['\'T2 ':=' < Variable (VAR) as triggdata > ]
```

Continues on next page

15.2.49 MT_TriggL – Linear robot movements with events
Continued

```
['\'T3 ':=' < Variable (VAR) as triggdata > ]
['\'T4 ':=' < Variable (VAR) as triggdata > ]
['\'T5 ':=' < Variable (VAR) as triggdata > ]
['\'T6 ':=' < Variable (VAR) as triggdata > ]
['\'T7 ':=' < Variable (VAR) as triggdata > ]
[Zone ':=' ] < expression (IN) of zonedata >
['\'Inpos ':=' < expression (IN) of stoppointdata > ]','
[Tool ':='] < persistent (PERS) of tooldata >
['\'WObj ':=' < persistent (PERS) of wobjdata > ]
['\'Corr ]';'
```

15.2.50 MT_UIMessage – Message display based on UIMessageBox

Usage

`MT_UIMessage` is used for outputting a message that has been declared as msgdata on the programming device.

`MT_UIMessage` essentially corresponds to the function `UIMessageBox`

Basic example

```
MT_UIMessage msgProgValid\NoErrorLog;
```

Message output `msgProgValid`, whereby the entry will be suppressed in the event log.

Arguments

```
MT_UIMessage Msg [\SubHeader] [\Info] [\NoErrorLog] [\DOBreak]
     [\DIBreak] [\Buttonresult] [\BtnArray] [\MaxTime]
     [\NoErrorDelete]
```

`Msg` Data type: `msgdata`
Message text and error number declaration

`[\Subheader]` Data type: `string`
Optional text which is output above the message text.

`[\Info]` Data type: `string`
Optional additional text which is output under the message text.

`[\NoErrorLog]` Data type: `switch`
Optional switch that will prevent the error text from being written into the event log. If the switch not used, then all the messages, whose error domains and error domain number are greater than zero will be entered in the event log.

`[\DOBreak]` Data type: `signaldo`
The digital output signal, which can interrupt the operator dialog. In case no button selected, if the signal is set to 1 (or is already at 1), the program will continue.

`[\DIBreak]` Data type: `signaldi`
The digital input signal, which can interrupt the operator dialog. In case no button selected, if the signal is set to 1 (or is already at 1), the program will continue.

`[\Buttonresult]` Data type: `btnres`
The numerical value of the button that was printed in the printed dialog.
Generally, the pre-defined symbolic constants of the type btnres are used as return value.
If a `BtnArray` is passed, then, the values 11 to 15 will be returned for the function keys.

`[\BtnArray]` Data type: `userbutton`
Separate definition of buttons that are saved in the data type user-button.

`[\MaxTime]` Data type: `num`
The maximum time in seconds for which the program execution will wait. If no key is selected within this time, then the program will be continued.

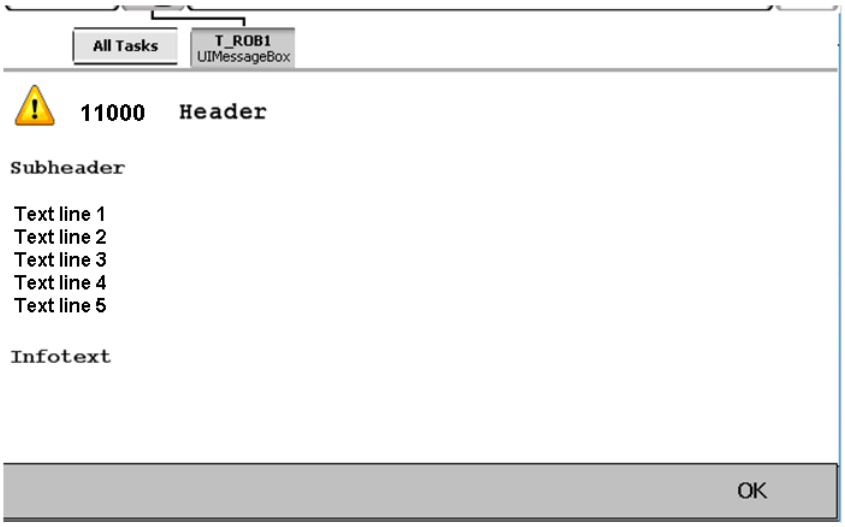
Continues on next page

`[\NoErrorDelete]` Data type: `switch`

If the error domain and error number of this message is greater than 0, this information is send through the appropriate digital group outputs (if available). The switch `\NoErrorDelete` prevents the group outputs from being reset after the message has been confirmed.

Program execution

The message window with the icon, the header, the message rows, image and buttons will be displayed in accordance with the programmed arguments. The program execution waits till a user presses a button or the message window is interrupted by a timeout or a signal. The user selection (button result) could be returned to the program.



en1300000160

If the cancel signal `DIBreak` or `DOBreak` is used, then the message will be ended prematurely and the program run will continue.

If the domain number and the error number are greater than zero, then these will be communicated to the PLC through the group outputs as error. In addition to this, the message will be entered in the event log, if this is not prevented explicitly by the optional parameter `\NoErrorLog`.

Syntax

```
MT_UIMessage

[ Msg ´:=´ < expression (IN) of msgdata >]
[´\Subheader´:=´ < expression (IN) of string >]
[´\Info´:=´ < expression (IN) of string>]
[´\NoErrorLog´]
[´\DOBreak ´:=´ < expression (VAR) of signaldo >]
[´\DIBreak ´:=´ < expression (VAR) of signaldi >]
[´\Buttonresult ´:=´ < expression (INOUT) of btnres >]
[´\BtnArray ´:=´ < expression (IN) of userbutton >]
[´\MaxTime ´:=´ < expression (IN) of num >]
[´\NoErrorDelete ´:=´ < expression (IN) of switch >]
```

Continues on next page

More information

Information about	See
Instruction `UIMessageBox`	*Technical Reference Manual – Instructions, Functions and Data Types* listed in the section *References on page 11*
Data type `msgdata`	*msgdata – Message declaration on page 280*

15.2.51 MT_UserInit – User routine for initialization

Usage

`MT_UserInit` is a routine for user initialization purpose, which should reasonably be located in the module `MT_Main.mod` but can also be located in any other user-defined RAPID module.

If the user program needs some initial steps to be done, when the program is started “from main”, then those steps can be placed in the routine `MT_UserInit`.

RWMT provides the routine `MT_UserInit` in the template module `MT_Main.mod`, which comes with the additional option. If it is not needed it can be removed by the integrator.

Basic example

```
PROC MT_UserInit()
!Do user initialization here
ENDPROC
```

Arguments

`MT_UserInit`

No arguments.

Program execution

The routine will be called automatically after the internal RWMT initialization (after the `EE_AFTER_INIT` event (see *Event handling on page 93*)). It does not need to be called by the user program itself.

Syntax

```
MT_UserInit´;´
```

15.2.52 MT_WaitMsgDI – Wait for input signal state

Usage

`MT_WaitMsgDI` is used to wait for the expected status of an input signal and if the condition is not fulfilled, to display a message on the Flexpendant.

The message will, beside of the user-defined content, automatically contain the signals which have not reached the expected state.

Basic example

```
MT_WaitMsgDI diRelease1,high\diAnd:=diRelease2 \SecValue :=
high,msgScrtest2\MaxTime:=1\NoErrorLog;
```

The instruction waits till the input signals `diRelease1` and `diRelease2` are "high".

Arguments

```
MT_WaitMsgDI [\InPos]|[\ZeroSpeed] diSignal Value
     [\diAND]|[\diOR]|[diXOR][\SecValue] Msg [\SubHeader] [\Info]
     [\NoSignalInfo] [\MaxTime] [\NoErrorLog] [\Pollrate] [\ERR]
```

`[\InPos]` Data type: `switch`

If this argument is used, then the robot and the external axes must have reached the stop point (ToPoint of the current movement instruction) before the processing can continue.

`[\ZeroSpeed]` Data type: `switch`

If this argument is used, then the robot and the external axes must have the speed zero before the processing can continue.

`diSignal` Data type: `signaldi`

The name of the digital input signal.

`Value` Data type: `dionum`

The desired value of the signal.

`[\diAnd]` Data type: `signaldi`

Optional name of a digital input, which is queried through an AND query together with diSignal.

`[\diOR]` Data type: `signaldi`

Optional name of a digital input, which is queried through an OR query together with the diSignal.

`[\diXOR]` Data type: `signaldi`

Optional name of a digital input, which is queried through an XOR query together with the diSignal.

`[\SecValue]` Data type: `dionum`

The desired value of the second signal. If this parameter is not passed, then the signal will be queried for high.

`Msg` Data type: `msgdata`

Message that is to be displayed. In the data type msgdata, all the data is saved for display on the hand held programming device.

`[\Subheader]` Data type: `string`

Optional additional text which is output above the message text.

`[\Info]` Data type: `string`

Optional additional text which is output under the message text.

Continues on next page

15.2.52 MT_WaitMsgDI – Wait for input signal state
Continued

[\NoSignalInfo]	Data type: switch Optional switch which omits the automatically created additional information about signals, that have not reached the expected state.
[\MaxTime]	Data type: num Optional value, duration for which the instruction waits for the signal before a message is displayed. The default waiting period is 5 seconds.
[\NoErrorLog]	Data type: switch Optional switch that determines whether the error text should be written into the event log or not.
[\Pollrate]	Data type: num The polling rate in seconds, for checking if the condition has been fulfilled. This means that the MT_WaitMsgDI will check the condition immediately, and in case the value is other than TRUE the check will be repeated with the specified polling rate. The minimum value for the polling rate is 0.01 s. If this argument not used wird, then the default polling rate of 0.1 s will be set.
[\ERR]	Data type: errnum Error number which is called in the case of a cancellation and is forwarded to the routine that has been called.

Program execution

The instruction waits for a definite time for the condition to be fulfilled. If the condition is not fulfilled, then, with the help of an internal use of MT_UIMessage a message will be displayed and the error recognition that is declared in the message will be communicated to the PLC.

Syntax

```
MT_WaitMsgDI
[´\InPos´ ]
| [´\ZeroSpeed´ ]
[diSignal´:=´ < expression (VAR) of signaldi >]
[Value´:=´ < expression (IN) of dionum >]
[´\diAnd´:=´ < expression (IN) of signaldi >]
| [´\diOr´:=´ < expression (IN) of signaldi >]
| [´\diXOr´:=´ < expression (IN) of signaldi >]
[´\SecValue´:=´ < expression (IN) of dionum >]
[Msg ´:=´ < expression (IN) of > msgdata]
[´\Subheader´:=´ < expression (IN) of string >]
[´\Info´:=´ < expression (IN) of string>]
[´\NoSignalInfo´]
[´\MaxTime´:=´ < expression (IN) of num >]
[´\NoErrorLog´]
[´\Pollrate´:=´ < expression (IN) of num >]
[´\Err´:=´ < expression (IN) of errnum >]
```

15.2.53 MT_WaitMsgDO – Wait for output signal state

Usage

`MT_WaitMsgDO` is used to wait for a group output signal and if the condition is not fulfilled, to display a message on the Flexpendant.

The message will, beside of the user-defined content, automatically contain the signals which have not reached the expected state.

Basic example

```
MT_WaitMsgDO doStart1,high\doAnd:=doStart2\SecValue:=high,
msgStart\MaxTime:=1\NoErrorLog;
```

The instruction waits till the output signals `doStart1` and `doStart2` change to "high". If this does not happen within one second, then, an error message will be output, but this will not be written into the event log.

Arguments

```
MT_WaitMsgDO [\InPos]|[\ZeroSpeed] doSignal Value
     [\doAND]|[\doOR]|[\SecValue] Msg [\SubHeader] [\Info]
     [\NoSignalInfo] [\MaxTime] [\NoErrorLog] [\Pollrate] [\ERR]
```

`[\InPos]` Data type: `switch`

If this argument is used, then the robot and the external axes must have reached the stop point (ToPoint of the current movement instruction) before the processing can continue.

`[\ZeroSpeed]` Data type: `switch`

If this argument is used, then the robot and the external axes must have the speed zero before the processing can continue.

`doSignal` Data type: `signaldo`

The name of the digital output signal.

`Value` Data type: `dionum`

The desired value of the signal.

`[\doAnd]` Data type: `signaldo`

Optional name of a digital output, which is queried through an AND query together with doSignal.

`[\doOR]` Data type: `signaldo`

Optional name of a digital input, which is queried through an OR query together with the doSignal.

`[\SecValue]` Data type: `dionum`

The desired value of the second signal. If this parameter is not passed, then the signal will be queried for high.

`Msg` Data type: `msgdata`

Message that is to be displayed. In the data type msgdata, all the data is saved for display on the hand held programming device.

`[\Subheader]` Data type: `string`

Optional additional text which is output above the message text.

`[\Info]` Data type: `string`

Optional additional text which is output under the message text.

Continues on next page

15.2.53 MT_WaitMsgDO – Wait for output signal state
Continued

[\NoSignalInfo]	Data type: switch Optional switch which omits the automatically created additional information about signals, that have not reached the expected state.
[\MaxTime]	Data type: num Optional value, duration for which the instruction waits for the signal before a message is displayed. The default waiting period is 5 seconds.
[\NoErrorLog]	Data type: switch Optional switch that determines whether the error text should be written into the event log or not.
[\Pollrate]	Data type: num The polling rate in seconds, for checking if the condition has been fulfilled. This means that MT_WaitMsgDO will check the condition immediately, and in case a value other than TRUE is present, the check will be repeated with the specified polling rate. The minimum value for the polling rate is 0.01 s. If this argument not used wird, then the default polling rate of 0.1 s will be set.
[\ERR]	Data type: errnum Error number which is called in the case of a cancellation and is forwarded to the routine that has been called.

Program execution

The instruction waits for a definite time for the condition to be fulfilled. If the condition is not fulfilled, then, with the help of an internal use of MT_UIMessage a message will be displayed and the error recognition that is declared in the message will be communicated to the PLC.

Syntax

```
MT_WaitMsgDO
[´\InPos´ ]
| [´\ZeroSpeed´ ]
[doSignal´:=´ < expression (VAR) of signaldo >]
[Value´:=´ < expression (IN) of dionum >]
[´\doAnd´:=´ < expression (IN) of signaldo >]
| [´\doOr´:=´ < expression (IN) of signaldo >]
[´\SecValue´:=´ < expression (IN) of dionum >]
[Msg ´:=´ < expression (IN) of > msgdata]
[´\Subheader´:=´ < expression (IN) of string >]
[´\Info´:=´ < expression (IN) of string>]
[´\NoSignalInfo´]
[´\MaxTime´:=´ < expression (IN) of num >]
[´\NoErrorLog´]
[´\Pollrate´:=´ < expression (IN) of num >]
[´\Err´:=´ < expression (IN) of errnum >]
```

15.2.54 MT_WaitMsgGI – Wait for a group input signal

Usage

`MT_WaitMsgGI` is used to wait for a group input signal and if the condition is not fulfilled, to display a message on the Flexpendant.

The message will, beside of the user-defined content, automatically contain the signals which have not reached the expected state.

Basic example

```
MT_WaitMsgGI giProgNumber, NOTEQ, 0, msgProgNumber;
```

The instruction waits till the group input signal giProgNumber is not equal to zero.

Arguments

```
MT_WaitMsgGI [\InPos]|[\ZeroSpeed] giSignal OPValue Value Msg
     [\SubHeader] [\Info] [\NoSignalInfo] [\MaxTime] [\NoErrorLog]
     [\Pollrate] [\ERR]
```

`[\InPos]` Data type: `switch`

If this argument is used, then the robot and the external axes must have reached the stop point (ToPoint of the current movement instruction) before the processing can continue.

`[\ZeroSpeed]` Data type: `switch`

If this argument is used, then the robot and the external axes must have the speed zero before the processing can continue.

`giSignal` Data type: `signalgi`

The name of the digital input group signal.

`OPValue` Data type: `opnum`

"OPValue" will be used for comparing of the value of the signal with the required "Value".

`[\doAnd]` Data type: `signaldo`

Optional name of a digital output, which is queried through an AND query together with doSignal.

The following constants can be used for the comparison:

Value	Symbolic constants	Comment
1	LT	Less than
2	LTEQ	Less than or equal
3	EQ	Equal
4	NOTEQ	Not equal
5	GTEQ	Greater than or equal
6	GT	Greater than

`Value` Data type: `num`

The desired value of the signal. Must be an integer within the validity range of the digital group input signal.

Permitted range of numbers: 0 to 8388608.

`Msg` Data type: `msgdata`

Specification of the message declaration which will be output after a waiting period on the programming device.

Continues on next page

15.2.54 MT_WaitMsgGI – Wait for a group input signal
Continued

[\Subheader]	Data type: string Optional additional text which is output above the message text.
[\Info]	Data type: string Optional additional text which is output under the message text.
[\NoSignalInfo]	Data type: switch Optional switch which omits the automatically created additional information about signals, that have not reached the expected state.
[\MaxTime]	Data type: num Optional value, duration for which the instruction waits for the signal before a message is displayed. The default waiting period is 5 seconds.
[\NoErrorLog]	Data type: switch Optional switch that determines whether the error text should be written into the event log or not.
[\Pollrate]	Data type: num The polling rate in seconds, for checking if the condition has been fulfilled. This means that the MT_WaitMsgGI will check the condition immediately, and case of a value other than TRUE, the check will be repeated with the specified polling rate. The minimum value for the polling rate is 0.01 s. If this argument not used wird, then the default polling rate of 0.1 s will be set.
[\ERR]	Data type: errnum Error number which is called in the case of a cancellation and is forwarded to the routine that has been called.

Program execution

The instruction waits for a definite time for the condition to be fulfilled. If the condition is not fulfilled, then, with the help of an internal use of MT_UIMessage a message will be displayed and the error recognition that is declared in the message will be communicated to the PLC.

Syntax

```
MT_WaitMsgGI
[´\InPos´ ]
| [´\ZeroSpeed´ ]
[giSignal´:=´ < expression (VAR) of signalgi >]
[OPValue´:=´ < expression (IN) of opvalue >]
[Value´:=´ < expression (IN) of num >]
[Msg ´:=´ < expression (IN) of > msgdata]
[´\Subheader´:=´ < expression (IN) of string >]
[´\Info´:=´ < expression (IN) of string>]
[´\NoSignalInfo´]
[´\MaxTime´:=´ < expression (IN) of num >]
[´\NoErrorLog´]
[´\Pollrate´:=´ < expression (IN) of num >]
[´\Err´:=´ < expression (IN) of errnum >]
```

15.2.55 MT_WaitMsgGI32 – Wait for a 32-Bit group input signal

Usage

`MT_WaitMsgGI32` is used to wait for a 32 bit wide group input signal and if the condition is not fulfilled, to display a message on the Flexpendant.

The message will, beside of the user-defined content, automatically contain the signals which have not reached the expected state.

Basic example

```
MT_WaitMsgGI32 giProgNumber, NOTEQ, 0, msgPrognumber;
```

The instruction waits till the group input signal giProgNumber is not equal to zero.

Arguments

```
MT_WaitMsgGI32 [\InPos] ]|[\ZeroSpeed] giSignal OPValue Value Msg
     [\SubHeader] [\Info] [\NoSignalInfo] [\MaxTime] [\NoErrorLog]
     [\Pollrate] [\ERR]
```

`[\InPos]` Data type: `switch`

If this argument is used, then the robot and the external axes must have reached the stop point (ToPoint of the current movement instruction) before the processing can continue.

`[\ZeroSpeed]` Data type: `switch`

If this argument is used, then the robot and the external axes must have the speed zero before the processing can continue.

`giSignal` Data type: `signalgi`

The name of the digital input group signal.

`OPValue` Data type: `opnum`

OPValue is used for comparing the value of the signal with the required value.

The following constants can be used for the comparison:

Value	Symbolic constants	Comment
1	LT	Less than
2	LTEQ	Less than or equal
3	EQ	Equal
4	NOTEQ	Not equal
5	GTEQ	Greater than or equal
6	GT	Greater than

`Value` Data type: `dnum`

The desired value of the signal. Must be an integer within the validity range of the digital group input signal.

Permitted range of numbers: 0 to 4294967295

`Msg` Data type: `msgdata`

Specification of the message declaration which will be output after a waiting period on the programming device.

`[\Subheader]` Data type: `string`

Optional additional text which is output above the message text.

Continues on next page

15.2.55 MT_WaitMsgGI32 – Wait for a 32-Bit group input signal
Continued

[\Info]	Data type: string Optional additional text which is output under the message text.
[\NoSignalInfo]	Data type: switch Optional switch which omits the automatically created additional information about signals, that have not reached the expected state.
[\MaxTime]	Data type: num Optional value, duration for which the instruction waits for the signal before a message is displayed. The default waiting period is 5 seconds.
[\NoErrorLog]	Data type: switch Optional switch that determines whether the error text should be written into the event log or not.
[\Pollrate]	Data type: num The polling rate in seconds, for checking if the condition has been fulfilled. This means that the MT_WaitMsgGI32 will check the condition immediately, and in the case of a value other than TRUE, the check will be repeated with the specified polling rate. The minimum value for the polling rate is 0.01 s. If this argument not used wird, then the default polling rate of 0.1 s will be set.
[\ERR]	Data type: errnum Error number which is called in the case of a cancellation and is forwarded to the error handling of the routine that has been called.

Program execution

The instruction waits for a definite time for the condition to be fulfilled. If the condition is not fulfilled, then, with the help of an internal use of MT_UIMessage a message will be displayed and the error recognition that is declared in the message will be communicated to the PLC.

Syntax

```
MT_WaitMsgGI32
[´\InPos´ ]
| [´\ZeroSpeed´ ]
[giSignal´:=´ < expression (VAR) of signalgi >]
[OPValue´:=´ < expression (IN) of opvalue >]
[Value´:=´ < expression (IN) of dnum >]
[Msg ´:=´ < expression (IN) of > msgdata]
[´\Subheader´:=´ < expression (IN) of string >]
[´\Info´:=´ < expression (IN) of string>]
[´\NoSignalInfo´]
[´\MaxTime´:=´ < expression (IN) of num >]
[´\NoErrorLog´]
[´\Pollrate´:=´ < expression (IN) of num >]
[´\Err´:=´ < expression (IN) of errnum >]
```

15.2.56 MT_WaitMsgGO – Wait for a group output signal

Usage

`MT_WaitMsgGO` is used to wait for a group output signal and if the condition is not fulfilled, to display a message on the Flexpendant.

The message will, beside of the user-defined content, automatically contain the signals which have not reached the expected state.

Basic example

```
MT_WaitMsgGO goExample, NOTEQ, 0, msgExample;
```

The instruction waits till the group output signal goExample is not equal to zero.

Arguments

```
MT_WaitMsgGO [\InPos]|[\ZeroSpeed] goSignal OPValue Value Msg
     [\SubHeader] [\Info] [\NoSignalInfo] [\MaxTime] [\NoErrorLog]
     [\Pollrate] [\ERR]
```

`[\InPos]` Data type: `switch`

If this argument is used, then the robot and the external axes must have reached the stop point (ToPoint of the current movement instruction) before the processing can continue.

`[\ZeroSpeed]` Data type: `switch`

If this argument is used, then the robot and the external axes must have the speed zero before the processing can continue.

`goSignal` Data type: `signalgo`

The name of the digital group output signal.

`OPValue` Data type: `opnum`

"OPValue" will be used for comparing the value of the signal with the required "Value".

The following constants can be used for the comparison:

Value	Symbolic constants	Comment
1	LT	Less than
2	LTEQ	Less than or equal
3	EQ	Equal
4	NOTEQ	Not equal
5	GTEQ	Greater than or equal
6	GT	Greater than

`Value` Data type: `num`

The desired value of the signal. Must be an integer within the validity range of the digital group input signal.

Permitted range of numbers: 0 to 8388608.

`Msg` Data type: `msgdata`

Specification of the message declaration which will be output after a waiting period on the programming device.

`[\Subheader]` Data type: `string`

Optional additional text which is output above the message text.

Continues on next page

15.2.56 MT_WaitMsgGO – Wait for a group output signal
Continued

[\Info]	Data type: string Optional additional text which is output under the message text.
[\NoSignalInfo]	Data type: switch Optional switch which omits the automatically created additional information about signals, that have not reached the expected state.
[\MaxTime]	Data type: num Optional value, duration for which the instruction waits for the signal before a message is displayed. The default waiting period is 5 seconds.
[\NoErrorLog]	Data type: switch Optional switch that determines whether the error text should be written into the event log or not.
[\Pollrate]	Data type: num The polling rate in seconds, for checking if the condition has been fulfilled. This means that the MT_WaitMsgGO will check the condition immediately, and in the case of a value other than TRUE, the check will be repeated with the specified polling rate. The minimum value for the polling rate is 0.01 s. If this argument not used wird, then the default polling rate of 0.1 s will be set.
[\ERR]	Data type: errnum Error number which is called in the case of a cancellation and is forwarded to the error handling of the routine that has been called.

Program execution

The instruction waits for a definite time for the condition to be fulfilled. If the condition is not fulfilled, then, with the help of an internal use of MT_UIMessage a message will be displayed and the error recognition that is declared in the message will be communicated to the PLC.

Syntax

```
MT_WaitMsgGO
[´\InPos´ ]
| [´\ZeroSpeed´ ]
[goSignal´:=´ < expression (VAR) of signalgo >]
[OPValue´:=´ < expression (IN) of opvalue >]
[Value´:=´ < expression (IN) of num >]
[Msg ´:=´ < expression (IN) of > msgdata]
[´\Subheader´:=´ < expression (IN) of string >]
[´\Info´:=´ < expression (IN) of string>]
[´\NoSignalInfo´]
[´\MaxTime´:=´ < expression (IN) of num >]
[´\NoErrorLog´]
[´\Pollrate´:=´ < expression (IN) of num >]
[´\Err´:=´ < expression (IN) of errnum >]
```

15.2.57 MT_WaitMsgGO32 – Wait for a 32-Bit group output signal

Usage

`MT_WaitMsgGO32` is used to wait for a 32 bit wide group input signal and if the condition is not fulfilled, to display a message on the Flexpendant.

The message will, beside of the user-defined content, automatically contain the signals which have not reached the expected state.

Basic example

```
MT_WaitMsgGO32 goExample, NOTEQ, 0, msgExample;
```

The instruction waits till the group output signal `goExample` is not equal to zero.

Arguments

```
MT_WaitMsgGO32 [\InPos] ]|[\ZeroSpeed] goSignal OPValue Value Msg
     [\SubHeader] [\Info] [\NoSignalInfo] [\MaxTime] [\NoErrorLog]
     [\Pollrate] [\ERR]
```

`[\InPos]` Data type: `switch`

If this argument is used, then the robot and the external axes must have reached the stop point (ToPoint of the current movement instruction) before the processing can continue.

`[\ZeroSpeed]` Data type: `switch`

If this argument is used, then the robot and the external axes must have the speed zero before the processing can continue.

`goSignal` Data type: `signalgo`

The name of the digital group output signal.

`OPValue` Data type: `opnum`

"OPValue" will be used for comparing the value of the signal with the required "Value".

The following constants can be used for the comparison:

Value	Symbolic constants	Comment
1	LT	Less than
2	LTEQ	Less than or equal
3	EQ	Equal
4	NOTEQ	Not equal
5	GTEQ	Greater than or equal
6	GT	Greater than

`Value` Data type: `dnum`

The desired value of the signal. Must be an integer within the validity range of the digital group input signal.

Permitted range of numbers: 0 to 4294967295

`Msg` Data type: `msgdata`

Specification of the message declaration which will be output after a waiting period on the programming device.

`[\Subheader]` Data type: `string`

Optional additional text which is output above the message text.

Continues on next page

15.2.57 MT_WaitMsgGO32 – Wait for a 32-Bit group output signal
Continued

[\Info]	Data type: string Optional additional text which is output under the message text.
[\NoSignalInfo]	Data type: switch Optional switch which omits the automatically created additional information about signals, that have not reached the expected state.
[\MaxTime]	Data type: num Optional value, duration for which the instruction waits for the signal before a message is displayed. The default waiting period is 5 seconds.
[\NoErrorLog]	Data type: switch Optional switch that determines whether the error text should be written into the event log or not.
[\Pollrate]	Data type: num The polling rate in seconds, for checking if the condition has been fulfilled. This means that the MT_WaitMsgGO32 will check the condition immediately, and in the case of a value other than TRUE, the check will be repeated with the specified polling rate. The minimum value for the polling rate is 0.01 s. If this argument not used wird, then the default polling rate of 0.1 s will be set.
[\ERR]	Data type: errnum Error number which is called in the case of a cancellation and is forwarded to the error handling of the routine that has been called.

Program execution

The instruction waits for a definite time for the condition to be fulfilled. If the condition is not fulfilled, then, with the help of an internal use of MT_UIMessage a message will be displayed and the error recognition that is declared in the message will be communicated to the PLC.

Syntax

```
MT_WaitMsgGO32
[´\InPos´ ]
| [´\ZeroSpeed´ ]
[goSignal´:=´ < expression (VAR) of signalgo >]
[OPValue´:=´ < expression (IN) of opvalue >]
[Value´:=´ < expression (IN) of dnum >]
[Msg ´:=´ < expression (IN) of > msgdata]
[´\Subheader´:=´ < expression (IN) of string >]
[´\Info´:=´ < expression (IN) of string>]
[´\NoSignalInfo´]
[´\MaxTime´:=´ < expression (IN) of num >]
[´\NoErrorLog´]
[´\Pollrate´:=´ < expression (IN) of num >]
[´\Err´:=´ < expression (IN) of errnum >]
```

15.2.58 MT_WaitMsgSync – Synchronization of movement tasks

Usage

`MT_WaitMsgSync` is used to synchronize several movement tasks with each other.

The instruction waits for a definite time for the specified tasks to synchronize with each other.

If the synchronization has not happened even after the waiting period has lapsed, then the instruction will decide with the help of the error number that has been passed, whether to continue waiting or end the waiting process.

Basic example

```
VAR syncident sidStart;
PERS tasks tskAllRobots{2}:=[["T_ROB1"],["T_ROB2"]];
CONST msgdata msgStart:=…

…
MT_WaitMsgSync sidStart,tskAllRobots,msgStart;
…
```

The system waits for the synchronization of the movement tasks `T_ROB1` and `T_ROB2` with the help of the `Sync-ID` `sidStart`. After 10 seconds have elapsed without synchronization, the message `msgStart` will be displayed.

Arguments

```
MT_WaitMsgSync SyncID TaskList Msg [\SubHeader] [\Info]
[\MaxTime] [\NoErrorLog]
```

`SyncID`	Data type: `syncident` ID for specifying the correct synchronization process.
`Tasklist`	Data type: `tasks` List of tasks that are to be synchronized
`Msg`	Data type: `msgdata` Message that is to be output if the maximum waiting period is exceeded
`[\Subheader]`	Data type: `string` Optional additional text which is output above the message text.
`[\Info]`	Data type: `string` Optional additional text which is output under the message text.
`[\MaxTime]`	Data type: `num` Optional value, duration for which the instruction waits for the synchronization before a message is displayed. The default waiting period is 10 seconds.
`[\NoErrorLog]`	Data type: `switch` Optional switch that determines whether the error text should be written into the event log or not.

Continues on next page

15.2.58 MT_WaitMsgSync – Synchronization of movement tasks
Continued

Program execution

The instruction waits for a definite time for the corresponding tasks to be synchronized. If this does not happen, a message will be displayed.

Restrictions

This instruction may only be used in Multimove applications.

Syntax

```
MT_WaitMsgSync
[SyncID´:=´ < expression (VAR) of syncident >]
[TaskList´:=´ < expression (IN) of tasks >]
[Msg ´:=´ < expression (IN) of > msgdata]
[´\Subheader´:=´ < expression (IN) of string >]
[´\Info´:=´ < expression (IN) of string>]
[´\MaxTime´:=´ < expression (IN) of num >]
[´\NoErrorLog´]´;´
```

15.2.59 MT_WaitTimeDI – Wait for input signal until time limit is exceeded

Usage

`MT_WaitTimeDI` is used to wait for the expected status of an input signal and if the condition is not fulfilled, to raise with a specified error number.

When raising to the error handler, a message will be shown. The message will, beside of the user-defined content, automatically contain the signals which have not reached the expected state.

Basic example

```
CONST errnum ERR_RELEASE:=78;

PROC Release()
!Wait until release is given or timeout
MT_WaitTimeDI diRelease,high, msgRelease,3, ERR_RELEASE;

ERROR
IF ERRNO = ERR_RELEASE THEN
RAISE;
ELSE
…
ENDIF

ENDPROC
```

The instruction waits till the input signals diRelease is "high". If the expected signal state is not reached until the maximum waiting time of 3 seconds is exceeded, the program flow is continued in the error handler, using the assigned error number.

Arguments

```
MT_WaitTimeDI diSignal Value [\diAND]|[\diOR] [\low] msg
     [\NoSignalInfo] MaxTime ERR
```

`diSignal` — Data type: `signaldi`
The name of the digital input signal.

`Value` — Data type: `dionum`
The desired value of the signal.

`[\diAND]` — Data type: `signaldi`
Optional name of a digital input, which is queried through an AND query together with `diSignal`.

`[\diOR]` — Data type: `signaldi`
Optional name of a digital input, which is queried through an OR query together with the diSignal.

`[\low]` — Data type: `switch`
If this parameter is passed, then the signal will be queried for low, otherwise for high.

`msg` — Data type: `msgdata`
Message that is to be displayed. In the data type msgdata, all the data is saved for display on the hand held programming device.

Continues on next page

15.2.59 MT_WaitTimeDI – Wait for input signal until time limit is exceeded
Continued

[\NoSignalInfo]	Data type: switch Optional switch which omits the automatically created additional information about signals, that have not reached the expected state.
MaxTime	Data type: num Duration for which the instruction waits for the signal before a message is displayed and the error handler is called. The default waiting period is 5 seconds.
ERR	Data type: errnum Error number which is called in the case of a timeout and is forwarded to the routine that has been called.

Program execution

The instruction waits for a definite time for the condition to be fulfilled. If the condition is not fulfilled, then, with the help of an internal use of MT_UIMessage a message will be displayed, the error recognition that is declared in the message will be communicated to the PLC and the error handler is called using the specified error number.

Syntax

```
MT_WaitTimeDI
[diSignal´:=´ < expression (VAR) of signaldi >]
[Value ´:=´ < expression (IN) of dionum >]
[´\diAND ´:=´ < expression (IN) of signaldi >]
| [´\diOR ´:=´ < expression (IN) of signaldi >]
[´\low´]
[msg ´:=´ < expression (IN) of > msgdata]
[´\NoSignalInfo´]
[MaxTime ´:=´ < expression (IN) of num >]
[ERR ´:=´ < expression (IN) of errnum >]
```

15.2.60 MT_WaitTimeDO – Wait for output signal until time limit is exceeded

Usage

`MT_WaitTimeDO` is used to wait for the expected status of an output signal and if the condition is not fulfilled, to raise with a specified error number.

When raising to the error handler, a message will be shown. The message will, beside of the user-defined content, automatically contain the signals which have not reached the expected state.

Basic example

```
CONST errnum ERR_RELEASE:=78;

PROC Release()
!Wait until release is given or timeout
MT_WaitTimeDO doRelease,high, msgRelease,3, ERR_RELEASE;

ERROR
IF ERRNO = ERR_RELEASE THEN
RAISE;
ELSE
…
ENDIF

ENDPROC
```

The instruction waits till the input signals `doRelease` is "high". If the expected signal state is not reached until the maximum waiting time of 3 seconds is exceeded, the program flow is continued in the error handler, using the assigned error number.

Arguments

```
MT_WaitTimeDO doSignal Value [\doAND]|[\doOR] [\low] msg
     [\NoSignalInfo] MaxTime ERR
```

`doSignal`
Data type: `signaldo`
The name of the digital output signal.

`Value`
Data type: `dionum`
The desired value of the signal.

`[\doAND]`
Data type: `signaldo`
Optional name of a digital output, which is queried through an AND query together with `doSignal`.

`[\doOR]`
Data type: `signaldo`
Optional name of a digital output, which is queried through an OR query together with the `doSignal`.

`[\low]`
Data type: `switch`
If this parameter is passed, then the signal will be queried for low, otherwise for high.

`msg`
Data type: `msgdata`
Message that is to be displayed. In the data type msgdata, all the data is saved for display on the hand held programming device.

Continues on next page

15.2.60 MT_WaitTimeDO – Wait for output signal until time limit is exceeded
Continued

[\NoSignalInfo]	Data type: switch Optional switch which omits the automatically created additional information about signals, that have not reached the expected state.
MaxTime	Data type: num Duration for which the instruction waits for the signal before a message is displayed and the error handler is called. The default waiting period is 5 seconds.
ERR	Data type: errnum Error number which is called in the case of a timeout and is forwarded to the routine that has been called.

Program execution

The instruction waits for a definite time for the condition to be fulfilled. If the condition is not fulfilled, then, with the help of an internal use of MT_UIMessage a message will be displayed, the error recognition that is declared in the message will be communicated to the PLC and the error handler is called using the specified error number.

Syntax

```
MT_WaitTimeDO
[doSignal´:=´ < expression (VAR) of signaldo >]
[Value ´:=´ < expression (IN) of dionum >]
[´\doAND ´:=´ < expression (IN) of signaldo >]
| [´\doOR ´:=´ < expression (IN) of signaldo >]
[´\low´]
[msg ´:=´ < expression (IN) of > msgdata]
[´\NoSignalInfo´]
[MaxTime ´:=´ < expression (IN) of num >]
[ERR ´:=´ < expression (IN) of errnum >]
```

15.3 Functions

15.3.1 MT_EndOfCycleOk – Check if "Halt after end of cycle" was acknowledged

Usage

`MT_EndOfCycleOk` is used to check if the "Halt after end of cycle" request has been acknowledged already by the application program. The function returns TRUE, if the request has been acknowledged already, else it returns FALSE.

Basic example

```
PROC Production()
!If "halt after end of cycle" has been requested
IF MT_EndOfCycleReq() THEN
!Execute run-out cycle
RunOutCycle;
!Send notification: "halt after end of cycle reached"
MT_EndOfCycleAck;
ELSE
!Execute normal production cycle
NormalCycle;
ENDIF
…
…
!Move to home if program has confirmed "halt after end
!of cycle reached" before
IF MT_EndOfCycleOk() MoveTo 999;
ENDPROC
```

At the start of the production cycle, there is a query asking if "halt after end of cycle" has been requested (`MT_EndOfCycleReq`). If this is the case, then a run-out cycle is executed for example, for emptying part buffers in the cell.

Now the program confirms the request for “halt after end of cycle” by “halt after end of cycle reached” (instruction `MT_EndOfCycleAck`). This causes the RWMT engine to terminate the program after having left the production routine.

In the further production flow, the user program makes sure through the function `MT_EndOfCycleOk` that it has confirmed the “halt after end of cycle” before. If this is the case then the robot moves to home position.

Program execution

With the check `MT_EndOfCycleOk`, it is possible to determine whether the application program has acknowledged the current "Halt after end of cycle" request.

If the request has been acknowledged already, it means that the program will be ended after the routines of the current production cycle and subsequent return to the RWMT Engine.

Syntax

```
MT_EndOfCycleOk ´(´ ´)´
```

Continues on next page

15.3.1 MT_EndOfCycleOk – Check if "Halt after end of cycle" was acknowledged
Continued

A function with a return value of the type bool.

More information

Information about	See
Confimation of the "halt after end of cycle" request	*MT_EndOfCycleAck – Acknowledge the request "Halt after end of cycle" on page 317*
Recognizing request for "halt after end of cycle"	*MT_EndOfCycleReq – Recognizing the request "Halt after end of cycle" on page 459*

15.3.2 MT_EndOfCycleReq – Recognizing the request "Halt after end of cycle"

Usage

MT_EndOfCycleReq is used to recognize a "Halt after end of cycle" request. Here the request can be triggered by a digital input signal or even at the RWMT user interface.

The function returns TRUE if "Halt after end of cycle" has been requested, else it returns FALSE.

Basic example

```
PROC Production()
!If “halt after end of cycle“ has been requested
IF MT_ EndOfCycleReq() THEN
!Execute run-out cycle
RunOutCycle;
!Send notification: “halt after end of cycle reached“
MT_EndOfCycleAck;
ELSE
!Execute normal production cycle
NormalCycle;
ENDIF
…
…
!Move to home if program has confirmed “halt after end
!of cycle reached” before
IF MT_EndOfCycleOk() MoveTo 999;
ENDPROC
```

At the start of the production cycle, there is a query asking if "halt after end of cycle" has been requested (MT_EndOfCycleReq). If this is the case, then a run-out cycle is executed for example, for emptying part buffers in the cell.

Now the program confirms the request for “halt after end of cycle” by “halt after end of cycle reached” (instruction MT_EndOfCycleAck). This causes the RWMT engine to terminate the program after having left the production routine.

In the further production flow, the user program makes sure through the function MT_EndOfCycleOk that it has confirmed the “halt after end of cycle” before. If this is the case then the robot moves to home position.

Program execution

If, after the "Halt after end of cycle" has been requested by the application program, this request is acknowledged, then the program run will end, leaving the application programs and returning to the RWMT Engine. A fresh program start will then start from the first program instruction (program start "from main").

Syntax

```
MT_EndOfCycleReq ´(´ ´)´
```

A function with a return value of the type bool.

Continues on next page

15.3.2 MT_EndOfCycleReq – Recognizing the request "Halt after end of cycle"
Continued

More information

Information about	See
Acknowledgement of the "Halt after end of cycle" request	*MT_EndOfCycleAck – Acknowledge the request "Halt after end of cycle" on page 317*
Query if the "Halt after end of cycle" request has been acknowledged already	*MT_EndOfCycleOk – Check if "Halt after end of cycle" was acknowledged on page 457*

15.3.3 MT_FirstCycle – Requesting first cycle status

Usage

`MT_FirstCycle` is used to find out if the production loop is executed for the first time after a “start from main”.

The function returns TRUE if the production loop is executed for the first time, otherwise it returns FALSE.

Basic example

```
PROC Production()
!If this is the first production cycle
IF MT_ FirstCycle() THEN
!Execute tool cleaning
ToolCleaning;
ENDIF
!Do the normal production
…
…
ENDPROC
```

At the beginning of the production routine, the tool is cleaned if this is the first production loop after “start from main”.

Program execution

The function `MT_FirstCycle` returns the “first cycle” state which can be used. For example, for different actions apart from the normal (not first) production flow.

Syntax

```
MT_FirstCycle ´(´ ´)´´
```

A function with a return value of the type bool.

More information

Information about	See
Ending the first cycle state	*MT_ResetFirstCycle – Declare first cycle as finished on page 402*

15.3.4 MT_GetActualPosition – Reading the start position for MT_MoveTo

Usage

`MT_GetActualPosition` is used to read the start position that is used by the MT_MoveTo instruction as a numerical value.

With the help of the instruction `MT_SetActualPosition`, the start position, which is used by MT_MoveTo, can be initialized.

The procedure `MT_MoveTo` is used to move from the current robot position to the desired target position. For this, the robot forms a string using the saved start position and the target position that has been passed. This string represents the name of the movement routine that is to be called. The movement routine will be called dynamically and the target position will be saved as the new start position.

At the start of a program, it may be necessary to read the start position or assign it once again.

Basic example

```
If the current position is neither the basic setting (10)
!nor the home position (999)
IF GetActualPosition() <> 10 AND GetActualPosition() <> 999 THEN
!Start position is 999 (home).
MT_SetActualPosition 999;
ENDIF
…
!Call movement routine mv999_20
MT_MoveTo 20;
!Call the movement routine mv20_30
MT_MoveTo 30;
…
```

Syntax

```
GetActualPosition
'(' ')'
```

Function with a return value of the type num.

More information

Information about	See
Call movement routines with `MT_MoveTo` dynamically	*MT_MoveTo – Dynamic execution of a movement routine on page 391*
Setting the current position for using `MT_MoveTo`	*stationdata – Definition of a station on page 297*

15.3.5 MT_GetAuxCode – Reading the auxiliary code of the current part type

Usage

`MT_GetAuxCode` is used to read the auxiliary code of the current part type (for current partdata declaration, please refer to the chapter *partdata – Part data on page 284*).

This information can be used, if external equipment like a vision system uses different part numbers than the robot program.

Basic example

```
!Send current auxiliary code instead of the part type
!number because of a different convention in the vision system
SendToVisionSystem MT_GetAuxCode();
…
```

Syntax

```
GetAuxCode
'(' ')'
```

Function with a return value of the type dnum.

More information

Information about	See
Data type `partdata`	*partdata – Part data on page 284*

15.3.6 MT_GetCycleCountDown – Count-down value for currently executed cycle

Usage

The `MT_GetCycleCountDown` function reads the count-down value of the cycle which is currently executed.

Basic example

```
PROC MT_Main
CONST eventdata edAfterProd:= [EE_AFTER_PROD," MT_Main:
     AfterProdEventRoutine","",1];
VAR num nCycCntDown:=0;
CONST infodata MT_InfoView{1}:=
[["Parts to be produced","nCycCntDown","MainModule",""]];
…
…
PROC
…
nCycCntDown:= MT_GetCycleCountDown();
…
ENDPROC
ENDMODULE
```

The event routine `AfterProdEventRoutine` is executed after each production cycle and reads the current value of the cycle count-down. The infodata declaration shows the value in the GUI.

Program execution

Each cycledata declaration has a “current number of executions” and a “set number of executions”. `GetCycleCountDown` calculates and returns the remaining number of executions for the currently executed cycle.

Syntax

```
MT_GetCycleCountDown ´(´ ´)´
```

A function with a return value of the type num.

More information

Information about	See
Cycle declarations (`cycledata`)	*Cycledata – Program cycle setting on page 246*

15.3.7 MT_GetCycleIndex – Reading the current cycle index

Usage

`MT_GetCycleIndex` is used to determine the cycle that is to be executed currently.

Cycles are execution variants during the production, such as in a production process, in which the system is first filled (filler cycle), then the normal production takes place (normal cycles) and at the end of the production, the system is run idle (idle run cycles).

The cycle thus tells the application program which subordinate run (for example, filler cycle of the system) is to be executed.

For this, the function returns the numerical index of the current cycle.

Basic example

```
!List of cycles
TASK PERS cycledata MT_CycleList{20}:=
[
["Start-up","",1,1,2,0,2,0],
["Normal","",2,1,3,0,3,0],
["Run-out","",3,1,1,0,0,0],
…
PROC Produktion()
!If a start up cycle is to be executed
IF MT_GetCycleIndex()=MT_CycleList{1}.Index THEN
StartUpCycle;
!If a normal cycle is to be executed
ELSEIF MT_GetCycleIndex()=MT_CycleList{2}.Index THEN
NormalCycle;
!If idle run cycle is to be executed
ELSEIF MT_GetCycleIndex()=MT_CycleList{3}.Index THEN
RunOutCycle;
ENDIF

ENDPROC
```

Depending on the specification of the RWMT Engine, a specific cycle is to be executed.

Program execution

Depending on the cycle index, a specific processing cycle must be executed in the program. It is not mandatory to use cycles in the production program is, but RWMT supports it.

Syntax

`MT_GetCycleIndex ´(´ ´)´`

A function with a return value of the type num.

Continues on next page

15.3.7 MT_GetCycleIndex – Reading the current cycle index
Continued

More information

Information about	See
Setting up of cycles	*Program cycles on page 68*

15.3.8 MT_GetOperationMode – Current cell operation mode

Usage

`MT_GetOperationMode` returns the current RWMT cell operation mode.

Basic example

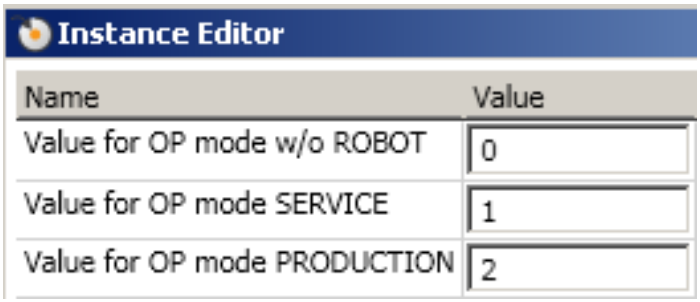
```
PROC Production()
…
!Find out cell operation mode
IF MT_GetOperationMode()=2 THEN
MT_Showtext ”RWMT operation mode is <Production>”;
…
ENDIF
…
ENDPROC
```

The current cell operation mode is retrieved.

Program execution

MT_GetOperationMode returns the current cell operation mode. The value which is returned depends on the settings in the system configuration (see *MT API Commands on page 154*).

Excerpt:



en1300000274

Syntax

```
MT_GetOperationMode ´(´ ´)´
```

A function with a return value of the type num.

More information

Information about	See
Cycle declarations (`cycledata`)	*Cycledata – Program cycle setting on page 246*

15.3.9 MT_GetPartType – Querying the current part type code

Usage

With the help of `MT_GetPartType`, the current part type code of the type dnum can be read.

To each declaration of type partdata a program number and, if present, a type code is related.

The type code is used to have a mapping between the program number, transferred by for example, an external PLC and a number that is to be used in the robot program.

Basic example

```
CONST partdata pdPartType1:=
["Teiletyp 1","Produktion","",FALSE,120,1,…;
CONST partdata pdPartType2:=
["Teiletyp 2","Production","",FALSE,199,2,…;

PROC Produktion()
…
!Call production routine by means of the type code
%“Prod_T“+ValToStr(MT_GetPartType());
…
ENDPROC
PROC Prod_T1()
…
ENDPROC
PROC Prod_T2()
…
ENDPROC
```

Depending on the part that has been assigned by the RWMT engine for the production, either the routine Prod_T1 or Prod_T2 are called.

Program execution

With the query `MT_GetPartType`, the part type code of the current part can be used for selecting the other program run.

Syntax

```
MT_GetPartType ´(´ ´)´
```

A function with a return value of the type dnum.

15.3.10 MT_GetToolCode – Current tool code

Usage

The `MT_GetToolCode` function reads the code of the current tool. The tool code is an integer >=0. If no tool code is agreed on, then the function returns -1.

The Toolcode is provided by a group input which has to be assigned in the process parameters of RWMT (see *MT Program Selection on page 159*).

Basic example

```
CONST partdata pdComponent8:=["Part with cycles",
"Production","", FALSE,1,8,3,27,[-1,-1,-1,-1,-1,-1,-1,-1],
     "Part8.GIF",[1.5,[0,0,0.001],[1,0,0,0],0,0,0],""];


PROC Production()
…
!Assign tool for production
IF MT_GetToolCode()=pdComponent8.Toolcode THEN
tCurrent := tTool8;

ELSEIF
…
ENDIF
…
ENDPROC
```

The apropriate tooldata is assigned for production.

Program execution

By specifying the tool code through `MT_GetToolCode`, the current code of the tool at the robot can be queried.

The appropriate tooldata can be assigned, when knowing the current tool code.

Syntax

```
MT_GetToolCode ´(´ ´)´
```

A function with a return value of the type num.

More information

Information about	See
Part types (Partdata)	*partdata – Part data on page 284*

15.3.11 MT_GhostModeActive – Ask if the ghost mode is active

Usage

Through the function `MT_GhostModeActive`, it is possible to determine whether a ghost mode is active currently.

A ghost mode should be provided always if the logical production process of the system is to be tested without real parts.

The ghost mode can be requested through the RWMT user interface (GUI) or by an external signal.

`MT_GhostModeActive` returns TRUE, if a ghost mode is present, otherwise it returns FALSE.

Basic example

```
PROC LoadMachine()
…
!Load the machine
LoadMachine;
!if real components are handled
IF MT_GhostModeActive()=FALSE THEN
!Start machine
StartMachine;
ENDIF
…
ENDPROC
```

If no ghost mode is present, that is, if the processing machine is actually loaded with a part, then this can be started for processing the part.

Program execution

The ghost mode can be recognized with the query MT_GhostModeActive and the program run modified accordingly.

Syntax

```
MT_GhostModeActive ´(´ ´)´
```

A function with a return value of the type bool.

15.3.12 MT_GripIsEmpty – Check if gripper is empty

Usage

`MT_GripIsEmpty` is used to check indirectly if a gripper is empty, by checking if the control element for closing the gripper, switching on the vacuum, , and so on. is reset.

Upto 6 control elements can be checked simultaneously for this.

Basic example

```
TASK PERS grpdata gdGripper:=
[
"gripper",1,TRUE,0.5,TRUE,TRUE,
["doCloseGripper",0,"doOpenGripper",0,
"Close Gripper","Open Gripper"],
["Gripper","diGripperClosed","diGripperOpen"],
["","",""],["","",""],["","",""]
];
```

Gripper with an output signal for closing and an output signal for opening. The gripper status is reported with an input for "Gripper is open" and an input for "Gripper is closed".

```
!Check, if gripper is empty
IF MT_GripIsEmpty(gdGripper\Output)=FALSE THEN
TPWrite “Gripper is not empty !”
Stop;
ENDIF
```

There is a query asking if the gripper is empty. For this, the "closed" response (`diGripperClosed`) must be low. Since the switch `\Output` is set, even the actuating output (`doClosegripper`) should be low. If the gripper is not empty, a message will be output and the program will be stopped.

Arguments

```
MT_GripIsEmpty Grp1 \Grp2 \Grp3 \Grp4 \Grp5 \Grp6 \Output
```

`Grp1`	Data type: `grpdata` First element that is to be checked.
`[\Grp2]`	Data type: `grpdata` Second element that is to be checked.
`[\Grp3]`	Data type: `grpdata` Third element that is to be checked.
`[\Grp4]`	Data type: `grpdata` Fourth element that is to be checked.
`[\Grp5]`	Data type: `grpdata` Fifth element that is to be checked.

Continues on next page

15.3.12 MT_GripIsEmpty – Check if gripper is empty
Continued

[\Grp6]	Data type: grpdata Sixth element that is to be checked.
[\Output]	Data type: switch Switch for checking even the output signals for actuating the actuators.

Program execution

With the check MT_GripIsEmpty, it is possible to determine indirectly if a part is present in the gripper, by checking the gripper messages (for example, "Gripper closed") and if necessary, even the actuating signals of the actuators (for example, "Close gripper") for the state low.

Syntax

```
MT_GripIsEmpty
'('
[Grp1 ':=' < expression (IN) of grpdata>]
['\' Grp2 ':=' < expression (IN) of grpdata>]
['\' Grp3 ':=' < expression (IN) of grpdata>]
['\' Grp4 ':=' < expression (IN) of grpdata>]
['\' Grp5 ':=' < expression (IN) of grpdata>]
['\' Grp6 ':=' < expression (IN) of grpdata>]
['\'Output]
')'
```

A function with a return value of the type bool.

More information

Information about	See
Data type gripper data	*grpdata – Configuration of a control element of the gripper on page 257*

15.3.13 MT_GripIsEmptyType – Check if gripper is empty

Usage

MT_GripIsEmptyType is used to check indirectly if a gripper is empty, by checking if the control element for closing the gripper, switching on the vacuum, , and so on. is reset.

Up to 6 control elements can be checked simultaneously for this.

MT_GripIsEmptyType provides mainly the same functionality as MT_GripIsEmpty but considers part type specific type numbers and type prefixes as follows:

There might be different grippers for each part type in the production cell. The grippers might work differently, thus each gripper will need its own grpdata declarations.

Instead of assigning the grpdata directly as this is done with MT_GripIsEmpty, a string is provided to MT_GripIsEmptyType which represents the name of the grpdata but without part type number and part type prefix.

MT_GripIsEmptyType will internally complete the grpdata name, depending on the current settings for the type prefix and the type number. Then the function will return the state for the appropriate type-depending grpdata declaration.

Basic example

Assuming, the current part type number is 6 and the standard part type prefix is “T”. Assuming a gripper with an output signal for closing and an output signal for opening. The gripper status is reported with an input for "Gripper is open" and an input for "Gripper is closed".

```
TASK PERS grpdata gdGripper_T6:=
[
"gripper",1,TRUE,0.5,TRUE,TRUE,
["doCloseGripper",0,"doOpenGripper",0,
"Close Gripper","Open Gripper"],
["Gripper","diGripperClosed","diGripperOpen"],
["","",""],["","",""],["","",""]
];

!Check, if gripper is empty
IF MT_GripIsEmpty("gdGripper”\Output)=FALSE THEN
TPWrite “Gripper is not empty !”
Stop;
ENDIF
```

There is a query asking if the gripper is empty (grpdata gdGripper_T6) . For this, the "closed" response (diGripperClosed) must be low. Since the switch \Output is set, even the actuating output (doCloseGripper) should be low. If the gripper is not empty, a message will be output and the program will be stopped.

Continues on next page

15.3.13 MT_GripIsEmptyType – Check if gripper is empty
Continued

Arguments

```
MT_GripIsEmptyType Grp1 [\Grp2] [\Grp3] [\Grp4] [\Grp5] [\Grp6]
[\Prefix] \Output
```

`Grp1`	Data type: `string` First element that is to be checked.
`[\Grp2]`	Data type: `string` Second element that is to be checked.
`[\Grp3]`	Data type: `string` Third element that is to be checked.
`[\Grp4]`	Data type: `string` Fourth element that is to be checked.
`[\Grp5]`	Data type: `string` Fifth element that is to be checked.
`[\Grp6]`	Data type: `string` Sixth element that is to be checked.
`[\Output]`	Data type: `switch` Switch for checking even the output signals for actuating the actuators.
`[\Prefix]`	Data type: `string` Assigns another part type prefix apart from the default prefix.

Program execution

With the check `MT_GripIsEmpty`, it is possible to determine indirectly if a part is present in the gripper, by checking the gripper messages (for example, "Gripper closed") and if necessary, even the actuating signals of the actuators (for example, "Close gripper") for the state low.

Syntax

```
MT_GripIsEmptyType
'('
[Grp1 ':=' < expression (IN) of string >]
['\' Grp2 ':=' < expression (IN) of string >]
['\' Grp3 ':=' < expression (IN) of string >]
['\' Grp4 ':=' < expression (IN) of string >]
['\' Grp5 ':=' < expression (IN) of string >]
['\' Grp6 ':=' < expression (IN) of string >]
['\'Output]
['\'Prefix':=' < expression (IN) of string>] ')'
```

A function with a return value of the type bool.

More information

Information about	See
Data type `gripper data`	*grpdata – Configuration of a control element of the gripper on page 257*

15.3.14 MT_JointCompare – Axis by axis comparison of two positions

Usage

The `MT_JointCompare` function compares the position that has been handed over, axis by axis, with the current position of the robot.

The permitted deviation can be specified for all the axes with the same value (Deviation) or it can be set separately for every axis (DevAx).

If the optional parameter `DevAx` is passed, the global value `Deviation` will not be considered.

If the difference between the angular settings of all the axes of the position that has been passed and the current position of the robot is less than the permitted deviation, then the value TRUE will be returned. If the deviation is larger, FALSE will be returned.

Basic example

```
VAR jointtarget jLimit:= [[5,5,4,2,2,2], [2,2,2,2,2,2]];
!Compare position pHome with global limit value for all axis
IF MT_JointCompare ( pHome, 2, tGripper) THEN
Tpwrite „Robot near home position“;
ENDIF
!Compare position p10 mit axis-specific limits:
IF
     MT_JointCompare(p10,0,\Limit:=jLimit,\tool:=tGripper\Wobj:=wPallet)
THEN
Tpwrite „Roboter near position p10“;
ENDIF
```

Arguments

```
MT_JointCompare ( Point Deviation \DevAx tool \ WObj )
```

`Point`	Data type: `robtarget` The position that is to be compared.
`Deviation`	Data type: `num` The maximum permitted deviation between the two positions for all the axes in degrees
`DevAx`	Data type: `jointtarget` The maximum permitted deviation between the two positions for all the axes in degrees.
`tool`	Data type: `tooldata` The tool with which the position that has been passed was programmed.
`[\WObj]`	Data type: `wobjdata` The work object with which the position that has been passed was programmed.

Syntax

```
MT_JointCompare'('
[Point:=’ ] < expression (IN) of robtarget> ','
[Deviation:=’ ] < expression (IN) of num>
```

Continues on next page

15.3.14 MT_JointCompare – Axis by axis comparison of two positions
Continued

```
['\' DevAx:=’ < expression (IN) of jointtarget> ]
[tool :=’ ] < persistent (PERS) of tooldata>
['\' WObj :=’ < persistent (PERS) of wobjdata> ]
')'
```

A function with a return value of the type bool.

15.3.15 MT_PosCompare – Determine linear deviation from a position

Usage

The MT_PosCompare function calculates the length of the vector between the robot position that has been handed over, and the current position.

If the length of the vector is greater than the maximum deviation that has been passed, then the function returns FALSE, else TRUE.

Basic example

```
CONST robtarget pHome:=…;
CONST num nDeviation:=10;
PERS tooldata tGripper:=…;
PERS wobjdata wTable:=…;

!If the robot is in the home position.
IF MT_PosCompare(pHome,nDeviation\tool:=tGripper\Wobj:=wTable)
THEN
!Start production
Production;
ELSE
TPWrite "The robot is not located in home position.“;
ENDIF
```

The MT_PosCompare function checks if the robot is present at a distance not exceeding 10 mm from its home position. If this is the case, then the production will be started, else an error message will be output.

Arguments

```
MT_PosCompare Point Deviation \tool \WObj
```

Point	Data type: robtarget Position, to which the linear distance from the current position is to be determined.
Deviation	Data type: num Maximum permitted linear distance from Point to the current position.
[\tool]	Data type: tooldata Tool, in which the linear distance is to be determined. If no tool has been specified, then tool0 will be assumed.
[\WObj]	Data type: wobjdata Work object, in which the linear distance is to be determined. If no work object has been specified, then wobj0 will be assumed.

Syntax

```
MT_PosCompare '('
[Point ':=' <expression (IN) of robtarget>]','
[Deviation ':=' <expression (IN) of num>]
['\' Tool:=' ] < persistent (PERS) of tooldata> ','
['\' Wobj:=' ] < persistent (PERS) of wobjdata> ')'
```

A function with a return value of the type bool.

15.3.16 MT_RecalcPoint – recalculating a position in a new coordinate system

Usage

The function `MT_RecalcPoint` recomputes a position, which has been programmed with a specific tool and work object, at a position with the same spatial position but with new tool and work object.

`MT_RecalcPoint` returns the new position as RobTarget.

Basic example

```
PERS tooldata tOld:=…;
PERS tooldata tNew:=…;
PERS wobjdata wOld:=…;
PERS wobjdata wNew:=…;
CONST robtarget pOld:=…;
VAR robtarget pNew:=…;
!Re-calculate position based on new tool data and workobject data
pNew:=MT_RecalcPoint(pOld,tOld,tNew,wOld,wNew);
```

Arguments

`MT_RecalcPoint ( Point tOld tNew wOld wNew)`

`Point`	Data type: `robtarget` The position to be re-calculated
`tOld`	Data type: `tooldata` The existing tool
`tNew`	Data type: `tooldata` The new tool
`wOld`	Data type: `wobjdata` The existing work object
`wNew`	Data type: `wobjdata` The new work object

Syntax

```
RecalcPoint'('
[Point:=’ ] < expression (IN) of robtarget> ','
[tOld :=’ ] < persistent (PERS) of tooldata> ','
[tNew :=’ ] < persistent (PERS) of tooldata> ','
[wOld :=’ ] < persistent (PERS) of wobjdata> ','
[wNew :=’ ] < persistent (PERS) of wobjdata > ')'
```

A function with a return value of the type robtarget.

15.3.17 MT_RelTCP – Moving (translation) and rotation of the tool coordinates

Usage

The function `MT_RelTCP` uses the tool data (tooldata), the values of the displacement and rotation that have been passed, to compute a new tool coordinate system, and returns this as tool data.

Basic example

```
!Tool with 45° rotation around the z-axis compared to the
!original tool
tGripper:=MT_RelTCP(tOriginal\Dz:=45);
```

Arguments

```
MT_RelTCP Tool \Dx \Dy \Dz \Rx \Ry \Rz
```

`Tool`	Data type: `tooldata` The name of the tool that is to be moved
`[\Dx]`	Data type: `num` Displacement value in the direction of the X-axis
`[\Dy]`	Data type: `num` Displacement value in the direction of the Y-axis
`[\Dz]`	Data type: `num` Displacement value in the direction of the Z-axis
`[\Rx]`	Data type: `num` Rotation value about the X-axis
`[\Ry]`	Data type: `num` Rotation value about the Y-axis
`[\Rz]`	Data type: `num` Rotation value about the Z-axis

Syntax

```
MT_RelTCP '('
[Tool:=’ ] < persistent (PERS) of tooldata>
[‘\’Dx ’:=’ <expression (IN) of num> ]
[‘\’Dy ’:=’ <expression (IN) of num> ]
[‘\’Dz ’:=’ <expression (IN) of num> ]
[’\’Rx ’:=’ <expression (IN) of num> ]
[’\’Ry ’:=’ <expression (IN) of num> ]
[’\’Rz ’:=’ <expression (IN) of num> ]
')'
```

A function with a return value of the type tooldata.

15.3.18 MT_RobotInHome – Checking whether the robot is in the home position.

Usage

`MT_RobotInHome` routine is used to check whether the robot is in the home position.

Basic example

```
PROC Test()
!
IF MT_RobotInHome() THEN
TPWrite “Robot is in home position”;
ELSE
TPWrite “Robot is not in home position”;
ENDIF
!
ENDPROC
```

Return value

Data type: `bool`

TRUE: Robot is in the home position.

FALSE: Robot is not in the home position.

Program execution

The digital input or output for the "In home position" signal, which is defined in the system parameters, is interrogated and the result returned.

If no signal has been defined or a virtual controller is used, checking takes place on the basis of a position comparison with a maximum permitted deviation of 1° in each axis.

Syntax

```
MT_RobotInHome’(’ ’)’
```

15.3.19 MT_StationIsEnabled – Checking station pre-selection for production

Usage

`MT_StationIsEnabled` is used to check if a Station (for example, a processing machine) has been selected for the production. The function returns TRUE, if a station has been pre-selected, otherwise it returns FALSE.

Basic example

```
!Declaration of cutting station
LOCAL PERS stationdata CUT_Station:=
["CUT","Cutting Tool","Station for Cutting",
"station-cutter.png","","CUT_sdiCuttInFwdPos",
"","",TRUE,FALSE,1.3];

PROC Production()
…
!If the cuting station has been pre-selected
IF MT_StationIsEnabled(CUT_Station) THEN
!Start cutting process
CutIt;
ELSE
…
ENDPROC
```

In the production routine, a query is run to determine if the cutting station has been selected for the current production. If this is the case, then the work piece will be cut, otherwise not.

Arguments

`MT_StationIsEnabled std`

`std`	Data type: `stationdata` The station, for which the query is to be run, to see if it has been pre-selected.

Program execution

With the check `MT_StationIsEnabled`, it is possible to determine if a station has been selected for the current production.

By pre-selecting the individual stations, the process of a production cycle can be modified to suit the requirements for processing a work piece.

Syntax

```
MT_StationIsEnabled '('
[ std ':=' ] < expression (IN) of stationdata> ')'
```

A function with a return value of the type bool.

Continues on next page

15.3.19 MT_StationIsEnabled – Checking station pre-selection for production
Continued

More information

Information about	See
Data type `station data`	*stationdata – Definition of a station on page 297*

16 Fault rectification (debugging)

16.1 Evaluation of the event log messages

No.	Error text	Causes and measures
119000	Prog no reading error	Read release was not granted. The external program number was not read. Check the program number handshake.
119001	Prog no reading error	Read release was not reset. The external program number was not read. Check the program number handshake.
119002	No service position feedback	Service position is not registered: Signal has not been set. Program waiting for message from the service position. Check the service position signal mentioned.
119003	No home position feedback	Home position is not reported. Signal has not been set. Program is waiting for message from the home position. Check the home position signal mentioned.
119004	Double use of program no	The program number is already in use. Error in partdata declaration. Program is waiting for new program number. Check the partdata declarations for double use of program numbers.
119005	Double use of program no	The program number is already in use. Error in menudata declaration. Program is waiting for new program number. Check the menudata declarations.
119006	Wrong operation mode	Conditions for production are not fulfilled. The mode of operation does not have the expected value. Program continues to check the conditions. Set the necessary mode of operation.
119007	Wrong tool code	Tool code is not correct. The tool code does not have the expected value. Create correct code for flanged tool. Allocation of the correct tool code in the part data.
119008	Wrong check code	Check code is not correct. The check code does not have the expected value. Provide correct check code. Allocation of the correct check code in the part data.
119009	Wrong operation mode	Conditions for service are not fulfilled. The mode of operation does not have the expected value. Set the necessary mode of operation.
119010	Routine not available	Dynamically called routine does not exist. Check if the routine call is correct. Check if the routine that has been called exists.
119011	No home position feedback	Robot not in home for first cycle. Signal has not been set. Move the robot to the home position first. Check the home position signal mentioned.

Continues on next page

No.	Error text	Causes and measures
119012	Air pressure missing	No pressurized air is reported. Signal has not been set. Program waits for pressurized air report. Robot will remain paused till the pressurized air is present again. Check the pressurized air system.
119013	Missing position signal	In the PROC.CFG, no position signal has been defined for ROBOT IN HOME/SAFEPOS/SERVICEPOS. In the system parameters PROC, configure a corresponding signal.
119014	Start position not allowed	The robot is not found in any of the start positions that are defined in the menu data. Check the parameter ValidPosition in the corresponding menudata declaration.
119015	User program cannot be called	The routine `MT_GetUserProgNo` has been selected, but without valid data for the program number or the routine that is to be called. It must be either passed a valid routine name or a valid program number. You may not pass both.
119016	No valid start position for robot	The robot is not located in the home position nor in the safe position. Hence, no production routines or service routines can be executed. Program continues to check the conditions. Make sure that the robot is either in the home position or in the safe position at the end of a production cycle.
119017	No part or service routine preselected	Neither a part type for production nor a service routine has been selected. Therefor no production- or service routine can be executed. Please choose a part type or service routine either on the FlexPendant or remote controlled, depending on your individual solution.
119018	Missing preselection of cycle	No cycle has been selected. The production routine cannot be executed since it needs a cycle selection first of all.
119019	No valid end position for robot	After execution of the service routine the robot is neither located in home nor in safe position.
119020	Instruction set fault	A data element is not defined in the instruction set array. Instruction will not be executed on switching the mode of operation! Change the data declaration in the specified array.
119021	Wrong signal type in instruction set	The type of a signal in the instruction set does not correspond to that in the system parameters. Instruction will not be executed on switching the mode of operation! Change the signal type of the specified signal in the instruction set (permitted values: DO,GO or AO).
119022	Data object not defined in instruction set	A signal is not defined in the system parameters. Instruction will not be executed on switching the mode of operation! Change the signal name of the specified signal in the instruction set or declare the signal in the system parameters.

Continues on next page

No.	Error text	Causes and measures
119023	Instruction set access error	A persistent in the specified task could not be described. Check if the specified persistent has been declared in the required task or if the name has been written correctly.
119024	Instruction set value error	The value of a persistent could not be converted for the corresponding data type. Check if the value specified corresponds to the data type.
119030	Logging-file system space critical	Logging is actually activated. in the file system. Logging will be stopped at a certain percentage of remaining space. Check if you can release disk space by deleting obsolete files.
119031	Logging-file system space not suffiencient	Logging is actually activated and has to be stopped because of insufficient disk space. Check if you can release disk space by deleting obsolete files.
119050	Event routine undefined	The routine that has been defined in the event does not exist. The routine will not be executed. Check if the specified routine has been declared locally or if the routine name or the module name matches with the declaration.
119100	Wrong tool position value	The position value for the tool function MT_GRIPSET should lie between 1 and 3! (1: Open, 2: Closed, 3: Reset) Change the position value and set the program pointer once again to the current MT_Gripset instruction to actuate the gripper function once again.
119101	Wrong gripper position value	The position value for the gripper function MT_GRIPCHECK should lie between 1 and 2! (1: Open, 2: Closed) Change the position value and set the program pointer once again to the current MT_Gripset instruction to actuate the gripper function once again.
119102	Wrong check value	Wrong check position. The check value for the gripper function MT_PARTCHECK should lie between 0 and 1. Change the check value and set the program pointer once again to the current MT_Gripset-instruction to actuate the gripper function once again.
119103	Wrong tool code	Tool code of the gripper does not correspond to the coding in the gripper data. The gripper function that has been called will not be executed! Check the tool code in the grpdata.
119104	Wrong tool code	Tool code of the gripper does not correspond to the coding in the part controls (grppart). The gripper function that has been called will not be executed! Check the tool code in the gripper data.

Continues on next page

No.	Error text	Causes and measures
119105	Wrong signal declaration	The digital output signal for actuating a control element is not present in the controls. Gripper function that has been called will not be executed. Please check the name of the signal in the grpdata.
119106	Wrong signal type	The signal for actuating a control element must be a digital output! The gripper function that has been called will not be executed. Please check the name of the signal in the grp data.
119107	Wrong signal declaration	The digital input signal for the gripper is not present in the controls. Gripper function that has been called will not be executed. Please check the name of the signal in the grp data.
119108	Wrong signal type	The feedback signal for the control element must be a digital input! Gripper function that has been called will not be executed Please check the name of the signal in the grp data.
119109	Wrong signal declaration	A digital input signal for the part controls is not present in the controls. Gripper function that has been called will not be executed. Please check the name of the signal in the grppart.
119110	Wrong signal type	The signals for the part controls must be digital inputs. Gripper function that has been called will not be executed. Please check the name of the signal in the grppart.
119111	Gripper sequence array fault	The size of the gripper sequence array lies outside the permitted range. Gripper function that has been called will not be executed. Change the size of the sequence array. Permitted array sizes: 1-20.
119112	Gripper sequence array fault	The size of gripper sequence array does not match the required standard dimension. When declaring a gripper sequence , the array dimension is limited to {5}, if the dimension is not explicitly defined by the optional parameter "ArraySize".
119113	No valid gripper	A gripper sequence should be passed for the execution. The gripper function that has been called will not be executed! Change the routine call MT_GrpSequence.
119114	Error in gripper sequence	A control element declaration does not exist in a gripper sequence. The gripper function that has been called will not be executed! Check and change the specified name of the variable of the concerned control element in the gripper sequence!

Continues on next page

No.	Error text	Causes and measures
119115	grpdata declaration not available	The grpdata declaration does not exist. Change variable name of the grpdata in the calling routine !
119116	grppart declaration not available	The grppart declaration does not exist. Change variable name of the grppart in the calling routine !
119900	Robot has not moved ! Position: xxxxxx	**Cause:** The position number is identical before and after calling the "MT_HomeRun" routine. **Action:** If the robot returns to the same position after executing a complete movement sequence (for example, loading a chute), the bMT_HomeRunCheckPos variable must be set to FALSE to disable checking (see chapter *Strategy for automatic movement into the home position on page 134*). If a movement is carried out in the opposite direction (for example, forward with mv10_11 and backwards with mv11_10), the number of intermediate positions may be wrong or the numbering may not correspond in both routines (see the section *Movement routines on page 110*).
119901	Invalid intermediate position number length. Only 6-8 digit position numbers allowed! Position: xxxxxx	**Action:** Check number of digits in specified intermediate positions. Intermediate position numbers may only have 6-8 digits.
119902	Intermediate position contains neither start nor destination position number. Start: xxx Destination: yyy Position: xxxyyyzz	**Cause:** An intermediate position must contain the start and destination position in the first 4-6 digits (movement path). Only a direction change is permitted for intermediate position numbers (for example, from destination to start), not a movement path change. **Action:** Adapt the intermediate position to the start and destination positions.  **Tip** If the error message is output in manual mode, the position check can be deactivated until the next restart by continuing the program instruction by instruction.

Continues on next page

No.	Error text	Causes and measures
119903	Each movement routine must contain a 2 or 3 digit start and end position. In the current and previously executed movement routines, the movement instruction with position number XX is missing!	**Cause:** Either the start position is missing in the current movement routine, or the destination position is missing in the previously executed routine. **Action:** Insert the start position into the current routine using the \NoMove argument and check whether the position number of the end position from the previously routine is identical to the start position of the current movement routine.  **Tip** If the error message is output in manual mode, the position check can be deactivated until the next restart by continuing the program instruction by instruction.
119904	The start and end positions of a movement routine must be 2 or 3-digit. A movement routine may not start or end with a 6 or 8-digit intermediate position number! Current position: ea2	**Cause:** Start position missing from movement routine. **Action:** Insert start position into current routine using the \NoMove.  **Tip** If the error message is output in manual mode, the position check can be deactivated until the next restart by continuing the program instruction by instruction.
119905	Start and end positions must have 2 or 3 digits and intermediate positions must have 6 or 8 digits.	**Cause:** Position number length invalid! **Action:** Check position number 1
119906	HomeposRunning position info: Start range: <> mm Current position: <> Spacing axes 1-6: <> mm Spacing axes 7-12: <> mm	If the robot is located outside the start area and therefore can only be moved directly into the home position, the data of the actual robot position are written to the error protocol (user log).
119907	The robot must be moved directly to the home position.	**Cause:** Robot cannot be moved automatically to home position! **Action:** You must switch to "Manual" mode to move the robot directly to the home position.
119908	Operating mode change required! Manual mode 250mm/s is required to move the robot directly to the home position.	**Cause:** Robot can only be moved directly to the home position.

Continues on next page

No.	Error text	Causes and measures
119909	Home position not reached! Please check monitoring of home position!	**Cause:** The monitoring signal for the home position is still active after the HomeRun. **Action:** Check signal for home position - controller restart may be required if the home position is monitored via a stationary world zone.
119910	Routine `MT_HomeDirect` does not exist !	**Cause:** The `MT_HomeDirect` routine is missing. **Action:** Create the `MT_HomeDirect` routine in accordance with the description in chapter *MT_HomeDirect – Movement directly to the home position on page 356*.
119911	Routine `MT_HomeRun` does not exist!	**Cause:** `MT_HomeRun` routine is missing. **Action:** Create the `MT_HomeRun` routine in accordance with the description in chapter *Strategy for automatic movement into the home position on page 134*.
119912	Reference to unknown routine in MT_MoveSync instruction. Position: xxxxxx Routine: sssss	**Cause:** The routine specified in `MT_MoveJSync` or `MT_MoveLSync` does not exist, or the passing parameters do not match the parameters of the routine. **Action:** Adapt the routine name or the passing parameters.
119913	Unable to call movement routine defined by position number when movement continued. Position: xxxxxx Routine: ssssss	**Cause:** No movement routine found when automatically continuing movement with `MT_ContHomeRun`. **Action:** Check intermediate position number in movement routine or HomeRun strategy.
119914	Backward movement is not possible! The movement routine that was called *mvXX_YY* does not exist !	**Cause:** The specified movement routine does not exist when moving backwards with `MT_MoveRoutine` or `MT_ContHomeRun` without parameter `\NoAutoBackw`. **Action:** Create the necessary routine in accordance with description in the chapter *Movement routines on page 110*. Modify the name of the movement routine when calling up the `MT_MoveRoutine` routine in the `MT_HomeRun` routine in the `HOME_USER.MOD` module, and do not execute the routine backwards. If you use movement routines with a type index (for example, mv10_20_1 or mv10_20_T1), please check whether the correct type index has been set in the system parameters.

Continues on next page

No.	Error text	Causes and measures
119915	Search instructions MT_SearchL cannot be executed backwards! Position: xxxxxx	**Cause:** The routine that is called by MT_MoveRoutine or MT_ContHomeRun that is to be executed backwards contains the MT_SearchL instruction. **Action:** Modify the intermediate position number or the routine call so that the movement routine is not executed backwards.
119916	Checking of the home position via the output was not deactivated after changing the home position (robtarget).	**Cause:** The home position has been changed, meaning that the stationary world zone does not signal the home position after moving to it. **Action:** Controller restart is required. The home position is now checked by means of an internal position comparison.
119917	The DO_IN_HOME output for monitoring the home position via a temporary world zone cannot be reset.	**Cause:** When the temporary world zone for monitoring the home position was created, the output remained active after the world zone was deleted. The subsequent reset of the output failed because the output was still active. **Action:** Check where the output is still being processed, and check the configuration of the output.
119918	The DO_IN_HOME output for monitoring the home position needed to be set to 0 before initialising the temporary world zone.	**Cause:** After deletion of the temporary world zone for monitoring the home position, the output was still set and had to be set to 0. **Action:** Check where the output is still being processed, and check the configuration of the output.
119919	Software option "608-1 World Zones" is required to monitor the monitoring of the home position via a temporary world zone!	**Cause:** A temporary world zone for monitoring the home position can only be used if the "608-1 World Zones" option is available in the robot system. **Action:** Create a new robot system with the "World Zones" option or deactivate the use of the world zone for the home position in the system parameters (PROC/MT_HOMERUN/ CreateWZone).
119920	Digital input DI_GO_HOME has not yet been reset!	**Cause:** The request to move to the home position is still not set after reaching the home position. **Action:** Reset the request or change the process system parameter WaitGoHomeLow to FALSE, so that the program does not wait for the change to "low".

Continues on next page

No.	Error text	Causes and measures
119921	The MT_ContHomeRun instruction can only be used within the HomeRun strategy!	**Reason:** The MT_ContHomeRun instruction was used in a program section that is not executed during a HomeRun. **Action:** Only use the MT_ContHomeRun instruction within the HomeRun strategy.
119922	The MT_MoveRoutine instruction can only be used within the HomeRun strategy!	**Cause:** The MT_MoveRoutine instruction was used in a program section that is not executed during a HomeRun. **Action:** Only use the MT_MoveRoutine instruction within the HomeRun strategy.
119924	The position number must be numeric	**Cause:** Conversion error when reading out the current position.

16.2 Logging the RWMT Engine actions

What the log does

The recording or logging of actions within the RWMT Engine provides an opportunity for an advanced error analysis, in case there are any unexpected problems while using the RWMT.

Therefor a log file will be created in the home directory of the current robotic system.

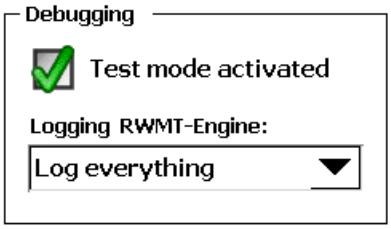
This log file contains advanced information that can be evaluated by XXX.

Activating and deactivating the log

The logging is activated or deactivated through the RWMT user interface. To do so, the robot icon should be selected first in the station view.

en1200000803

In the view that opens, now, the type of data that is to be logged should be determined in the "Debugging" area, using the check box:



en1300000179

The following filter options are available:

Filter	Data that is to be logged
Off	No information is recorded
Execution Loop	Only information about the min execution loop of the RWMT Engine, in which the higher order selection of the next action takes place, will be collected.
Execution Handling	Only information about the evaluation of the program requests or service requests to the RWMT Engine is recorded.
Cycle Selection Handling	Only information about the cycle handling is recorded.
Production Handling	Only information about the evaluation of production requests and the conditions associated with them is collected.
Service	Only information about the evaluation of service requests and the conditions associated with them is collected.
Program Number Handling	Only information pertaining to the transmission of an external program number will be recorded.
Log Everything	All the available information will be recorded.

Continues on next page

Executing the recording (logging)

A different filter is to be selected instead of the filter "Off". Now, as soon as the robot program is started, the recording in accordance with the selected filter.

To end the recording, the filter should be reset again to "Off".

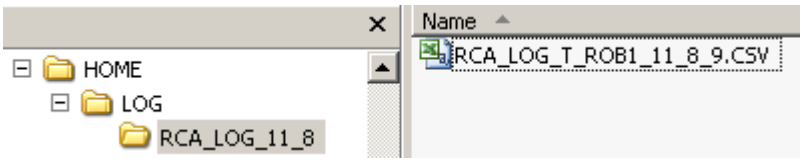
The log file is located in a director of the current system with the following convention for the names of the directory:

`HOME \ LOG \ MT_LOG_<Year>_<Month>`

The file itself has a name in accordance with the following convention:

`MT_LOG_<Taskname>_<Year>_<Month>_<Day>.CSV`

Example:



en1200000805

This file can be transferred with the help of RobotStudio to a local PC and viewed there, for instance, with the help of Microsoft Excel or any other software product, which can read the format of a CSV-file.

This page is intentionally left blank

Index

Contact us

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

ABB AS, Robotics
Discrete Automation and Motion
Nordlysvegen 7, N-4340 BRYNE, Norway
Box 265, N-4349 BRYNE, Norway
Telephone: +47 51489000

ABB Engineering (Shanghai) Ltd.
5 Lane 369, ChuangYe Road
KangQiao Town, PuDong District
SHANGHAI 201319, China
Telephone: +86 21 6105 6666

www.abb.com/robotics

3HAC044398-001, Rev A, en