ROBOTICS

Application manual

Continuous Application Platform

Trace back information:
Workspace 21A version a10
Checked in 2021-03-16
Skribenta version 5.4.005

Application manual

Continuous Application Platform

RobotWare 6.12

Document ID: 3HAC050990-001

Revision: D

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, damage 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.

This manual and parts thereof must not be reproduced or copied without ABB's written permission.

Keep for future reference.

Additional copies of this manual may be obtained from ABB.

Original instructions.

Table of contents

Overview of this manual

About this manual

This manual describes the option *Continuous Application Platform* and contains instructions for the configuration.

Who should read this manual?

This manual is intended for:

- Personnel responsible for installations and configurations of fieldbus hardware/software
- Personnel responsible for I/O system configuration
- System integrators

Prerequisites

The reader should have the required knowledge of:

- Mechanical installation work
- Electrical installation work
- System parameter configuration

References

References	Document ID
Application manual - Arc and Arc Sensor	*3HAC050988-001*
Application manual - Controller software IRC5	*3HAC050798-001*
Technical reference manual - RAPID Instructions, Functions and Data types	*3HAC050917-001*
Technical reference manual - RAPID Overview	*3HAC050947-001*

Revisions

Revision	Description
-	Released with RobotWare 6.0.
A	Released with RobotWare 6.01. Minor corrections.
B	Released with RobotWare 6.04. • Minor corrections. • The RAPID instructions, functions, and data types are moved to *Technical reference manual - RAPID Instructions, Functions and Data types*.
C	Released with RobotWare 6.05. • Minor corrections.
D	Released with RobotWare 6.12. • Updated information for `weavestartdata`, `cycle_time`. • The full information for CAP RAPID instructions, functions, and data types are added back to this manual.

Product documentation

Categories for user documentation from ABB Robotics

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

Tip

All documents can be found via myABB Business Portal, *www.abb.com/myABB*.

Product manuals

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

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

Technical reference manuals

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

Application manuals

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

An application manual generally contains information about:

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

Continues on next page

Operating manuals

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

Safety

Safety regulations

Before beginning mechanical and/or electrical installations, ensure you are familiar with the safety information in the product manuals for the robot.

The integrator of the robot system is responsible for the safety of the robot system.

1 Continuous Application Platform

Introduction

The Continuous Application Platform (CAP) consists of a number of RAPID instructions and data types that make development of continuous applications easier, faster, and more robust.

The basic idea of CAP is to separate motion synchronization and application control by moving the application control from the controller software kernel to an application layer in RAPID. By this two things are achieved:

- The CAP core is robust and generic.
- The application layer is easier to customize.

CAP is a process event dispatcher. The core of CAP is a generic state engine from which the application builder (developer) can choose to subscribe specific process events (`ICap`).

This page is intentionally left blank

2 Functionality of CAP

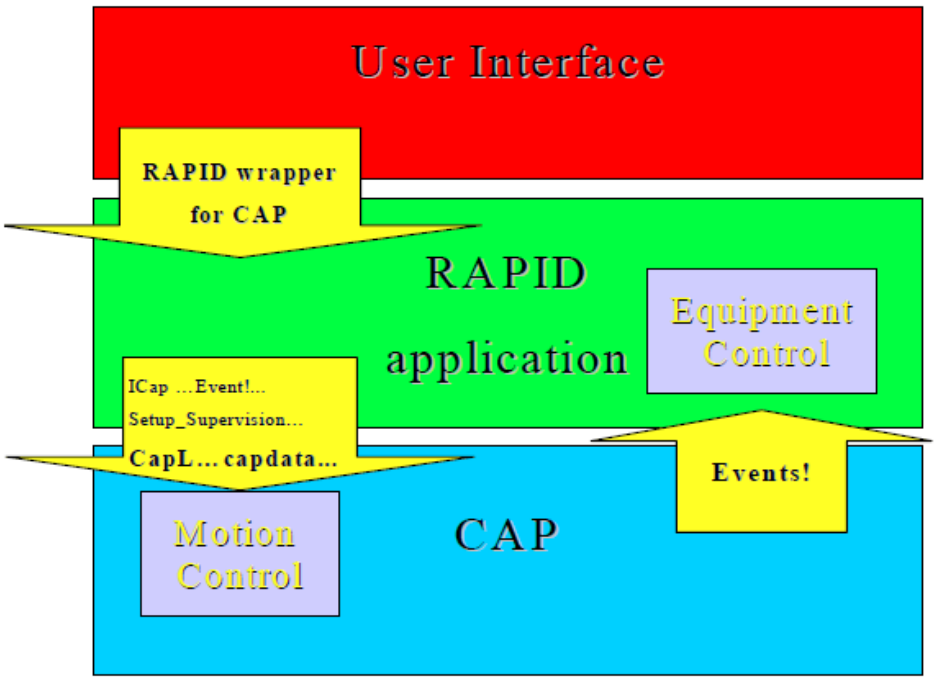
Description of CAP

With CAP it is possible to control a continuous application process and at the same time synchronize that process with the TCP movement of a robot.

The synchronization between motion and application layer is handled via events from the CAP core triggering trap routines in RAPID (*Predefined events on page 25*).

CAP enables the RAPID user to order supervision of I/O signals depending on the motion of the robot (*Supervision on page 16*).

For synchronization of motion and process, the process is divided into different phases. These phases tell the CAP core which I/O signals to supervise and in which way (*Process phases on page 14*).



xx1200000163

Continues on next page

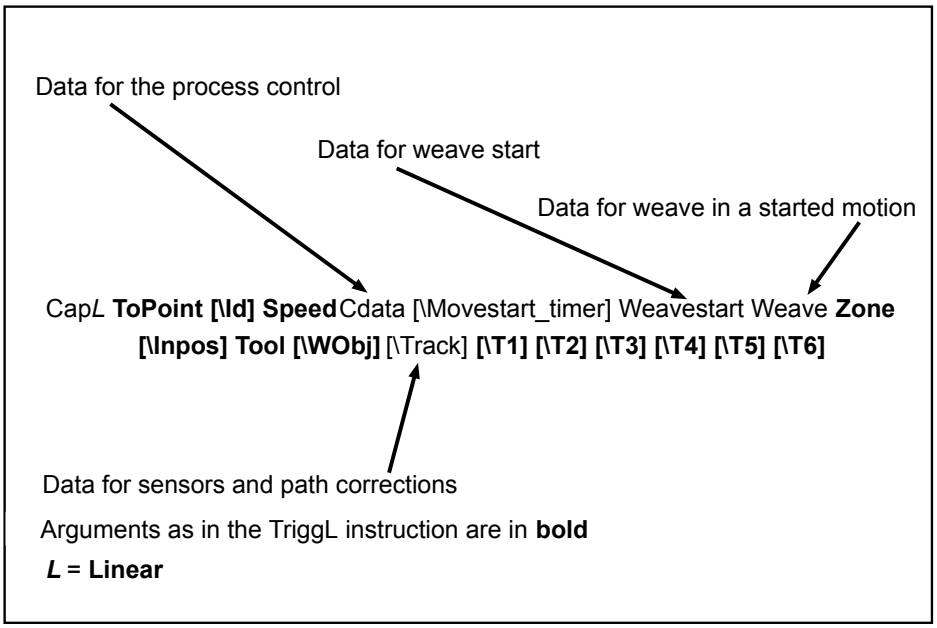
2.1 Robot movement

Instructions and motion

A CAP motion instruction (`CapL` or `CapC`) is similar to other motion instructions (for example, `MoveL`, `TriggL`). Compared to the `TriggL` instruction it contains also the information necessary for CAP. That information is given through the arguments `Cdata`, `Weavestart`, `Weave` and optional parameter `\Track`.

The motion control is handled by a state engine based on the general process graph class.

The CAP process is path persistent, which means that it is active over several CAP motion instructions from the first CAP instruction to the next CAP instruction containing the `Cdata.last_instr` set to `TRUE`, see *capdata - CAP data on page 108*.



xx1200000164

The `Speed` argument in the instruction is only valid during step-wise execution (forward or backward) and flying end from the point where the application process has been stopped. The CAP process will in this case automatically be inhibited. During normal execution, the process speed in different phases of the process is defined in the component `Cdata` of the type `capdata`.

For more information on programming CAP motion instructions see *Programming examples on page 39*.

Process phases

The CAP process is divided into four different phases to synchronize the robot movement with the application process:

- Pre_start
- Main

Continues on next page

- Post1
- Post2

The process phases are tightly bound to the supervision phases (*Supervision on page 16*).

During the process phases several events are generated by CAP, which can be connected to TRAP routines in the RAPID layer.

2.2 Supervision

Introduction to supervision

Supervision is used to control signals during the process and perform proper actions if some supervised signal fails.

Supervision is ordered at RAPID level, see *SetupSuperv - Setup conditions for signal supervision in CAP on page 101*.

Supervision phases

As mentioned in *Process phases on page 14*, the CAP process is divided in four phases. Those phases are tightly bound to the supervision phases. Supervision is divided in twelve supervision phases:

- PRE
- PRE_START (corresponds to CAP process phase Pre)
- END_PRE
- START
- MAIN (corresponds to CAP process phase Main)
- END MAIN
- START_POST1
- POST1 (corresponds to CAP process phase Post1)
- END_POST1
- START_POST2
- POST2 supervision (corresponds to CAP process phase Post2)
- END_POST2

The different supervision phases use corresponding supervision lists, which book-keep the signal supervisions ordered by the RAPID developer. The naming of the lists is similar to that of the corresponding phases, for example, the list for the PRE supervision phase is called SUPERV_PRE, see *SetupSuperv - Setup conditions for signal supervision in CAP on page 101*. The maximum number of signals that can be supervised during each phase is 32.

There are two different types of supervision phases:

- Handshake supervision.
- Status supervision.

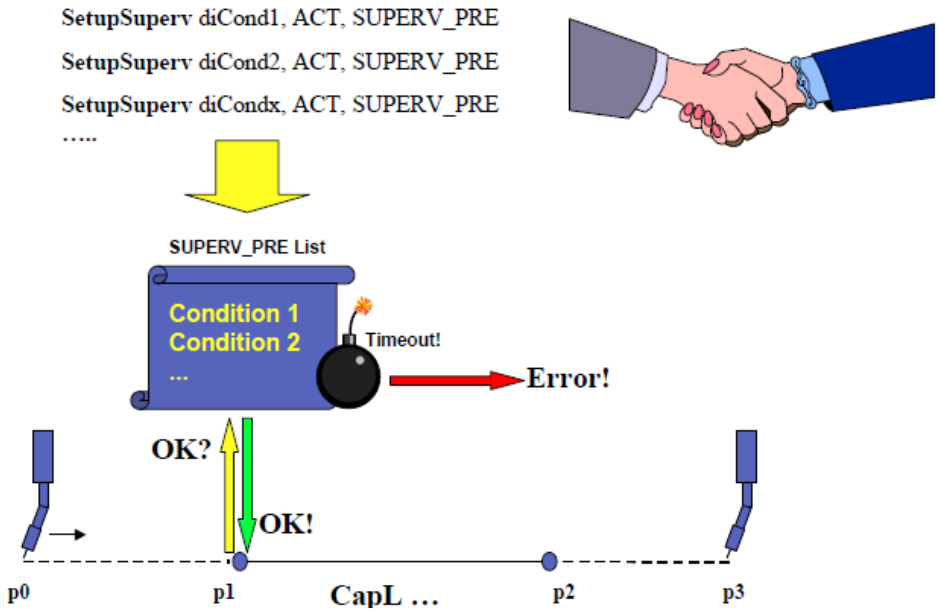
Handshake supervision

There is a handshake supervision phase before and after each status supervision phase. They prevent the process phase from starting until all conditions in the handshake supervision list are fulfilled.

It is possible to use a time-out for handshake supervision. If a time-out is used and not all supervision requests are fulfilled when it expires, an ERROR is generated. The time-out can be set to last forever, that is, the CAP process will be waiting for

Continues on next page

all supervision request to be fulfilled. The time-outs are specified in `supervtimeouts` which is part of the `capdata`.



xx1200000165

The handshake supervision works as a threshold, to prevent the process from proceeding to the next phase. If no handshake supervision is set up that phase is skipped.

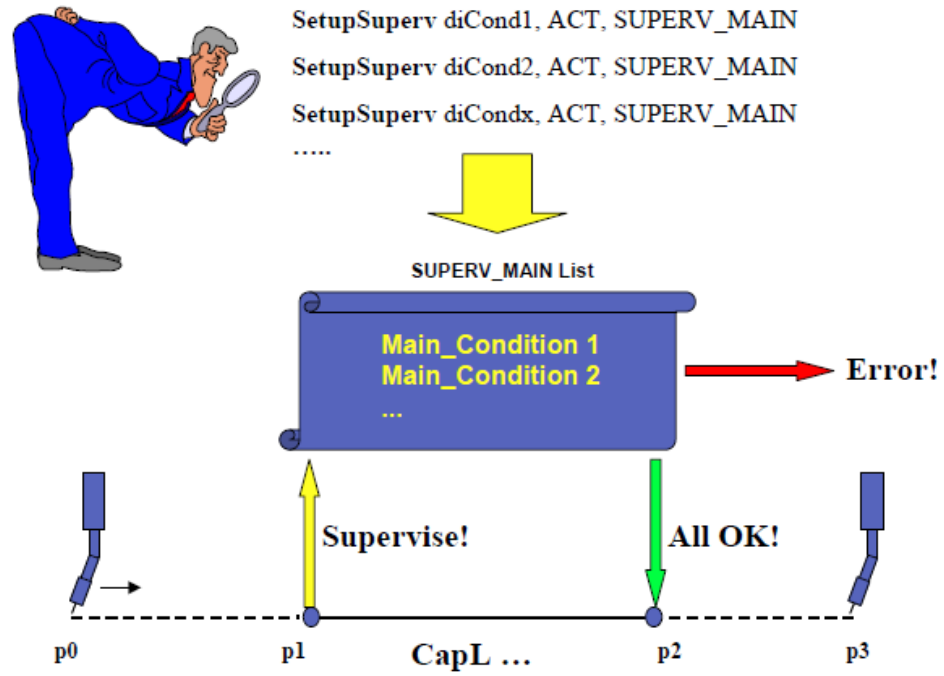
These are the handshake supervision phases.

- PRE
- END_PRE
- START
- END MAIN
- START_POST1
- END_POST1
- START_POST2
- END_POST2

Continues on next page

Status supervision

During the status supervision phases all signals the user requested supervision on (`SetupSuperv`) are supervised (see figure below).



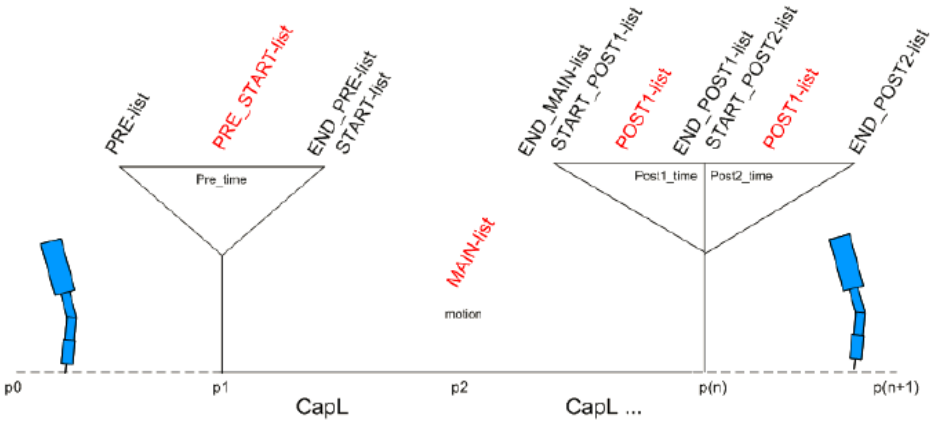
xx1200000166

The component `proc_times` in `capdata` defines the duration of the phases PRE_START, POST1, and POST2 will be. If supervision is requested during any of these phases, the duration time for each phase must be bigger than zero; otherwise the supervision will fail. No time has to be specified for the MAIN phase, because this time is defined by the movement of the robot.

These are the status supervision phases.

- PRE_START
- MAIN
- POST1
- POST2

Continues on next page



BLACK = Handshake Supervision list
RED = Status supervision list

xx1200000167

2.3 Supervision and process phases

Phases

The PRE_START phases, the POST1 phases and the POST2 phases are common to one single CAP process path, that is:

- the first CAP instruction that starts or restarts the CAP process is the only one that has PRE_START supervision phases. At restart the presence of these phases depends on the setting of pre_phase in the data type `restartblkdata`. See *restartblkdata - blockdata for restart on page 134*.
- the last CAP instruction (`last_instr` set to `TRUE` in `capdata`) that terminates the CAP process is the only one that has POST1 phases and POST2 phases. See *capdata - CAP data on page 108*.

PRE_START phases

When the robot reaches the start point of the path, all conditions in the PRE supervision list must be fulfilled for the process to be allowed to enter the PRE_START status supervision phase. If a time-out is specified and the conditions cannot be met within that time, the process is stopped, and an error is sent.

During the PRE_START phase all conditions defined in the PRE_START supervision list must be fulfilled. If some of these conditions fail, the process is stopped and an error message is sent.

After the PRE_START phase all conditions in the END_PRE supervision list must be fulfilled for the process to be allowed to enter the START handshake supervision phase. If a time-out is specified and the conditions cannot be met within that time, the process is stopped, and an error is sent.

When using *flying start* this phase will not be available, but the duration time can be used to create an ignition delay.

Summary

- Starts when all conditions in the PRE supervision list are met.
- Supervised by the PRE_START supervision list.
- Ends when all conditions in the END_PRE supervision list are met.

MAIN phases

All conditions in the START supervision list must be fulfilled for the process to be allowed to enter the MAIN status supervision phase. If a time-out is specified and the conditions cannot be met within that time, the process is stopped, and an error is sent.

During the MAIN phase all conditions defined in the MAIN supervision list must be fulfilled. If some of these conditions fail, the process is stopped and an error message is sent.

Continues on next page

All conditions in the END_MAIN supervision list must be fulfilled for the process to end the MAIN phases. If the conditions cannot be met within that time, the process is stopped, and an error is sent.

Summary

- Starts when all conditions in the START supervision list are met.
- Supervised by the MAIN supervision list.
- Ends when all conditions in the END_MAIN supervision list are met.

POST1 phase

All conditions in the START_POST1 supervision list must be fulfilled for the process to be allowed to enter the POST1 status supervision phase. If a time-out is specified for START_POST1 and the conditions cannot be met within that time, the process is stopped, and an error is sent.

During the POST1 phase all conditions defined in the POST1 supervision list must be fulfilled. If some of these conditions fail, the process is stopped and an error message is sent.

All conditions in the END_POST1 supervision list must be fulfilled for the process to end the POST1 phases. If the conditions cannot be met within that time, the process is stopped, and an error is sent.

This phase is not available for *flying start*.

Summary

- Starts when all conditions in the START_POST1 supervision list are met.
- Supervised by the POST1 supervision list.
- Ends when all conditions in the END_POST1 supervision list are met.

POST2 phase

All conditions in the START_POST2 supervision list must be fulfilled for the process to be allowed to enter the POST2 status supervision phase. If a time-out is specified for START_POST2 and the conditions cannot be met within that time, the process is stopped, and an error is sent.

During the POST2 phase all conditions defined in the POST2 supervision list must be fulfilled. If some of these conditions fail, the process is stopped and an error message is sent.

This phase is not available for *flying start*.

Summary

- Starts when all conditions in the START_POST2 supervision list are met.
- Supervised by the POST2 supervision list.
- Ends when all conditions in the END_POST2 supervision list are met.

2.4 Motion delay

Description

Motion delay gives the user the possibility to delay the start of the motion. This is useful for applications, for example laser cutting, where the movement cannot be started before the material has been burned through. The time for the motion delay is specified in `capspeeddata`. See *capspeeddata - Speed data for CAP on page 116*.

This functionality is not available for *flying start*.

2.5 Programming recommendations

Corner zones

A sequence of CAP instructions shall have corner zones (for example, z10) on the path.

For example:

```
MoveL p10,v100,fine,tool;
CapL p20,v50,cdata,nowvst,nowv,z20,tool;
CapC p30,p40,v50,cdata,nowvst,nowv,z20,tool;
CapL p50,v50,cdata,nowvst,nowv,z20,tool;
CapL p60,v50,cdata,nowvst,nowv,fine,tool;
MoveL p70,v100,fine,tool;
```

If the last movement instruction before the first CAP instruction has a zone, CAP will start the application process with a *flying start*.

If the last instruction of a sequence of CAP instructions has a zone, CAP will end the application process with a *flying end*.

Within a sequence of CAP instructions, avoid logical instructions that take long time. That will cause the errors *50024 Corner path failure* and *110013 Application process interrupted*, which means, that a corner zone is converted to a stop point, and the application process is interrupted and restarted with the next CAP instruction.

2.6 Program execution

Corner zones

If `last_instr` is set to `TRUE` in `capdata` in the middle of a sequence of CAP instructions, the application process is finished with all phases as described in *Process phases on page 14*. Which phases are executed depends on the presence of *flying end*. The following CAP instruction will start the process again, with all phases as described in *Process phases on page 14*. Which phases are executed, depends on the presence of *flying start*.

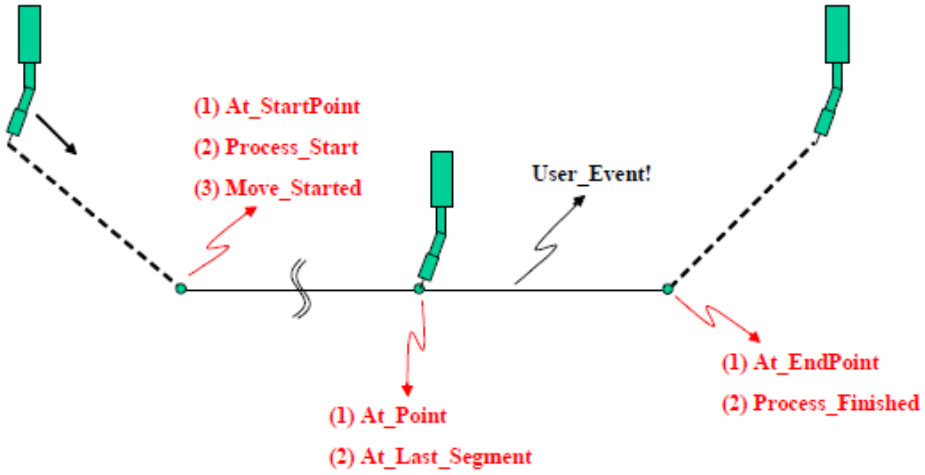
If a stop point occurs in the middle of a sequence of CAP instructions without `last_instr` set to `TRUE` in `capdata`, the application process will not be interrupted, the program execution will proceed to the next CAP instruction in advance (prefetch), and the movement will execute a stop point.

If execution of logical instructions in the middle of a sequence of CAP instructions take so long time, that a programmed corner path is converted to a stop point (*50024 Corner path failure*), the application process is interrupted (*110013 Application process interrupted*) without executing the phases described in *Process phases on page 14*.

2.7 Predefined events

Description

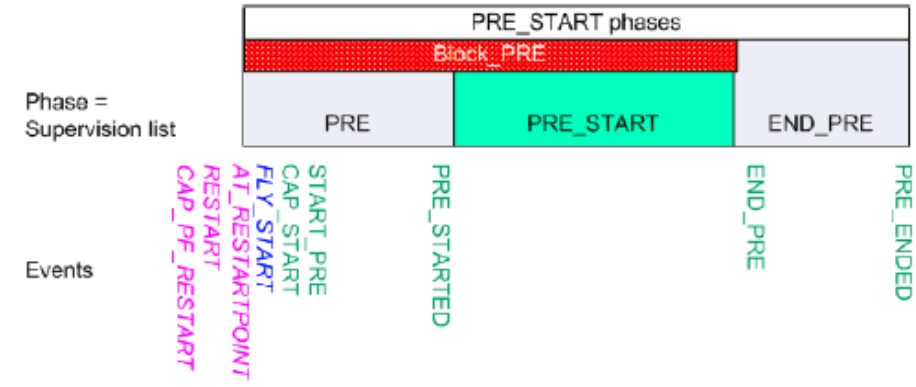
CAP can connect to RAPID interrupt routines with a number of predefined events, which occur during the CAP process. To do this, the RAPID instruction `ICap` is used before running the first CAP movement instruction. This enables the user to synchronize external equipment with the robot movement using RAPID. See *ICap - connect CAP events to trap routines on page 91*.



xx1200000168

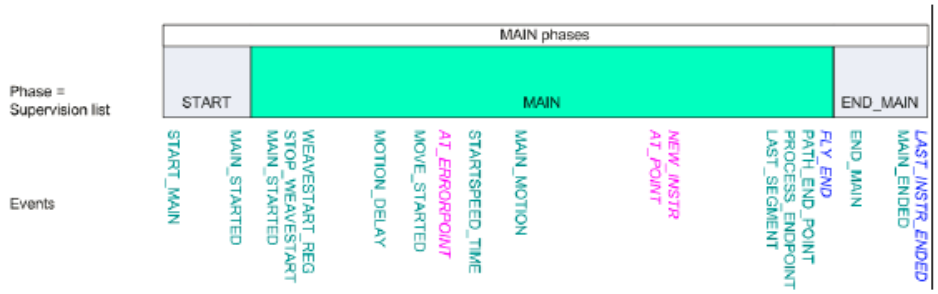
2.8 Coupling between phases and events

PRE_START supervision phases



xx1200000169

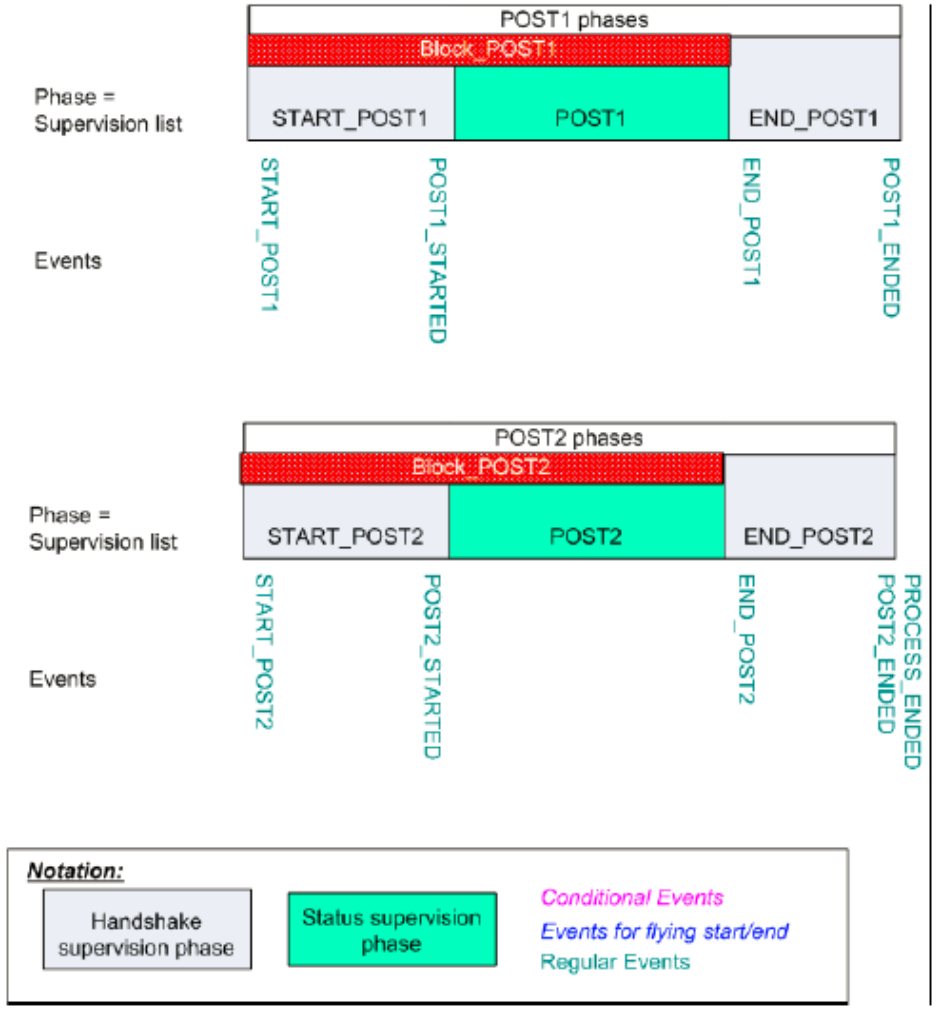
MAIN supervision phases



xx1200000170

Continues on next page

POST1 and POST2 supervision phases



xx1200000171

User events

The CAP movement instructions, `CapL` and `CapC`, offer the possibility to define up to six trigger events (switches `\T1` to `\T8`). These trigger events can be coupled to CAP movement instructions with `TriggIO`, `TriggEquip` or `TriggInt`. See *CapL - Linear CAP motion instruction on page 69* and *CapC - Circular CAP motion instruction on page 55*.

2.9 Error handling

Description

Two different types of error can occur during `CapL` or `CapC`:

- Recoverable errors: these errors can be handled in a RAPID error handler, see error handling for *CapL - Linear CAP motion instruction on page 69*. The system variable ERRNO is set and the user can check ERRNO to get information about which error occurred and choose the adequate recovery measures. For recoverable errors it is possible to use the RAPID instruction `RETRY` in the error handler. An error message is generated.
- Fatal errors: if such an error occurs, the robot controller has to be restarted. A fatal error message is generated.

Continues on next page

2.9.1 Recoverable errors

Introduction

Recoverable errors can be handled in a RAPID error handler. The CAP user can then choose to RETRY a several number of times depending on the application and the error. If for example, the arc in an arc welding application does not strike the first time, it makes sense to retry arc ignition a several number of times. If these attempts are unsuccessful the error may be raised to the next level of RAPID (RAISE) or, if it is in a NOSTEPIN/NOVIEW module, to user level (`RaiseToUser`) - see examples below.

Errors from CapL and CapC

Errors from the instructions `CapL` and `CapC` are CAP specific. See *CapL - Linear CAP motion instruction on page 69* and *CapC - Circular CAP motion instruction on page 55*. That means, that those error codes have to be translated to application specific error codes in the error handler, to make it easier for users of that application to understand the error message. After remapping of the error the new error code is raised to the user (RAISE new_err_code) - see *Example 1 on page 30*.

As a consequence these errors should be converted to application specific errors, depending on which application CAP is used for. To achieve this the `ERRNO` has to be checked in the error handler. See *Example 1 on page 30*.

If for example a supervised signal fails, for example, in the main supervision phase, the enduser should not get a general `CAP_MAIN_ERR` error. The application layer should return a more specific error, since this error depends on how CAP is used by the RAPID application. If several signals are supervised during a supervision phase, all these signals have to be checked in the application error handler to identify the error more specifically.

No error handler

If no error handler can be found or there is an error handler, but it does not handle the error - that is, none of the instructions `RETRY`, `TRYNEXT`, `RETURN` or `RAISE` are present in the error handler - the active motion path is cleared. That means, that neither *regain to path* nor *backing on the path* is possible. The robot movement starts from the current position of the TCP, which might result in a *path shortcut*.

Start phase supervision errors

If a START phase supervision error occurs during *flying start*, the movement is stopped at the end of the *START distance* and at restart the application process is handled like an ordinary restart after an error - with all user defined restart functionality like *scrape start*, *start delay*, etc.

Examples

Below are two examples of different error handling type. The type of error handling that is recommended is the one in example 2, where the CAP process survives and no extra code has to be executed in a retry from user level. See *Example 2 on page 31*.

Continues on next page

2.9.1 Recoverable errors
Continued

The SkipWarn instruction is used in the error handlers to prevent the CAP specific error from being sent to the error log. For an application (for example, Arc Welding) user CAP errors are not interesting. The errors shown in the event log shall be application specific.

Example 1

This is an example with RAPID modules, that are not NOSTEPIN/NOVIEW. If the error is RAISEd to the RAPID main routine the CAP process will exit.

A RETRY order in the error handler case MY_AW_ERR_1 will continue execution and make a retry on arcl_move_only. The value of example_count1 will be 2 when executing the CapL instruction after a retry from the higher level.

```
MODULE CAP_EXAMPLE1
VAR num example1_count:=0;
PROC main()
  MoveJ p10,v200,fine,tool0;
  arcl_move_only p11, v20, z20, tool0;
  ERROR
    TEST ERRNO
    CASE MY_AW_ERR_1:
      RETRY;
    CASE MY_AW_ERR_2:
      EXIT;
    DEFAULT:
      EXIT;
    ENDTEST
ENDPROC

LOCAL PROC arcl_move_only (robtarget ToPoint, speeddata Speed,
     zonedata Zone, PERS tooldata Tool \PERS wobjdata WObj \switch
     Corr)

example1_count := example1_count + 1;
CapL Topoint, Speed, IntCdata, IntWeavestart, IntWeave, Zone, Tool
     \wobj?wobj;
ERROR
  ResetIoSignals;

  IF no_of_retries > 0 THEN
    IF err_cnt < no_of_retries THEN
      err_cnt := err_cnt + 1;
      Skipwarn; !Remove CAP error from event log err_code :=
           new_aw_errMsg();
      RETRY; !*StartMoveRetry
    ELSE
      err_cnt := 0;
      Skipwarn;
      err_code := new_aw_errMsg();
      RAISE err_code;
      !Kills the CAP process, and raises mapped error
    ENDIF
```

Continues on next page

```
  ELSE
    Skipwarn;
    err_code := new_aw_errMsg();
    RAISE err_code;
      !Kills the CAP process, and raises mapped error
  ENDIF
ENDPROC

FUNC errnum new_aw_errMsg (\switch W)

  VAR errnum ret_code;

  TEST ERRNO

  CASE CAP_PRE_ERR:
    ! Check of signals here
    ret_code := AW_EQIP_ERR;
  ENDTEST

  RETURN ret_code;

ENDFUNC

ENDMODULE
```

Example 2

This is an example with one RAPID module, that is NOSTEPIN/NOVIEW, where main is located. The `arcl_move_only` procedure is located in a NOVIEW module. If the error is raised to main routine with the instruction `RaiseToUser \Continue` the CAP process is still active. The `RETRY` order in the error handler case MY_AW_ERR_1 will continue execution and make a retry directly on the `CapL` instruction a second time. The example_count1 will be 1 when executing the `CapL` instruction after a retry from the user level.

Note

The `RaiseToUser` instruction can only be used in a NOVIEW module.

```
MODULE CAP_EXAMPLE

VAR num example1_count:=0;

PROC main()
  MoveJ p10,v200,fine,tool0;
  arcl_move_only p11, v20, z20, tool0;

  ERROR
    TEST ERRNO
    CASE MY_AW_ERR_1:
      RETRY;
```

Continues on next page

2.9.1 Recoverable errors
Continued

```
    CASE MY_AW_ERR_2:
      EXIT;
    DEFAULT:
      EXIT;
    ENDTEST
ENDPROC
ENDMODULE

MODULE ARCX_MOVE_ONLY(NOSTEPIN, NOVIEW)
LOCAL PROC arcl_move_only(robtarget ToPoint, speeddata Speed,
     zonedata Zone, PERS tooldata Tool \PERS wobjdata WObj \switch
     Corr)
example1_count:=example1_count + 1;

  CapLTopoint, Speed, IntCdata, IntWeavestart, IntWeave, Zone, Tool
       \wobj?wobj;
ERROR
  ResetIoSignals;

  IF no_of_retries > 0 THEN
    IF err_cnt < no_of_retries THEN
      err_cnt := err_cnt + 1;
      Skipwarn;
      err_code := new_aw_errMsg();
      RETRY;
    ELSE
      err_cnt := 0;
      Skipwarn;
      err_code := new_aw_errMsg();
      RaiseToUser \Continue \ErrorNumber:=err_code;
    ENDIF
  ELSE
    Skipwarn;
    err_code := new_aw_errMsg();
    RaiseToUser \Continue \ErrorNumber:=err_code;
  ENDIF
ENDPROC

FUNC errnum new_aw_errMsg (\switch W)
  VAR errnum ret_code;
TEST ERRNO
  CASE CAP_PRE_ERR:
    ! Check of signals here
    ret_code := AW_EQIP_ERR;
  ENDTEST
  RETURN ret_code;
ENDFUNC
ENDMODULE
```

The `errnum` raised to the level above arcl_move_only is AW_EQIP_ERR, that is, the CAP error CAP_PRE_ERR is replaced by AW_EQIP_ERR and the CAP error will not appear in the error log (topic *Process*).

2.9.2 Writing a general error handler

General error handlers

For a CAP single system you can write an error handler as shown in examples 1 and 2. For a MultiMove system, errors must be handled as in example 3 below.

The error handlers for all tasks executing synchronized motion must contain the following:

For recoverable errors handled application internally

```
StorePath;
! Here you may have some application specific actions/movements
RestoPath;
StartMoveRETRY;
```

For errors handled by the user:

```
RaiseToUser \Continue \ErrorNumber:=error_code;
```

Not all instructions must be in all tasks, but some are important. `RETRY` must be changed to `StartMoveRETRY` in all RAPID tasks if running in synchronized mode. `StartMoveRETRY` restarts all path processes. For a task using the instruction `MoveExtJ` or other move instructions like `TriggX` or `MoveX` the `StartMoveRETRY` orders that mechanical unit to start again. The `RETRY` order works only on the `CapL`/`CapC` instructions.

Example 3

This example only shows the error handler.

```
IF err_cnt < no_of_retries THEN
  err_cnt := err_cnt + 1;
  ! Suppress CAP error message
  Skipwarn;
  ! Remap CAP error to application specific error
  err_code := new_aw_errMsg();
  ! Store current movement information
  StorePath;

  ! Here you may have some application specific actions/movements

  ! Restore movement information stored with previous StorePath
  RestoPath;
  ! Restart the movement
  StartMoveRETRY;
ELSE
  err_cnt := 0;
  ! Suppress CAP error message
  Skipwarn;
  ! Remap CAP error to application specific error
  err_code := new_aw_errMsg();
  ! Raise the application specific error to the RAPID user level
  RaiseToUser \Continue \ErrorNumber:=err_code;!***
ENDIF
```

Continues on next page

2.9.2 Writing a general error handler
Continued

> **Note**
>
> `StartMoveRETRY` is an instruction that combines `StartMove` and `RETRY`. A `StartMove` executed on `TriggX`, `MoveX` or `MoveExtJ` will order a restart of movement for that specific robot/additional axis. It is the `RETRY` that triggers CAP process to restart and order a restart of movement.

RaiseToUser

`RaiseToUser` can be used with `\Continue` or `\Breakoff`.

- `\Continue` will raise the error to user level (first level that is not in a NOVIEW module). A `RETRY` from that level will start the instruction that failed exactly where the error occurred - that is, it will not re-run the NOVIEW procedure as a whole.
- \Breakeoff will rerun the instruction where the program pointer is on user level.

If `RaiseToUser` is used, the user itself has to restart or enable restart of robot movement by using:

- `StartMove` or `StartMoveRETRY` will restart movement as soon as all synchronized robots have sent their restart orders.
- `StartMoveReset` enables manual restart (start button on the FlexPendant).

2.10 Restart

Description

If the execution of a CapL/CapC instruction is stopped due to an recoverable error or a program stop, a backing distance can be specified at restart of the instruction. The backing distance is a value the user must specify in the capdata data structure, see *capdata - CAP data on page 108*.

Sensors

Sensors can be used together with CAP. The different sensor systems implemented are At-PointTracker (for example, serial *WeldGuide*) and Look-Ahead-Tracker (for example, laser tracker). See also *Application manual - Controller software IRC5* as well as *CapL - Linear CAP motion instruction on page 69*, *CapC - Circular CAP motion instruction on page 55*, and *captrackdata - CAP track data on page 119*.

Units

In CAP the following units are used:

length	mm
time	s
speed	mm/s
angle	degree

Tuning

It is possible to change the value of (tune) the following data when it is active, during execution:

weavedata components:

- width
- height
- bias

capdata components:

- speeddata

Example

The example changes the main speed and weave within a TRAP.

```
VAR intnum intno0;

PROC main()
  IDelete intno0;
  CONNECT intno0 WITH MainMotionTrp;
  ICap intno0, MAIN_MOTION;
  CapL p11, v100, cdata1, weavestart, weave, z20, tool0;
ENDPROC

TRAP MainMotionTrp
  cdata1.speed_data.main := 23;
```

Continues on next page

```
    weave.width := 5;
ENDTRAP
```

Note

In this example the TRAP-routine is inside the main module. The recommendation is that all TRAP-routines should be executed by a background task.

2.11 System event routines

Introduction

CAP knows nothing about external equipment so the control of the equipment must be handled in shelf-hooks (stop-, start-, restart-, ...) or in `ICap` trap events, see *ICap - connect CAP events to trap routines on page 91*. Normally the `ICap` trap events are used to activate and deactivate equipment.

All errors and stops generated directly from the CAP source code in the controller software, can be handled in the `CAP_STOP ICap` event. The `CAP_STOP` trap event is executed as soon as CAP detects an error (fatal or recoverable) or if the RAPID program is stopped with an active CAP process, that is, a `CapX` instruction has been executed, but no `CapX` instruction with `last_instruction:=TRUE`, see *capdata - CAP data on page 108*. The procedure that should be run when a stop occurs must deactivate external equipment. The stop shelf is necessary to take the system to a fail-safe state, if anything unexpected happens in the controller software.

Exceptions

Not all errors can be handled in shelf-hooks or with the `CAP_STOP` trap. If the system, for some reason, is forced to system failure state, all execution of RAPID code is immediately stopped. To handle this situation, the signal definitions (in EIO.cfg) for signals that handle external equipment must contain the flag `-SysfailReset`, which will set the signal to its default value (0 or defined by the flag `-default`). We recommend defining those signals with `CapCondSetDO` to be set to a defined status, when TCP movement stops, see *CapCondSetDO - Set a digital output signal at TCP stop on page 65*.

2.12 Limitations

Limitations

- Problems can occur if much logging to files is added in CAP and user event trap routines. The reason is that file writing takes long time and will delay the execution of the next instruction. That may cause corner path failure, stopping the movement of the robot for a short time, which may be fatal for the process (for example, arc welding).
- CAP does not support error recovery with long jump.

3 Programming examples

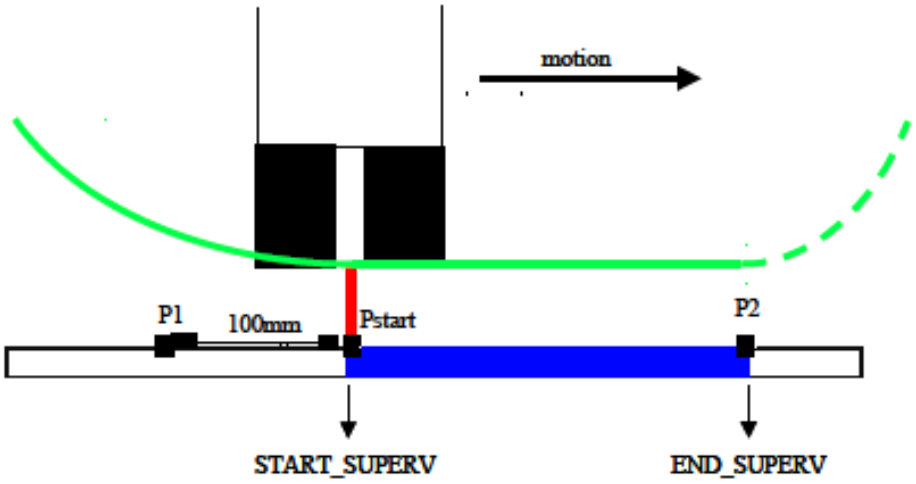
3.1 Laser cutting example

Requirements

- a slot is to be cut into a number of metal sheets with a laser
- at the starting point of the slot accuracy is not critical
- at the finishing point of the slot accuracy is required
- the application is time critical, that is, it has to be as fast as possible

CAP setup to meet the requirements

- *flying start*: the robot can move with speed past the start point (P1) and start the process on the fly between point P1 and Pstart.
- *normal end*: the robot must cut all the way to the end point (P2) and stop before turning off the laser and moving on to next cycle.
- In order to assure the quality of the cuts the process needs to be started at the latest one second after passing Pstart. Three seconds are given for ending the process.



xx1200000172

3.2 Step by step

Set up CAP events

First there is a need to set up the events that are going to be ordered from CAP. For this application a minimum of two events are needed: START_SUPERV given at the Pstart point for starting the process. END_SUPERV given at the end point for turning off the process. That is done the following way:

```
VAR intnum start_intno:=0;
VAR intnum end_intno:=1;

TRAP start_trap
  SetDo doLaserOn, high;
ENDTRAP

TRAP end_trap
  SetDo doLaserOn, low;
ENDTRAP

IDelete start_intno;
IDelete end_intno;
CONNECT start_intno WITH start_trap;
CONNECT end_intno WITH end_trap;
ICap start_intno, START_MAIN;
ICap end_intno, END_MAIN;
```

Set up supervision

In this case only one signal, *diLaserOn*, needs to be supervised, but in three different process phases:

1. *diLaserOn* goes high (ACT) in the START phase.
2. *diLaserOn* stays high (that is, triggers on PAS) during the MAIN phase.
3. *diLaserOn* goes low (PAS) in the END phase.

Furthermore we need to setup the start point for supervision of the START phase, and setup the time-out timers for the START and the END phase.

```
SetupSuperv diLaserOn, ACT, SUPERV_START;
SetupSuperv diLaserOn, ACT, SUPERV_MAIN;
SetupSuperv diLaserOn, PAS, SUPERV_END_MAIN;

capdata.start_fly_point.process_dist := 0;
capdata.start_fly_point.distance := 100;
capdata.sup_timeouts.start_cond := 1;
capdata.end_fly_point.process_dist := 0;
capdata.end_fly_point.distance := 0;
capdata.sup_timeouts.end_main_cond := 3;
```

Continues on next page

The main program

The user might use an encapsulation of `CapL` and call it for example `CutL`, which might be defined as follows:

```
PROC CUTL (...)
  MoveL p1, v100, z10,...
  CapL p2, v100, cdata, startweave, weave, fine, tool0, ...
  MoveL px, ...
ENDPROC
```

This page is intentionally left blank

4 RAPID references

4.1 Instructions

4.1.1 CapAPTrSetup - Setup an At-Point-Tracker

Usage

`CapAPTrSetup` (*Setup an At-Point-Tracker*) is used to setup an At-Point-Tracker type of sensor, for example, *WeldGuide* or AWC.

Basic example

SIO.cfg:

```
COM_TRP:
-Name "wg:" -Type "SOCKETDEV" -RemoteAddress "192.168.1255.101"
     -RemotePort "6344"
```

RAPID code:

```
! Define variable numbers
CONST num SensorOn := 6;
CONST num XCoord := 8;
CONST num YCoord := 9;
CONST num ZCoord := 10;
VAR pos SensorPos;

! Setup a Weldguide
CapAPTrSetup "wg:", do_left, 80, do_right, 80;
```

Arguments

```
CapAPTrSetup device DoLeft LevelLeft DoRight LevelRight [\LogFile]
     [\LogSize]
```

device

Data type: `string`

The I/O device name configured in sio.cfg for the sensor used.

DoLeft

Data type: `signaldo`

Digital output signal for weave synchronization on the left weave cycle.

LevelLeft

Data type: `num`

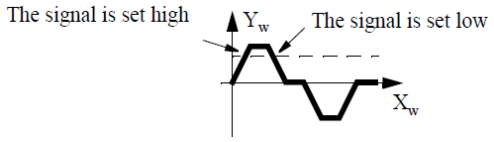
The coordination position on the left side of the weaving pattern. The value specified is a percentage of the width on the left of the weaving center. When weaving is carried out beyond this point, a digital output signal is automatically set high (provided the signal is defined).

Continues on next page

4.1.1 CapAPTrSetup - Setup an At-Point-Tracker

Continuous Application Platform (CAP)
Continued

This type of coordination can be used for seam tracking using Through-the-Arc Tracker.



xx1200000178

DoRight

Data type: `signaldo`

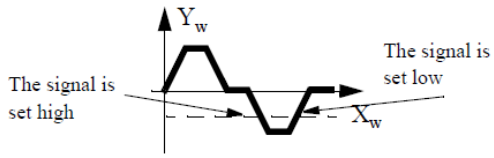
Digital output signal for weave synchronization on the right weave cycle.

LevelRight

Data type: `num`

The coordination position on the right side of the weaving pattern. The value specified is a percentage of the width on the right of the weaving center. When weaving is carried out beyond this point, a digital output signal is automatically set high (provided the signal is defined).

This type of coordination can be used for seam tracking using Through-the-Arc Tracker.



xx1200000179

[\LogFile]

Data type: `string`

Name of tracklog log file.

[\LogSize]

Data type: `num`

Size of the tracklog ring buffer, that is the number of sensor measurements that can be buffered during tracking.

Default value: 1000.

Syntax

```
CapAPTrSetup
  [device ':='] < expression (IN) of string> ','
  [DoLeft ':='] < expression (IN) of signaldo > ','
  [LevelLeft ':='] < expression (IN) of num > ','
  [DoRight ':='] < expression (IN) of signaldo > ','
  [LevelRight ':='] < expression (IN) of num >
  ['\'LogFile ':='] < expression (IN) of string >
  ['\'LogSize ':='] < expression (IN) of num > ';'
```

Continues on next page

Related information

	Described in:
Sensor Interface	*Application manual - Controller software IRC5*

4.1.2 CapAPTrSetupAI - Setup an At-Point-Tracker controlled by analog input signals

Usage

`CapAPTrSetupAI` is used to setup an At-Point-Tracker controlled by analog input signals.

Basic examples

The following example illustrates the instruction `CapAPTrSetupAI`.

Example 1

```
TASK PERS capdata cData:=[.....];
TASK PERS weavestartdata wsData:=[.....];
TASK PERS capweavedata wData:=[.....];
TASK PERS captrackdata trackData:=["ANALOG_TRACKER",.....];

VAR capaptrreferencedata referenceData:=[2,2,1,1,0.1,0.1];
VAR signalai ai_y;
VAR signalai ai_z;

AliasIO realsignal_y, ai_y;
AliasIO realsignal_z, ai_z;
CapAPTrSetupAI ai_y, ai_z, referenceData;

CapL p1, v200, cData, wsData, wData , fine, tWeldGun
     \Track:=trackData;
```

Arguments

```
CapAPTrSetupAO ai_y, ai_z, ReferenceData [\MaxIncrCorr]
     [\WarnMaxCorr] [\Filter] [\SampleTime] [\Logfile] [\LogSize]
     [\LatestCorr] [\AccCorr]
```

`ai_y`

Data type: `signalai`

Analog input signal used as process position for the y-direction.

`ai_z`

Data type: `signalai`

Analog input signal used as process position for the z-direction.

`ReferenceData`

Data type: `capaptrreferencedata`

Setup data used for the correction regulator loop.

`MaxIncCorr`

Data type: `num`

Maximum incremental correction allowed (in mm).

If the incremental TCP correction is larger than `\MaxIncCorr` and `\WarnMaxCorr`, the robot will continue its path but the applied incremental correction will not exceed

Continues on next page

\MaxIncCorr. If \WarnMaxCorr is not specified, a track error is reported and the program execution is stopped.

WarnMaxCorr

Data type: switch

If this switch is present the program execution is not interrupted when the limit for maximum correction is exceeded, specified in \MaxIncCorr. Only a warning is sent.

Filter

Data type: num

Size of the reference sample data filter. A value between 1 and 15 is allowed, the default value is 1.

SampleTime

Data type: num

Sample time in milliseconds for the correction loop. The value is rounded to a multiple of 24. The minimum value allowed is 24, and the default value is 24.

LogFile

Data type: string

The name of the tracklog log file. The log file is placed in the HOME directory of the system.

LatestCorr

Data type: pos

Size of the latest added correction (in mm).

AccCorr

Data type: pos

Size of the total accumulated correction added (in mm).

LogSize

Data type: num

The size of the tracklog ring buffer that is the number of sensor measurements that can be buffered during tracking.

Default value: 1000.

Syntax

```
CapAPTrSetupAI
  [aoi_y ':='] <expression (IN) of signalai> ','
  [ai_z ':='] <expression (IN) of signalai> ','
  [ReferenceData ':='] <expression (IN) of capaptrreferencedata>
       ','
  [\MaxIncrCorr ':='] <expression (IN) of num> ','
  [\WarnMaxCorr ':='] <expression (IN) of switch> ','
  [\Filter ':='] <expression (IN) of num> ','
  [\SampleTime ':='] <expression (IN) of num> ','
  [\LogFile ':='] <expression (IN) of string> ','
```

Continues on next page

4.1.2 CapAPTrSetupAI - Setup an At-Point-Tracker controlled by analog input signals

Continuous Application Platform (CAP)
Continued

```
[\LogSize ':='] <expression (IN) of num> ','
[\LatestCorr ':='] <expression (PERS) of pos> ','
[\AccCorr ':='] <expression (PERS) of pos> ';'
```

Related information

For information about	See
Instruction `CapAPTrSetupAO`	*CapAPTrSetupAO - Setup an At-Point-Tracker controlled by analog output signals on page 49*
Instruction `CapAPTrSetupPERS`	*CapAPTrSetupPERS - Setup an At-Point-Tracker controlled by persistent variables on page 52*
Data type `capaptrreferencedata`	*capaptrreferencedata - Variable setup data for At-Point-Tracker on page 106*
Sensor Interface	*Application manual - Controller software IRC5*

4.1.3 CapAPTrSetupAO - Setup an At-Point-Tracker controlled by analog output signals

Usage

`CapAPTrSetupAO` is used to setup an At-Point-Tracker controlled by analog output signals.

Basic examples

The following example illustrates the instruction `CapAPTrSetupAO`.

Example 1

```
TASK PERS capdata cData:=[.....];
TASK PERS weavestartdata wsData:=[.....];
TASK PERS capweavedata wData:=[.....];
TASK PERS captrackdata trackData:=["ANALOG_TRACKER",.....];

VAR capaptrreferencedata referenceData:=[2,2,1,1,0.1,0.1];
VAR signalao ao_y;
VAR signalao ao_z;

AliasIO realsignal_y, ao_y;
AliasIO realsignal_z, ao_z;
CapAPTrSetupAO ao_y, ao_z, referenceData;

CapL p1, v200, cData, wsData, wData , fine, tWeldGun
     \Track:=trackData;
```

Arguments

```
CapAPTrSetupAO ao_y, ao_z, ReferenceData [\MaxIncrCorr]
     [\WarnMaxCorr] [\Filter] [\SampleTime] [\Logfile] [\LogSize]
     [\LatestCorr] [\AccCorr]
```

`ao_y`

Data type: `signalao`

Analog output signal used as process position for the y-direction.

`ao_z`

Data type: `signalao`

Analog output signal used as process position for the z-direction.

`ReferenceData`

Data type: `capaptrreferencedata`

Setup data used for the correction regulator loop.

`MaxIncCorr`

Data type: `num`

Maximum incremental correction allowed (in mm).

If the incremental TCP correction is larger than `\MaxIncCorr` and `\WarnMaxCorr`, the robot will continue its path but the applied incremental correction will not exceed

Continues on next page

4.1.3 CapAPTrSetupAO - Setup an At-Point-Tracker controlled by analog output signals

Continuous Application Platform (CAP)
Continued

\MaxIncCorr. If \WarnMaxCorr is not specified, a track error is reported and the program execution is stopped.

WarnMaxCorr

Data type: switch

If this switch is present the program execution is not interrupted when the limit for maximum correction is exceeded, specified in \MaxIncCorr. Only a warning is sent.

Filter

Data type: num

Size of the reference sample data filter. A value between 1 and 15 is allowed, the default value is 1.

SampleTime

Data type: num

Sample time in milliseconds for the correction loop. The value is rounded to a multiple of 24. The minimum value allowed is 24, and the default value is 24.

LogFile

Data type: string

The name of the tracklog log file. The log file is placed in the HOME directory of the system.

LogSize

Data type: num

The size of the tracklog ring buffer that is the number of sensor measurements that can be buffered during tracking.

Default value: 1000.

LatestCorr

Data type: pos

Size of the latest added correction (in mm).

AccCorr

Data type: pos

Size of the total accumulated correction added (in mm).

Syntax

```
CapAPTrSetupAO
  [ao_y ':='] <expression (IN) of signalao> ','
  [ao_z ':='] <expression (IN) of signalao> ','
  [ReferenceData ':='] <expression (IN) of capaptrreferencedata>
       ','
  [\MaxIncrCorr ':='] <expression (IN) of num> ','
  [\WarnMaxCorr ':='] <expression (IN) of switch> ','
  [\Filter ':='] <expression (IN) of num> ','
  [\SampleTime ':='] <expression (IN) of num> ','
  [\LogFile ':='] <expression (IN) of string> ','
```

Continues on next page

```
[\LogSize ':='] <expression (IN) of num> ','
[\LatestCorr ':='] <expression (PERS) of pos> ','
[\AccCorr ':='] <expression (PERS) of pos> ';'
```

Related information

For information about	See
Instruction `CapAPTrSetupAI`	*CapAPTrSetupAI - Setup an At-Point-Tracker controlled by analog input signals on page 46*
Instruction `CapAPTrSetupPERS`	*CapAPTrSetupPERS - Setup an At-Point-Tracker controlled by persistent variables on page 52*
Data type `capaptrreferencedata`	*capaptrreferencedata - Variable setup data for At-Point-Tracker on page 106*
Sensor Interface	*Application manual - Controller software IRC5*

4.1.4 CapAPTrSetupPERS - Setup an At-Point-Tracker controlled by persistent variables

Usage

`CapAPTrSetupPERS` is used to setup an At-Point-Tracker controlled by persistent variables.

Basic examples

The following example illustrates the instruction `CapAPTrSetupPERS`.

Example 1

```
TASK PERS capdata cData:=[.....];
TASK PERS weavestartdata wsData:=[.....];
TASK PERS capweavedata wData:=[.....];
TASK PERS captrackdata trackData:=["ANALOG_TRACKER",.....];
PERS pos corr:=[0,-0.05,-0.025];

VAR capaptrreferencedata referenceData:=[2,2,1,1,0.1,0.1];

IDelete intno1;
CONNECT intno1 WITH trOffset;
CapAPTRSetupPERS corr.y, corr.z, referenceData;

ITimer 1,intno1;
CapL p1, v200, cData, wsData, wData , fine,
     tWeldGun\Track:=trackData;

TRAP trOffset
  corr.y := referenceData.reference_y +- .....;
  corr.z := referenceData.reference_z +- .....;
ENDTRAP
```

Arguments

```
CapAPTrSetupPERS var_y, var_z, ReferenceData [\MaxIncrCorr]
     [\WarnMaxCorr] [\Filter] [\SampleTime] [\Logfile] [\LogSize]
     [\LatestCorr] [\AccCorr]
```

`var_y`

Data type: `num`

Analog input signal used as process position for the y-direction.

`var_z`

Data type: `signalai`

Analog input signal used as process position for the z-direction.

`ReferenceData`

Data type: `capaptrreferencedata`

Setup data used for the correction regulator loop.

Continues on next page

`MaxIncCorr`

Data type: `num`

Maximum incremental correction allowed (in mm).

If the incremental TCP correction is larger than `\MaxIncCorr` and `\WarnMaxCorr`, the robot will continue its path but the applied incremental correction will not exceed `\MaxIncCorr`. If `\WarnMaxCorr` is not specified, a track error is reported and the program execution is stopped.

`WarnMaxCorr`

Data type: `switch`

If this switch is present the program execution is not interrupted when the limit for maximum correction is exceeded, specified in `\MaxIncCorr`. Only a warning is sent.

`Filter`

Data type: `num`

Size of the reference sample data filter. A value between 1 and 15 is allowed, the default value is 1.

`SampleTime`

Data type: `num`

Sample time in milliseconds for the correction loop. The value is rounded to a multiple of 24. The minimum value allowed is 24, and the default value is 24.

`LogFile`

Data type: `string`

The name of the tracklog log file. The log file is placed in the HOME directory of the system.

`LatestCorr`

Data type: `pos`

Size of the latest added correction (in mm).

`AccCorr`

Data type: `pos`

Size of the total accumulated correction added (in mm).

`LogSize`

Data type: `num`

The size of the tracklog ring buffer that is the number of sensor measurements that can be buffered during tracking.

Default value: 1000.

Syntax

```
CapAPTrSetupPERS
  [var_y ':='] <expression (PERS) of num> ','
  [var_z ':='] <expression (PERS) of vnum> ','
```

Continues on next page

4.1.4 CapAPTrSetupPERS - Setup an At-Point-Tracker controlled by persistent variables

Continuous Application Platform (CAP)
Continued

```
[ReferenceData ':='] <expression (IN) of capaptrreferencedata>
     ','
[\MaxIncrCorr ':='] <expression (IN) of num> ','
[\WarnMaxCorr ':='] <expression (IN) of switch> ','
[\Filter ':='] <expression (IN) of num> ','
[\SampleTime ':='] <expression (IN) of num> ','
[\LogFile ':='] <expression (IN) of string> ','
[\LogSize ':='] <expression (IN) of num> ','
[\LatestCorr ':='] <expression (PERS) of pos> ','
[\AccCorr ':='] <expression (PERS) of pos> ';'
```

Related information

For information about	See
Instruction `CapAPTrSetupAI`	*CapAPTrSetupAI - Setup an At-Point-Tracker controlled by analog input signals on page 46*
Instruction `CapAPTrSetupAO`	*CapAPTrSetupAO - Setup an At-Point-Tracker controlled by analog output signals on page 49*
Data type `capaptrreferencedata`	*capaptrreferencedata - Variable setup data for At-Point-Tracker on page 106*
Sensor Interface	*Application manual - Controller software IRC5*

4.1.5 CapC - Circular CAP motion instruction

Usage

`CapC` is used to move the tool center point (TCP) along a circular path to a given destination and at the same time control a continuous process. Furthermore it is possible to connect up to eight events to `CapC`. The events are defined using the instructions `TriggRampAO`, `TriggIO`, `TriggEquip`, `TriggInt`, `TriggCheckIO`, or `TriggSpeed`.

Basic examples

Example 1

Circular movements with `CapC`.

```
CapC cirp, p1, v100, cdata, weavestart, weave, fine, gun1;
```

The TCP of the tool, `gun1`, is moved circularly to the fine point `p1` with speed defined in `cdata`.

Example 2

Circular movement with user event and CAP event.

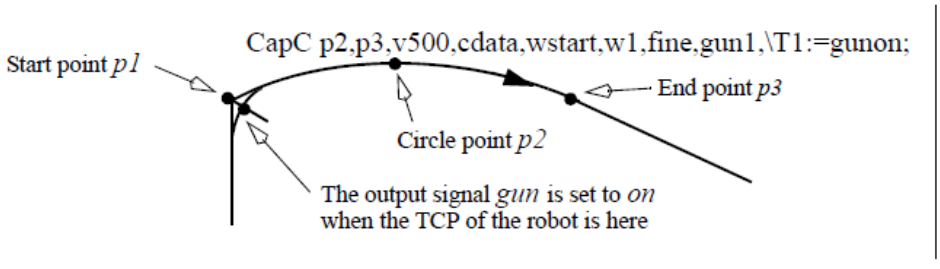
```
VAR intnum start_intno;
...
PROC main()
  VAR triggdata gunon;

  IDelete start_intno;
  CONNECT start_intno WITH start_trap;
  ICap start_intno, CAP_START;
  TriggIO gunon, 0 \Start \DOp:=gun, on;

  MoveJ p1, v500, z50, gun1;
  CapC p2,p3,v500,cdata,wstart,w1,fine,gun1,\T1:=gunon;
ENDPROC

TRAP start_trap
  ! This routine will be executed when the event CAP_START is
       reported
ENDTRAP
```

The digital output signal gun is set when the robot's TCP passes the midpoint of the corner path of the point `p1`. The trap routine start_trap is executed when the CAP process is starting.



xx1200000174

Continues on next page

4.1.5 CapC - Circular CAP motion instruction

Continuous Application Platform (CAP)
Continued

Arguments

```
CapC Cirpoint ToPoint [\Id] Speed Cdata [\MoveStartTimer] Weavestart
     Weave Zone [\Inpos] Tool [\WObj] [\Track] | [\Corr]
     [\PreProcessTracking] [\Time] [\T1] [\T2] [\T3] [\T4] [\T5]
     [\T6] [\T7] [\T8] [\TLoad]
```

`Cirpoint`

Data type: robtarget

The circle point of the robot. The circle point is a point on the circle between the start point and the destination point. To obtain the best accuracy it should be placed about halfway between the start and destination points. If it is placed to close to the start or end point, the robot may give a warning. The circle point is defined as a named position or stored directly in the instruction (if marked with an * in the instruction).

`ToPoint`

Data type: robtarget

The destination point of the robot and additional axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

`[\Id]`

(Sync identity)

Data type: identno

Synchronization identity for RAPID movement instructions in a MultiMove system in synchronized mode.

`Speed`

Data type: speeddata

The speed data that applies to movements without active CAP process. Speed data defines the velocity of the tool center point, the additional axes and of the tool reorientation. If a CAP process is active (not blocked), then the Cdata argument defines the TCP velocity.

`Cdata`

(CAP process Data)

Data type: capdata

CAP process data, see *capdata - CAP data on page 108* for a detailed description.

`[\Movestart_timer]`

(Time in s)

Data type: num

Upper limit for the time difference between the order of the process start and the actual start of the robots TCP movement in a MultiMove system in synchronized mode.

`Weavestart`

(Weavestart Data)

Data type: weavestartdata

Continues on next page

Weave start data for the CAP process, see *weavestartdata - weave start data on page 138* for a detailed description.

`Weave`

(Weave Data)

Data type: capweavedata

Weaving data for the CAP process, see *capweavedata - Weavedata for CAP on page 122* for a detailed description.

`Zone`

Data type: zonedata

Zone data for the movement. Zone data describes the size of the generated corner path.

`[\Inpos]`

(In position)

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 center point (TCP) is the point that is moved to the specified destination position.

`[\WObj]`

Data type: wobjdata

The work object (coordinate system) to which the robot position in the instruction is related.

This argument can be omitted, and if it is, the position is related to the world coordinate system. If, on the other hand, a stationary TCP or coordinated additional axes are used, this argument must be specified for a linear movement relative to the work object to be performed.

`[\Track]`

(Track Sensor Data)

Data type: captrackdata

This data structure contains data needed for use of path correction generating sensors together with `CapC`, see *captrackdata - CAP track data on page 119*. This argument is not allowed together with the argument `\Corr`.

`[\Corr]`

(Use Correction Generator)

Data type: switch

This argument tells `CapC` to read path corrections from a correction generator, `CorrCon`. This argument is not allowed together with the argument `\Track`.

Continues on next page

4.1.5 CapC - Circular CAP motion instruction
Continuous Application Platform (CAP)
Continued

The RobotWare option *Path Offset* is required when using this argument.

`[\PreProcessTracking]`

Data type: switch

This argument is effective only if `first_instruction` is set to `TRUE` and the `\Track` argument is present.

This argument activates *Pre Process Tracking*, which means that the robot will be tracking only, without process, during that CapX instruction. Thereby sensor data are available for successful tracking right off the start of the path with process, e.g. welding.

For more information see *Operating manual - Tracking and searching with optical sensors*.

`[\Time]`

Data type: num

This argument is used to specify the total time in seconds during which the robot and additional axes move. It is then substituted for the corresponding speed data.

`[\T1] [\T2] [\T3] [\T4] [\T5] [\T6] [\T7] [\T8]`
(Trigg x)

Data type: triggdata

Variables that refer to trigger conditions and trigger activity, defined earlier in the program using the instructions `TriggRampAO`, `TriggIO`, `TriggEquip`, or `TriggInt`.

`[\TLoad]`

Data type: loaddata

The argument `\TLoad` describes the total load used in the movement. The total load is the tool load together with the payload that the tool is carrying. If the argument `\TLoad` is used, then the `loaddata` in the current `tooldata` is not considered.

If the argument `\TLoad` is set to `load0`, then the argument is not considered and the `loaddata` in the current `tooldata` is used instead. For a complete description of the argument `TLoad`, see `MoveL`.

Error handling

There are several different types of errors that can be handled in the error handler for the `CapC`/`CapL` instructions:

- supervision errors
- sensor specific errors
- errors specific to a MultiMove system
- errors inherited from `TriggX` functionality
- other CAP errors

If one of the signals that is supposed to be supervised does not have the correct value, or if it changes value during supervision, the system variable `ERRNO` is set.

If no values can be read from the track sensor, the system variable `ERRNO` is set.

Continues on next page

For a MultiMove system running in synchronized mode the error handler must take care of two other errors. One is used to report that some other application has detected an recoverable error. This enables recoverable error handling in synchronized RAPID tasks. The other error, CAP_MOV_WATCHDOG, is reported if the time between the order of the process start and the actual start of the robots TCP movement in a MultiMove system in synchronized mode expires. The time used is specified in the optional parameter Movestart_timer in the CapC instruction.

If anything abnormal is detected, program execution will stop. If, however, an error handler is programmed, the errors defined below can be remedied without stopping production. However, a recommendation is that some of the errors (the errors with CAP_XX) these errors should not be presented for the end user. Map those errors to a application specific error. For the supervision errors the instruction CapGetFailSigs can be used to get which specific signal that failed.

Supervision errors

The following recoverable errors are generated and can be handled in an error handler. The system variable ERRNO will be set to:

CAP_PRE_ERR	This error occurs when there is an error in the PRE supervision list, that is, when the conditions in the list are not met within the specified time frame (specified in pre_cond time-out).
CAP_PRESTART_ERR	This error occurs when there is an error during the supervision of the PRE phase.
CAP_END_PRE_ERR	This event occurs when there is an error in the END_PRE supervision list, that is, when the conditions in the list are not met within the specified time frame (specified in start_cond time-out).
CAP_START_ERR	This event occurs when there is an error in the START supervision list, that is, when the conditions in the list are not met within the specified time frame (specified in start_cond time-out).
CAP_MAIN_ERR	This error occurs when there is an error during the supervision of the main phase.
CAP_ENDMAIN_ERR	This error occurs when there is an error in the END_MAIN supervision list, that is, when the conditions in the list are not met within the specified time frame (specified in end_main_cond time-out).
CAP_START_POST1_ERR	This event occurs when there is an error in the START_POST1 supervision list, that is, when the conditions in the list are not met within the specified time frame (specified in end_main_cond time-out).
CAP_POST1_ERR	This error occurs when there is an error during the supervision of the POST1 phase.
CAP_POST1END_ERR	This error occurs when there is an error in the END_POST1 supervision list, that is, when the conditions in the list are not met within the specified time frame (specified in end_main_cond time-out).

Continues on next page

4.1.5 CapC - Circular CAP motion instruction

Continuous Application Platform (CAP)

Continued

CAP_START_POST2_ERR	This event occurs when there is an error in the START_POST1 supervision list, that is, when the conditions in the list are not met within the specified time frame (specified in end_main_cond time-out).
CAP_POST2_ERR	This error occurs when there is an error during the supervision of the POST2 phase.
CAP_POST2END_ERR	This error occurs when there is an error in the END_POST2 supervision list, that is, when the conditions in the list are not met within the specified time frame (specified in end_main_cond time-out). If supervision is done on two different signals in the same phase, and both of them fails, the first one that is setup with SetupSuperv is the one that generates the error.

Sensor related errors

The following recoverable errors are generated and can be handled in an error handler. The system variable ERRNO will be set to:

CAP_TRACK_ERR	Track error occurs when reading data from sensor and after a time no valid data are received. One reason for this could be that the sensor cannot indicate the seam.
CAP_TRACKSTA_ERR	Track start error occurs when no valid data has been read from the laser track sensor.
CAP_TRACKCOR_ERR	Track correction error occurs when something goes wrong in the calculation of the offset.
CAP_TRACKCOM_ERR	The communication between the robot controller and the sensor equipment is broken.
CAP_TRACKPFR_ERR	It is not possible to continue tracking, if a power failure occurred during tracking.
CAP_SEN_NO_MEAS	The controller did not get a valid measurement from sensor.
CAP_SEN_NOREADY	The sensor is not ready yet.
CAP_SEN_GENERRO	A general sensor error occurred.
CAP_SEN_BUSY	The sensor is busy and cannot answer the request.
CAP_SEN_UNKNOWN	The command sent to the sensor is unknown to sensor.
CAP_SEN_ILLEGAL	The variable or block number sent to the sensor is illegal.
CAP_SEN_EXALARM	An external alarm occurred in the sensor.
CAP_SEN_CAALARM	A camera alarm occurred in the sensor.
CAP_SEN_TEMP	The sensor temperature is out of range.
CAP_SEN_VALUE	The value sent to the sensor is out of range.
CAP_SEN_CAMCHECK	The camera check failed.
CAP_SEN_TIMEOUT	The sensor did not respond within the time out time.

Continues on next page

Errors possible in MultiMove systems

The following recoverable errors are generated and can be handled in an error handler. The system variable ERRNO will be set to:

ERR_PATH_STOP	When using synchronized motion this error is reported when an application controlling one mechanical unit detects a recoverable error and notifies other applications that something went wrong. If this error code is received from a CapC instruction, the error is a reaction on another error. All tasks using motion instructions in synchronized mode in a MultiMove system should have this ERRNO value defined in the error handler.

Errors inherited from TriggX

The instruction CapC is based on the instruction TriggC. As a consequence you can get and handle the errors ERR_AO_LIM and ERR_DIPLAG_LIM, as in TriggC.

The system variable ERRNO will be set to:

ERR_AO_LIM	If the programmed ScaleValue/SetValue argument for the specified analog output signal AOp/AOutput in some of the connected TriggSpeed/TriggRampAO instructions, results are out of limit for the analog signal together with the programmed Speed in this instruction. The system variable ERRNO is set to ERR_AO_LIM.
ERR_DIPLAG_LIM	If the programmed DipLag argument in some of the connected TriggSpeed instructions, is too big in relation to the used system parameter *Event Preset Time*, the system variable ERRNO is set to ERR_DIPLAG_LIM.

Other CAP errors

The following recoverable errors are generated and can be handled in an error handler. The system variable ERRNO will be set to:

CAP_ATPROC_START	This recoverable error is generated at the end of the first CapC/L instruction of a sequence if the optional argument \PreProcessTracking is used. It can be handled in the error handler to start the process. For more information see *Operating manual - Tracking and searching with optical sensors*.
CAP_NOPROC_END	This error occurs when the instruction CapNoProcess is used to run a certain distance without application process and the end of this distance is reached. This is not really an error, but it uses the mechanisms of error recovery.
CAP_MOV_WATCHDOG	This error occurs when the switch \Movestart_timer is specified and the time between the process start (MAIN_STARTED) and the start of the robot movement exceeds the time specified with the switch.

Program execution

See *Technical reference manual - RAPID Instructions, Functions and Data types* for information about the MoveL and TriggL.

Continues on next page

4.1.5 CapC - Circular CAP motion instruction
Continuous Application Platform (CAP)
Continued

CAP process

During continuous execution in both Auto mode and Manual mode, the CAP process is running, unless it is blocked. That means, that all data controlling the CAP process (that is, `Cdata`, `Weavestart`, `Weave` and `Movestart_timer`), are used. In these modes all CAP trigger activities are carried out, see *ICap - connect CAP events to trap routines on page 91*.

In all other execution modes the CAP process is not running, and the `CapC` instruction behaves like a `MoveC` instruction.

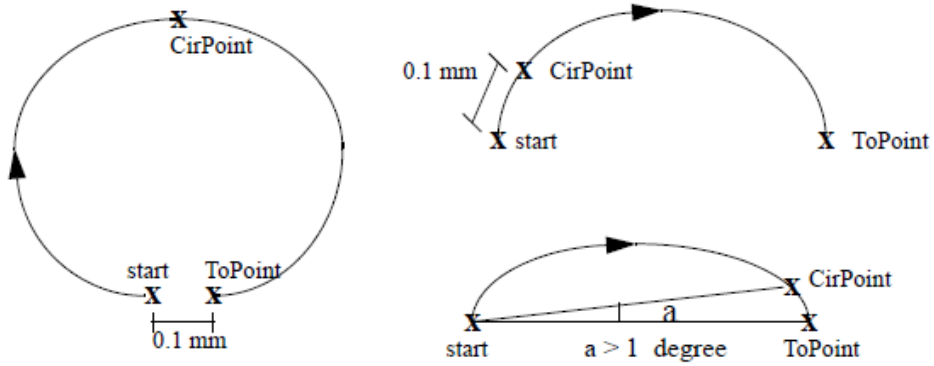
Trigger conditions [\T1] to [\T8]

As the trigger conditions are fulfilled when the robot is positioned closer and closer to the end point, the defined trigger activities are carried out. The trigger conditions are fulfilled either at a certain distance before the end point of the instruction, or at a certain distance after the start point of the instruction, or at a certain point in time (limited to a short time) before the end point of the instruction.

During stepping execution forwards, the I/O activities are carried out but the interrupt routines are not run. During stepping execution backwards, no trigger activities at all are carried out.

Limitations

There are some limitations in how the *CirPoint* and the *ToPoint* can be placed, as shown in the figure below.



xx1200000175

- Minimum distance between start and ToPoint is 0.1 mm.
- Minimum distance between start and CirPoint is 0.1 mm.
- Minimum angle between CirPoint and ToPoint from the start point is 1 degree.

The accuracy can be poor near the limits, for example, if the start point and the ToPoint on the circle are close to each other, the fault caused by the leaning of the circle can be much greater than the accuracy with which the points have been programmed.

A change of execution mode from forward to backward or vice versa, while the robot is stopped on a circular path, is not permitted and will result in an error message.

Continues on next page

The instruction CapC (or any other instruction including circular movement) should never be started from the beginning, with TCP between the circle point and the end point. Otherwise the robot will not take the programmed path (positioning around the circular path in another direction compared with that programmed).

Make sure that the robot can reach the circle point during program execution and divide the circle segment if necessary.

If the current start point deviates from the usual, so that the total positioning length of the instruction CapC is shorter than usual, it may happen that several or all of the trigger conditions are fulfilled immediately and at the same position. In such cases, the sequence in which the trigger activities are carried out will be undefined. The program logic in the user program may not be based on a normal sequence of trigger activities for an "incomplete movement".

CapC cannot be executed in a RAPID routine connected to any of the following special system events: PowerOn, Stop, QStop, Restart, Reset or Step.

Syntax

```
CapC
  [CirPoint ':='] < expression (IN) of robtarget >
  [ToPoint ':='] < expression (IN) of robtarget >
  ['\' Id ':=' < expression (IN) of identno > ] ','
  [Speed ':='] < expression (IN) of speeddata >
  [Cdata ':='] < persistent (PERS) of capdata >
  ['\' Movestart_timer ':=' < expression (IN) of num > ] ','
  [Weavestart ':='] <persistent (PERS) of weavestartdata >
  [Weave ':='] < persistent (PERS) of capweavedata >
  [Zone ':='] < expression (IN) of zonedata >
  ['\' Inpos ':=' < expression (IN) of stoppointdata >] ','
  [Tool ':='] < persistent (PERS) of tooldata >
  ['\' WObj ':=' < persistent (PERS) of wobjdata > ]
  ['\' Track ':=' < persistent (PERS) of captrackdata > ]
  |[ '\' Corr]
  |[ '\' PreProcessTracking]
  ['\' Time ':=' < expression (IN) of num > ]
  ['\' T1 ':=' < variable (VAR) of triggdata > ]
  ['\' T2 ':=' < variable (VAR) of triggdata > ]
  ['\' T3 ':=' < variable (VAR) of triggdata > ]
  ['\' T4 ':=' < variable (VAR) of triggdata > ]
  ['\' T5 ':=' < variable (VAR) of triggdata > ]
  ['\' T6 ':=' < variable (VAR) of triggdata > ]
  ['\' T7 ':=' < variable (VAR) of triggdata > ]
  ['\' T8 ':=' < variable (VAR) of triggdata > ]
  ['\' TLoad ':=' < persistent (PERS) of loaddata > ] ';'
```

Related information

For information about	See
Definition of CAP data	*capdata - CAP data on page 108*
Definition of weave start data	*weavestartdata - weave start data on page 138*

Continues on next page

4.1.5 CapC - Circular CAP motion instruction

Continuous Application Platform (CAP)

Continued

For information about	See
Definition of weave data	*capweavedata - Weavedata for CAP on page 122*
Definition of track data	*captrackdata - CAP track data on page 119*
Path Offset	*Application manual - Controller software IRC5*
Using optical sensors for tracking or searching.	*Operating manual - Tracking and searching with optical sensors*
`MoveL` `CorrCon`	*Technical reference manual - RAPID Instructions, Functions and Data types*

4.1.6 CapCondSetDO - Set a digital output signal at TCP stop

Usage

`CapCondSetDO` is used to define a digital output signal and its value, which will be set when the TCP of the robot that runs CAP, stops moving during a CAP instruction (`CapL` or `CapC`) before the CAP sequence is finished.

An existing definition of such signals, is cleared with the CAP instruction `InitSuperv`.

Basic example

```
CapCondSetDO do15, 1;
```

The signal do15 is set to 1 when the TCP stops.

```
CapCondSetDO weld, off;
```

The signal weld is set to `off` when the TCP stops.

Arguments

```
CapCondSetDO Signal Value
```

Signal

Data type: `signaldo`

The name of the signal to be changed.

Value

Data type: `dionum`

The desired value of the signal 0 or 1.

Specified Value	Set digital output to
0	0
Any value except 0	1

Limitations

The final value of the signal depends on the configuration of the signal. If the signal is inverted in the system parameters, the value of the physical channel is the opposite.

A maximum of 10 signals per RAPID task may be set up.

Syntax

```
CapCondSetDO
  [Signal ':='] < variable (VAR) of signaldo > ','
  [Value ':='] < expression (IN) of dionum > ';'
```

Related information

For information about	See
`InitSuperv` instruction	*InitSuperv - Reset all supervision for CAP on page 96*
`SetupSuperv` instruction	*SetupSuperv - Setup conditions for signal supervision in CAP on page 101*

Continues on next page

4.1.6 CapCondSetDO - Set a digital output signal at TCP stop

Continuous Application Platform (CAP)

Continued

For information about	See
`RemoveSuperv` instruction	*RemoveSuperv - Remove condition for one signal on page 99*

4.1.7 CapEquiDist - Generate equidistant event

Usage

`CapEquiDist` is used to tell CAP to generate an equidistant RAPID event (EQUIDIST) on the CAP path. The first event is generated at the startpoint of the first CAP instruction in a sequence of CAP instructions. From RAPID it is possible to subscribe this event using `ICap`.

Basic example

```
VAR intnum intno_equi;

PROC main()
  ......
  IDelete intno_equi;
  Connect intno_equi equi_trp;
  ICap intno_equi, EQUIDIST
  ......
  CapEquiDist\Distance:=5.0;
  MoveL p60, v1000, fine, tWeldGun;
  CapL p_fig3_l_1, v500, cd, wsd, cwd, z10, tWeldGun;
  CapL p_fig3_l_2, v500, cd, wsd, cwd, fine, tWeldGun;
  ......
  CapEquiDist\Reset;
  MoveL p70, v1000, fine, tWeldGun;
  CapL p_fig3_l_3, v500, cd, wsd, cwd, fine, tWeldGun;
  ......

  ERROR
    Retry;
ENDPROC

TRAP equi_trp
  ! do whatever you want, but it must not take too long time
ENDTRAP
```

In this example, the event `EQUIDIST` will be generated on the first CAP path. It will be sent every 5 mm on the path over several CAP instructions with zones.

Arguments

```
CapEquiDist [\Distance] [\Reset]
```

[\Distance]

Distance in mm

Data type: `num`

The data provided with this optional argument defines the distance in mm between two consecutive equidistant events.

[\Reset]

Reset event generation

Continues on next page

4.1.7 CapEquiDist - Generate equidistant event

Continuous Application Platform (CAP)
Continued

Data type: `switch`

If this switch is present, the event generation is reset, that is, the equidistant event will not be generated any longer on a `CapL`/`CapC` path. This switch has precedence before the `\Distance` switch.

Limitations

If the CAP path is long compared to the event distance, the system can run out of event resources, and the error message **50368 Too Short distance between equidistant events**.

Syntax

```
CapEquiDist
  ['\' Distance ':=' < expression (IN) of num >]
  ['\' Reset] ';'
```

4.1.8 CapL - Linear CAP motion instruction

Usage

`CapL` is used to move the tool center point (TCP) linearly to a given destination and at the same time control a continuous process. Furthermore it is possible to connect up to eight events to `CapL`. The events are defined using the instructions `TriggRampAO`, `TriggIO`, `TriggEquip`, `TriggInt`, `TriggCheckIO`, or `TriggSpeed`.

Basic examples

Example1

Linear movements with `CapL`.

```
CapL p1, v100, cdata, weavestart, weave, z50, gun1;
```

The TCP of the tool, gun1, is moved linearly to the position p1, with speed defined in `cdata`, and zone data z50.

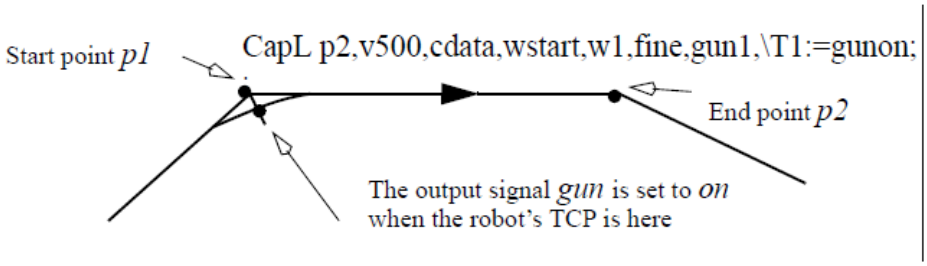
Example 2

Circular movement with user event and CAP event.

```
VAR intnum start_intno;
...
PROC main()
  VAR triggdata gunon;
  IDelete start_intno;
  CONNECT start_intno WITH start_trap;
  ICap start_intno, CAP_START;
  TriggIO gunon, 0 \Start \DOp:=gun, on;
  MoveJ p1, v500, z50, gun1;
  CapL p2, v500, cdata, wstart, w1, fine, gun1 \T1:=gunon;
ENDPROC

TRAP start_trap
  !This routine is executed when event CAP_START arrives
ENDTRAP
```

The digital output signal gun is set when the robot TCP passes the midpoint of the corner path of the point p1. The trap routine `start_trap` is executed when the CAP process is starting.



xx1200000173

Continues on next page

4.1.8 CapL - Linear CAP motion instruction

Continuous Application Platform (CAP)
Continued

Arguments

```
CapLToPoint [\Id] Speed Cdata [\MoveStartTimer] Weavestart Weave
     Zone [\Inpos] Tool [\WObj] [\Track] | [\Corr]
     [\PreProcessTracking] [\Time] [\T1] [\T2] [\T3] [\T4] [\T5]
     [\T6] [\T7] [\T8] [\TLoad]
```

`ToPoint`

Data type: robtarget

The destination point of the robot and additional axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

`[ \Id ]`

(Sync identity)

Data type: identno

Synchronization identity for RAPID movement instructions in a MultiMove system in synchronized mode.

`Speed`

Data type: speeddata

The speed data that applies to movements without active CAP process. Speed data defines the velocity of the tool center point, the additional axes and of the tool reorientation. If a CAP process is active (not blocked), then the Cdata argument defines the TCP velocity.

`Cdata`

(CAP process Data)

Data type: capdata

CAP process data, see *capdata - CAP data on page 108* for a detailed description.

`[ \Movestart_timer ]`

(Time in s)

Data type: num

Upper limit for the time difference between the order of the process start and the actual start of the robots TCP movement in a MultiMove system in synchronized mode.

`Weavestart`

(Weavestart Data)

Data type: weavestartdata

Weave start data for the CAP process, see *weavestartdata - weave start data on page 138* for a detailed description.

`Weave`

(Weave Data)

Data type: capweavedata

Weaving data for the CAP process, see *capweavedata - Weavedata for CAP on page 122* for a detailed description.

Continues on next page

`Zone`

Data type: zonedata

Zone data for the movement. Zone data describes the size of the generated corner path.

`[\Inpos ]`

(In position)

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 center point (TCP) is the point that is moved to the specified destination position.

`[ \WObj ]`

Data type: wobjdata

The work object (coordinate system) to which the robot position in the instruction is related.

This argument can be omitted, and if it is, the position is related to the world coordinate system. If, on the other hand, a stationary TCP or coordinated additional axes are used, this argument must be specified for a linear movement relative to the work object to be performed.

`[ \Track ]`

(Track Sensor Data)

Data type: captrackdata

This data structure contains data needed for use of path correction generating sensors together with `CapL`, see *captrackdata - CAP track data on page 119*. This argument is not allowed together with the argument `\Corr`.

`[ \Corr ]`

(Use Correction Generator)

Data type: switch

This argument tells `CapL` to read path corrections from a correction generator, `CorrCon`. This argument is not allowed together with the argument `\Track`.

The RobotWare option *Path Offset* is required when using this argument.

`[\PreProcessTracking]`

Data type: switch

This argument is effective only if `first_instruction` is set to `TRUE` and the `\Track` argument is present.

This argument activates *Pre Process Tracking*, which means that the robot will be tracking only, without process, during that CapX instruction. Thereby sensor data

Continues on next page

4.1.8 CapL - Linear CAP motion instruction
Continuous Application Platform (CAP)
Continued

are available for successful tracking right off the start of the path with process, e.g. welding.

For more information see *Operating manual - Tracking and searching with optical sensors*.

`[ \Time ]`

Data type: num

This argument is used to specify the total time in seconds during which the robot and additional axes move. It is then substituted for the corresponding speed data.

`[ \T1 ]` to `[ \T8 ]`

(Trigg x)

Data type: triggdata

Variables that refer to trigger conditions and trigger activity, defined earlier in the program using the instructions `TriggRampAO`, `TriggIO`, `TriggEquip`, or `TriggInt`.

`[\TLoad]`

Data type: loaddata

The argument `\TLoad` describes the total load used in the movement. The total load is the tool load together with the payload that the tool is carrying. If the argument `\TLoad` is used, then the `loaddata` in the current `tooldata` is not considered.

If the argument `\TLoad` is set to `load0`, then the argument is not considered and the `loaddata` in the current `tooldata` is used instead. For a complete description of the argument `TLoad`, see `MoveL`.

Error handling

There are several different types of errors that can be handled in the error handler for the `CapC`/`CapL` instructions:

- supervision errors
- sensor specific errors
- errors specific to a MultiMove system
- errors inherited from `TriggX` functionality
- other CAP errors

If one of the signals that is supposed to be supervised does not have the correct value, or if it changes value during supervision, the system variable `ERRNO` is set.

If no values can be read from the track sensor, the system variable `ERRNO` is set.

For a MultiMove system running in synchronized mode the error handler must take care of two other errors. One is used to report that some other application has detected an recoverable error. This enables recoverable error handling in synchronized RAPID tasks. The other error, `CAP_MOV_WATCHDOG`, is reported if the time between the order of the process start and the actual start of the robots TCP movement in a MultiMove system in synchronized mode expires. The time used is specified in the optional parameter `Movestart_timer` in the `CapL` instruction.

Continues on next page

If anything abnormal is detected, program execution will stop. If, however, an error handler is programmed, the errors defined below can be remedied without stopping production. However, a recommendation is that some of the errors (the errors with CAP_XX) these errors should not be presented for the end user. Map those errors to a application specific error. For the supervision errors the instruction CapGetFailSigs can be used to get which specific signal that failed.

Supervision errors

The following recoverable errors are generated and can be handled in an error handler. The system variable ERRNO will be set to:

CAP_PRE_ERR	This error occurs when there is an error in the PRE supervision list, that is, when the conditions in the list are not met within the specified time frame (specified in pre_cond time-out).
CAP_PRESTART_ERR	This error occurs when there is an error during the supervision of the PRE phase.
CAP_END_PRE_ERR	This event occurs when there is an error in the END_PRE supervision list, that is, when the conditions in the list are not met within the specified time frame (specified in start_cond time-out).
CAP_START_ERR	This event occurs when there is an error in the START supervision list, that is, when the conditions in the list are not met within the specified time frame (specified in start_cond time-out).
CAP_MAIN_ERR	This error occurs when there is an error during the supervision of the main phase.
CAP_ENDMAIN_ERR	This error occurs when there is an error in the END_MAIN supervision list, that is, when the conditions in the list are not met within the specified time frame (specified in end_main_cond time-out).
CAP_START_POST1_ERR	This event occurs when there is an error in the START_POST1 supervision list, that is, when the conditions in the list are not met within the specified time frame (specified in end_main_cond time-out).
CAP_POST1_ERR	This error occurs when there is an error during the supervision of the POST1 phase.
CAP_POST1END_ERR	This error occurs when there is an error in the END_POST1 supervision list, that is, when the conditions in the list are not met within the specified time frame (specified in end_main_cond time-out).
CAP_START_POST2_ERR	This event occurs when there is an error in the START_POST1 supervision list, that is, when the conditions in the list are not met within the specified time frame (specified in end_main_cond time-out).
CAP_POST2_ERR	This error occurs when there is an error during the supervision of the POST2 phase.
CAP_POST2END_ERR	This error occurs when there is an error in the END_POST2 supervision list, that is, when the conditions in the list are not met within the specified time frame (specified in end_main_cond time-out). If supervision is done on two different signals in the same phase, and both of them fails, the first one that is setup with SetupSuperv is the one that generates the error.

Continues on next page

4.1.8 CapL - Linear CAP motion instruction

Continuous Application Platform (CAP)
Continued

Sensor related errors

The following recoverable errors are generated and can be handled in an error handler. The system variable ERRNO will be set to:

CAP_TRACK_ERR	Track error occurs when reading data from sensor and after a time no valid data are received. One reason for this could be that the sensor cannot indicate the seam.
CAP_TRACKSTA_ERR	Track start error occurs when no valid data has been read from the laser track sensor.
CAP_TRACKCOR_ERR	Track correction error occurs when something goes wrong in the calculation of the offset.
CAP_TRACKCOM_ERR	The communication between the robot controller and the sensor equipment is broken.
CAP_TRACKPFR_ERR	It is not possible to continue tracking, if a power failure occurred during tracking.
CAP_SEN_NO_MEAS	The controller did not get a valid measurement from sensor.
CAP_SEN_NOREADY	The sensor is not ready yet.
CAP_SEN_GENERRO	A general sensor error occurred.
CAP_SEN_BUSY	The sensor is busy and cannot answer the request.
CAP_SEN_UNKNOWN	The command sent to the sensor is unknown to sensor.
CAP_SEN_ILLEGAL	The variable or block number sent to the sensor is illegal.
CAP_SEN_EXALARM	An external alarm occurred in the sensor.
CAP_SEN_CAALARM	A camera alarm occurred in the sensor.
CAP_SEN_TEMP	The sensor temperature is out of range.
CAP_SEN_VALUE	The value sent to the sensor is out of range.
CAP_SEN_CAMCHECK	The camera check failed.
CAP_SEN_TIMEOUT	The sensor did not respond within the time out time.

Errors possible in MultiMove systems

The following recoverable errors are generated and can be handled in an error handler. The system variable ERRNO will be set to:

ERR_PATH_STOP	When using synchronized motion this error is reported when an application controlling one mechanical unit detects a recoverable error and notifies other applications that something went wrong. If this error code is received from a CapL instruction, the error is a reaction on another error. All tasks using motion instructions in synchronized mode in a MultiMove system should have this ERRNO value defined in the error handler.

Continues on next page

Errors inherited from TriggX

The instruction CapL is based on the instruction TriggL. As a consequence you can get and handle the errors ERR_AO_LIM and ERR_DIPLAG_LIM, as in TriggL.

The system variable ERRNO will be set to:

ERR_AO_LIM	If the programmed ScaleValue/SetValue argument for the specified analog output signal AOp/AOutput in some of the connected TriggSpeed/TriggRampAO instructions, results are out of limit for the analog signal together with the programmed Speed in this instruction. The system variable ERRNO is set to ERR_AO_LIM.
ERR_DIPLAG_LIM	If the programmed DipLag argument in some of the connected TriggSpeed instructions, is too big in relation to the used system parameter *Event Preset Time*, the system variable ERRNO is set to ERR_DIPLAG_LIM.

Other CAP errors

The following recoverable errors are generated and can be handled in an error handler. The system variable ERRNO will be set to:

CAP_ATPROC_START	This recoverable error is generated at the end of the first CapC/L instruction of a sequence if the optional argument \PreProcessTracking is used. It can be handled in the error handler to start the process. For more information see *Operating manual - Tracking and searching with optical sensors*.
CAP_NOPROC_END	This error occurs when the instruction CapNoProcess is used to run a certain distance without application process and the end of this distance is reached. This is not really an error, but it uses the mechanisms of error recovery.
CAP_MOV_WATCHDOG	This error occurs when the switch \Movestart_timer is specified and the time between the process start (MAIN_STARTED) and the start of the robot movement exceeds the time specified with the switch.

Program execution

See *Technical reference manual - RAPID Instructions, Functions and Data types* for information about the MoveL and TriggL.

CAP process

During continuous execution in both Auto mode and Manual mode, the CAP process is running, unless it is blocked. That means, that all data controlling the CAP process (that is, Cdata, Weavestart, Weave and Movestart_timer), are used. In these modes all CAP trigger activities are carried out, see *ICap - connect CAP events to trap routines on page 91*.

In all other execution modes the CAP process is not running, and the CapL instruction behaves like a MoveL instruction.

Continues on next page

4.1.8 CapL - Linear CAP motion instruction

Continuous Application Platform (CAP)
Continued

Trigger conditions [\T1] to [\T8]

As the trigger conditions are fulfilled when the robot is positioned closer and closer to the end point, the defined trigger activities are carried out. The trigger conditions are fulfilled either at a certain distance before the end point of the instruction, or at a certain distance after the start point of the instruction, or at a certain point in time (limited to a short time) before the end point of the instruction.

During stepping execution forwards, the I/O activities are carried out but the interrupt routines are not run. During stepping execution backwards, no trigger activities at all are carried out.

Limitations

If the current start point deviates from the usual, so that the total positioning length of the instruction `CapL` is shorter than usual (for example, at the start of `CapL` with the robot position at the end point), it may happen that several or all of the trigger conditions are fulfilled immediately and at the same position. In such cases, the sequence in which the trigger activities are carried out will be undefined. The program logic in the user program may not be based on a normal sequence of trigger activities for an "incomplete movement".

The behavior of the CAP process may be undefined if an error occurs during `CapL` or `CapC` instructions with extremely short TCP movements (< 1 mm).

`CapL` cannot be executed in a RAPID routine connected to any of the following special system events: PowerOn, Stop, QStop, Restart, Reset or Step.

Syntax

```
CapL
  [ToPoint ':='] < expression (IN) of robtarget >
  ['\' Id ':=' < expression (IN) of identno >] ','
  [Speed ':='] < expression (IN) of speeddata > ','
  [Cdata ':='] < persistent (PERS) of capdata >
  ['\' Movestart_timer ':=' < expression (IN) of num >] ','
  [Weavestart ':='] <persistent (PERS) of weavestartdata > ','
  [Weave ':='] < persistent (PERS) of capweavedata > ','
  [Zone ':='] < expression (IN) of zonedata >
  ['\' Inpos ':=' < expression (IN) of stoppointdata >] ','
  [Tool ':='] < persistent (PERS) of tooldata >
  ['\' WObj ':=' < persistent (PERS) of wobjdata >]
  ['\' Track ':=' < persistent (PERS) of captrackdata >]
  |[ '\' Corr]
  |[ '\' PreProcessTracking]
  ['\' Time ':=' < expression (IN) of num >]
  ['\' T1 ':=' < variable (VAR) of triggdata >]
  ['\' T2 ':=' < variable (VAR) of triggdata >]
  ['\' T3 ':=' < variable (VAR) of triggdata >]
  ['\' T4 ':=' < variable (VAR) of triggdata >]
  ['\' T5 ':=' < variable (VAR) of triggdata >]
  ['\' T6 ':=' < variable (VAR) of triggdata >]
  ['\' T7 ':=' < variable (VAR) of triggdata >]
  ['\' T8 ':=' < variable (VAR) of triggdata >]
```

Continues on next page

```
['\' TLoad':=' < persistent (PERS) of loaddata >] ';'
```

Related information

For information about	See
Definition of CAP data	*capdata - CAP data on page 108*
Definition of weave start data	*weavestartdata - weave start data on page 138*
Definition of weave data	*capweavedata - Weavedata for CAP on page 122*
Definition of track data	*captrackdata - CAP track data on page 119*
Path Offset	*Application manual - Controller software IRC5*
Using optical sensors for tracking or searching.	*Operating manual - Tracking and searching with optical sensors*
`MoveL` `TriggL`	*Technical reference manual - RAPID Instructions, Functions and Data types*

4.1.9 CapLATrSetup - Set up a Look-Ahead-Tracker

Usage

`CapLATrSetup` (*Set up a Look-Ahead-Tracker*) is used to set up a Look-Ahead-Tracker type of sensor, for example, Laser Tracker.

The sensor interface communicates with a maximum of two sensors over serial channels using the RTP1 transport protocol. The two channels must be named *laser1:* and *swg:*.

Basic example

SIO.cfg:

```
COM_TRP:
-Name "SCOUT:" -Type "RTP1"
-Name "digi-ip:" -Type "SOCKDEV" -PhyChannel "LAN1" -RemoteAdress
     "192.168.125.5"
```

RAPID code:

```
! Define variable numbers
CONST num SensorOn := 6;
CONST num XCoord := 8;
CONST num YCoord := 9;
CONST num ZCoord := 10;
! Sensor calibration frame
PERS pose calibFrame := [[236.4,0.3,96.3],[1,0,0,0]];
! Trackdata
PERS captrackdata captrack1 := ["digi-ip:", [1,10,1,0,0,0,0,0]];

! Set up a Laser Tracker
CapLATrSetup “digi-ip:”,
     calibFrame\SensorFreq:=20\CorrFilter:=5\MaxBlind:=100\MaxIncCorr:=2;

! Request start of sensor measurements
WriteVar "digi-ip:", SensorOn, 1;

! Track using Cap
CapL p_fig1_l_1, v200, cd_event1, wsd_event, cwd_event, z20,
     tWeldGun\Track:=captrack1;

! Stop sensor
WriteVar "digi-ip:", SensorOn, 0;
```

Arguments

```
CapLATrSetup device CalibFrame CalibPos [\WarnMaxCorr] [\LogFile]
     [\LogSize] [\SensorFreq] [\IpolServoDelay] [\IpolCorrGain]
     [\ServoSensFactor] [\CorrFilter] [\IpolCorrFilter]
     [\ServoCorrFilter] [\ErrRampIn] [\ErrRampOut] [\CBAngle]
     [\MaxBlind] [\MaxIncCorr] [\CalibFrame2] [\CalibFrame3]
```

device

Data type: `string`

Continues on next page

Device name as defined in sio.cfg.

calibframe

Data type: `pose`

LATR calibration frame (position and orientation relative the predefined tool tool0.

CalibPos

Data type: `pose`

LATR calibration offset. Adjustment of the sensor frame which places the origo of the path correction frame near the level of the tool frame used during calibration.

[\WarnMaxCorr]

Data type: `switch`

If this switch is present, program execution is not interrupted, when the limit for maximum correction, specified in the trackdata, is exceeded. Only a warning will be sent.

[\Logfile]

Data type: `string`

Name of tracklog log file.

[\LogSize]

Data type: `num`

Size of the tracklog ring buffer, that is the number of sensor measurements that can be buffered during tracking.

Default: 1000.

[\SensorFreq]

Data type: `num`

Defines the sample frequency of the sensor used (for example, M-Spot-90 has 5Hz sampling frequency).

The highest available value is dependent on the communication link and its speed. We recommend not to use values higher than 20Hz.

Default: 5 Hz.

[\IpolServoDelay]

Data type: `num`

Defines an robot controller internal time delay between ipol task and servo task.

Default: 74 ms.

Note

Do not change the default value!

[\IpolCorrGain]

Data type: `num`

Defines, the gain factor for the correction imposed on ipol.

Continues on next page

4.1.9 CapLATrSetup - Set up a Look-Ahead-Tracker
Continuous Application Platform (CAP)
Continued

Default: 0.0.

> **Note**
>
> Do not change the default value!

[\ServoSensFactor]

Data type: num

Defines the number of servo corrections per sensor reading.

Default: 0.

> **Note**
>
> Do not change the default value!

[\CorrFilter]

Data type: num

Defines filtering of the correction calculated, using mean value over corr filter values.

Default: 1.

> **Note**
>
> Do not change the default value!

[\IpolCorrFilter]

Data type: num

Defines filtering of the ipol correction, using mean value over path filter values.

Default: 1.

> **Note**
>
> Do not change the default value!

[\ServoCorrFilter]

Data type: num

Defines filtering of the servo correction, using mean value over path servo filter values.

Default: 1.

> **Note**
>
> Do not change the default value!

[\ErrRampIn]

Data type: num

Defines during how many sensor readings ramp in is done after error caused by sensor reading.

Continues on next page

Default: 1.

[\ErrorRampOut]

Data type: `num`

Defines during how many sensor readings ramp out is done when an error caused by sensor reading occurred.

Default: 1.

[\CBAngle]

Data type: `num`

Defines the angle between a 3D sensor beam and the sensor z-axis

Default: 0.0.

[\MaxBlind]

Data type: `num`

Maximum distance the TCP may move assuming, that the latest correction is still valid.

At the start of the tracking, the `MaxBlind` distance is automatically increased by the look ahead of the sensor.

Default: no limit.

[\MaxIncCorr]

Data type: `num`

Maximum incremental correction allowed.

If the incremental TCP correction is bigger than `\MaxIncCorr` and `\WarnMaxCorr` was specified, the robot will continue its path but the applied incremental correction will not exceed `\MaxIncCorr`. If `\WarnMaxCorr` was not specified, a track error is reported and program execution is stopped.

Default: 5 mm.

[\CalibFrame2]

Data type: `pose`

Alternative LATR calibration frame number 2 (position and orientation relative the predefined tool `tool0`).

[\CalibFrame3]

Data type: `pose`

Alternative LATR calibration frame number 3 (position and orientation relative the predefined tool `tool0`).

Syntax

```
CapLATrSetup
  [device ':='] < expression (IN) of string> ','
  [CalibFrame ':='] < persistent (PERS) of pose > ','
  [CalibPos ':='] < persistent (PERS) of pos >
  [\WarnMaxCorr]
  [\LogFile ':=' < expression (IN) of string >]
  [\LogSize ':=' < expression (IN) of num >]
```

Continues on next page

4.1.9 CapLATrSetup - Set up a Look-Ahead-Tracker

Continuous Application Platform (CAP)
Continued

```
[\SensorFreq ':=' < expression (IN) of num >]
[\IpolServoDelay ':=' < expression (IN) of num >]
[\IpolCorrGain ':=' < expression (IN) of num >]
[\ServoSensFactor ':=' < expression (IN) of num >]
[\CorrFilter ':=' < expression (IN) of num >]
[\IpolCorrFilter ':=' < expression (IN) of num >]
[\ServoCorrFilter ':=' < expression (IN) of num >]
[\ErrRampIn ':=' < expression (IN) of num >]
[\ErrRampOut ':=' < expression (IN) of num >]
[\CBAngle ':=' < expression (IN) of num >]
[\MaxBlind ':=' < expression (IN) of num >]
[\MaxIncCorr ':=' < expression (IN) of num >]
[\CalibFrame2 ':=' < persistent (PERS) of pose >]
[\CalibFrame3 ':=' < persistent (PERS) of pose >] ';'
```

Related information

For information about	See
Sensor Interface	*Application manual - Controller software IRC5*

4.1.10 CapNoProcess - Run CAP without process

Usage

`CapNoProcess` is used to run CAP a certain distance without process.

With `CapNoProcess`, it is possible to tell CAP to execute a certain distance (in mm) without process. This is useful, if there was a recoverable process error, which in some way makes it impossible to restart the process at the error location.

In the beginning and at the end of the skip distance, backing on the path (`restart_dist` component in `capdata`) is suppressed.

At the end of the skip distance a error with errno CAP_NOPROC_END is generated.

Basic example

```
VAR num skip_dist := 0.0;
VAR bool cap_skip := FALSE;

PROC main()
  ......
  skip_dist := 25.0;
  CapL p_fig3_l_1, v500, cd, wsd, cwd, fine, tWeldGun;
  ......
  skip_dist := 15.0;
  CapL p_fig3_l_3, v500, cd, wsd, cwd, fine, tWeldGun;
  ......

  ERROR
    StorePath;
    TEST ERRNO
    CASE CAP_NOPROC_END:
      IF cap_skip THEN
        ! This is the end of the skip distance
        cap_skip := FALSE;
      ENDIF
    CASE CAP_MAIN_ERR:
      IF skip_dist > 0.0 THEN
        ! This is the start of the skip distance
        CapNoProcess skip_dist;
        cap_skip := TRUE;
      ENDIF
    DEFAULT:
    ENDTEST
    RestoPath;
    StartMoveRetry;
ENDPROC
ENDMODULE
```

In this example, the recoverable error CAP_MAIN_ERR is followed by 25 mm movement (at 10 mm/s) without process for the first `CapL` instruction and by 15

Continues on next page

4.1.10 CapNoProcess - Run CAP without process

Continuous Application Platform (CAP)
Continued

mm for the second. At the end of that distance, CAP_NOPROC_END is generated and the process is restarted.

Arguments

```
CapNoProcess skip_distance
```

skip_distance

Distance in mm

Data type: `num`

`CapNoProcess` has a `num` variable as input parameter, that defines the skip distance in mm.

Limitations

The speed of the TCP during skip is predefined with 10 mm/s. The shortest skip distance is predefined with 10 mm.

In synchronized MultiMove systems, the shortest distance of all skip distances defined for the different synchronized process robots will be the actual one.

If the skip distance is longer than the distance from the current TCP position to the end of the current sequence of CAP instructions, nothing special will happen: RAPID execution continues as usual, without stopping the robot.

Syntax

```
CapNoProcess
  [skip_dist ':='] < variable (IN) of num >';'
```

Related information

For information about	See
`InitSuperv` instruction	*InitSuperv - Reset all supervision for CAP on page 96*
`SetupSuperv` instruction	*SetupSuperv - Setup conditions for signal supervision in CAP on page 101*
`RemoveSuperv` instruction	*RemoveSuperv - Remove condition for one signal on page 99*

4.1.11 CapRefresh - Refresh CAP data

Usage

`CapRefresh` is used to tell the CAP process to refresh its process data. It can for example, be used to tune CAP process parameters during program execution.

Basic example

```
PROC PulseSpeed()
  ! Setup a 1 Hz timer interrupt
  CONNECT intno1 WITH TuneTrp;
  ITimer 1, intno1;
  CapL p1, v100, cdata, wstartdata, wdata, fine, gun1;
  IDelete intno1;
ENDPROC

TRAP TuneTrp
  ! Modify the main speed component of active cdata
  IF HighValueFlag = TRUE THEN
    cdata.speed_data.start := 10;
    HighValueFlag := FALSE;
  ELSE
    cdata.speed_data.start := 15;
    HighValueFlag := TRUE;
  ENDIF
  ! Order the process control to refresh process parameters
  CapRefresh;
ENDTRAP
```

In this example the speed will be switched between 10 and 15 mm/s at a rate of 1 Hz.

Arguments

```
CapRefresh [\MainSpeed] [\MainWeave] [\StartWeave] [\RestartDist]
```

Without optional argument the CAP data `capdata`, `capweavedata`, `weavestartdata`, `captrackdata`, and `movestarttimer` are - if present - re-read from the PERSISTENT RAPID variable specified in the currently active CAP instruction.

[\MainSpeed]

Data type: `switch`

If this switch is present, CAP will reread the component `capdata.speed_data.main` of the currently active CAP instruction.

[\MainWeave]

Data type: `switch`

If this switch is present, CAP will reread the components `capweavedata.width`, `capweavedata. length`, `capweavedata.bias`, and `capweavedata.active` of the currently active CAP instruction.

Continues on next page

4.1.11 CapRefresh - Refresh CAP data

Continuous Application Platform (CAP)
Continued

[\StartWeave]

Data type: `bool`

If this switch is present, CAP will use its value instead of `weavestartdata.active` of the currently active CAP instruction. The data of the currently active CAP instruction remain untouched.

[\RestartDist]

Data type: `num`

If this switch is present, CAP will use its value instead of `capdata.restart_dist` of the currently active CAP instruction. The data of the currently active CAP instruction remain untouched.

Syntax

```
CapRefresh
  ['\' MainSpeed]
  ['\' MainWeave]
  ['\' Startweave ':=' < expression (IN) of bool >]
  ['\' RestartDist ':=' < expression (IN) of num >] ';'
```

4.1.12 CAPSetStopMode - Set the stop mode for execution errors

Usage

`CAPSetStopMode` sets the stop mode that should be used when the robot movement is stopped due to a process error. The default stop mode is SMOOTH_STOP_ON_PATH.

Basic example

```
CAPSetStopMode QUICK_STOP_ON_PATH;
```

Arguments

```
CAPSetStopMode StopMode;
```

StopMode

Data type: `capstopmode`

Stop mode, see *capstopmode - Defines stop modes for CAP on page 118*.

Syntax

```
CAPSetStopMode
  [StopMode ':='] <variable (VAR) of capstopmode> ';'
```

Related information

For information about	See
Data type capstopmode	*capstopmode - Defines stop modes for CAP on page 118*

4.1.13 CapWeaveSync - set up signals and levels for weave synchronization

Usage

`CapWeaveSync` is used to setup weaving synchronization signals without sensors. The I/O signals must be defined in EIO.cfg.

Basic example

RAPID program:

```
PROC main()
  ...
  CapWeaveSync \DoLeft:=do_sync_left \LevelLeft:=80
       \DoRight:=do_sync_right \LevelRight:=80;
  ...
ENDPROC
```

In this example the signals `do_sync_left and do_sync_right` are set up with weaving level 80%.

The `CapWeaveSync` instruction should be executed only once, for example, from the startup shelf.

Arguments

```
CapWeaveSync [\Reset] [\DoLeft] [\LevelLeft] [\DoRight]
     [\LevelRight]
```

[\Reset]

Data type: `switch`

Clear weave synchronization data.

[\DoLeft]

Data type: `signaldo`

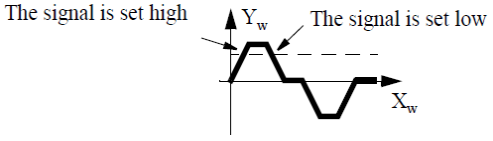
Digital output signal for weave synchronization on the left weave cycle.

[\LevelLeft]

Data type: `num`

The coordination position on the left side of the weaving pattern. The value specified is a percentage of the width on the left of the weaving centre. When weaving is carried out beyond this point, a digital output signal is automatically set high (if the signal is defined).

This type of coordination can be used for seam tracking using Through-the-Arc Tracker.



xx1200000176

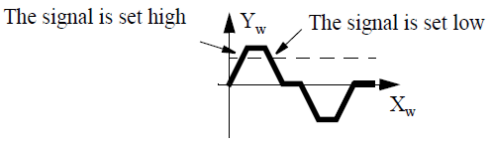
[\LevelLeft]

Data type: `num`

Continues on next page

The coordination position on the left side of the weaving pattern. The value specified is a percentage of the width on the left of the weaving centre. When weaving is carried out beyond this point, a digital output signal is automatically set high (if the signal is defined).

This type of coordination can be used for seam tracking using Through-the-Arc Tracker.



xx1200000176

[\DoRight]

Data type: `signaldo`

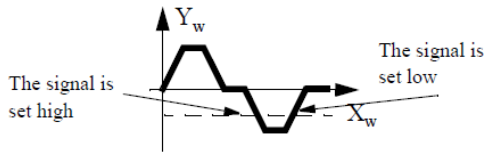
Digital output signal for weave synchronization on the right weave cycle.

[\LevelRight]

Data type: `num`

The coordination position on the right side of the weaving pattern. The value specified is a percentage of the width on the right of the weaving centre. When weaving is carried out beyond this point, a digital output signal is automatically set high (provided the signal is defined).

This type of coordination can be used for seam tracking using Through-the-Arc Tracker.



xx1200000177

Program execution

The defined signals are checked and set when running without a sensor.

Limitations

The signals must be defined in EIO.cfg.

It is not possible to use only either level or corresponding signal. It will not result in errors when loading the RAPID file, but it will result in RAPID run-time errors for the instruction `CapWeaveSynch`.

Syntax

```
CapWeaveSync
  ['\' Reset]
  [DoLeft ':=' < expression (IN) of signaldo >]
  [LevelLeft ':=' < expression (IN) of num >]
  [DoRight ':=' < expression (IN) of signaldo >]
  [LevelRight ':=' < expression (IN) of num >] ';'
```

Continues on next page

4.1.13 CapWeaveSync - set up signals and levels for weave synchronization

Continuous Application Platform (CAP)
Continued

Related information

For information about	See
`capweavedata` data type	*capweavedata - Weavedata for CAP on page 122*

4.1.14 ICap - connect CAP events to trap routines

Usage

`ICap` is used to connect an interrupt number (which is already connected to a trap routine) with a specific CAP Event, see Arguments below for a listing of available Events. When using `ICap`, an association between a specific process event and a user defined Trap routine is created. In other words, the Trap routine in question is executed when the associated CAP event occurs.

We recommend placing the traps in a background task.

Basic example

Below is an example where the CAP Event CAP_START is associated with the trap routine `start_trap`.

```
VAR intnum start_intno:=0;
...

TRAP start_trap
  ! This routine will be executed when the event CAP_START is
       reported from the core
  ! Do what you want to do
ENDTRAP

PROC main()
  IDelete start_intno;
  CONNECT start_intno WITH start_trap;
  ICap start_intno, CAP_START;
  CapL p1, v100, cdata, weavestart, weave, z50, gun1;
ENDPROC
```

Arguments

```
ICap Interrupt Event
```

Interrupt

Data type: `intnum`

The interrupt identity. This should have previously been connected to a trap routine by means of the instruction `CONNECT`.

Event

Data type: `num`

The CAP event number to be associated with the interrupt. These events are predefined constants.

Continues on next page

4.1.14 ICap - connect CAP events to trap routines

Continuous Application Platform (CAP)
Continued

Available CAP events

See section *Coupling between phases and events* in *Application manual - Continuous Application Platform*.

Events	Phase	Event number	Description
CAP_START		0	This event occurs as soon as the CAP process is started.
START_PRE	PRE	1	This event occurs when the supervision of the PRE-phase, if present, is activated. If using a *flying start* no event is distributed, because there is a TCP movement already. At a restart this event is distributed.
PRE_STARTED	PRE	2	This event occurs when all the requirements of the PRE Supervision list are fulfilled, that is, when the PRE_START-phase is started. If using a *flying start* no event is distributed, because there is a TCP movement already. At a restart this event is distributed.
START_MAIN	START	3	This event occurs when the PRE_START-phase is ended and the MAIN-phase is started.
MAIN_STARTED	START	4	This event occurs when all conditions of the START Supervision list are fulfilled, that is, when the MAIN-phase is started.
STOP_WEAVESTART	MAIN	5	This event occurs, before each weave start - but only if weave start is ordered. If using a *flying start* no event is distributed, because there is a TCP movement already. At a restart this event is distributed.
WEAVESTART_REGAIN	MAIN	6	This event occurs when the robot has regained back to the path after a weave start. If using a *flying start* no event is distributed, because there is a TCP movement already. At a restart this event is distributed.
MOTION_DELAY	MAIN	7	This event occurs after the delay, if any, of motion start. If using a *flying start* no event is distributed, because there is a TCP movement already. At a restart this event is distributed.
STARTSPEED_TIME	MAIN	8	This event occurs when the time to use *Start Speed* runs out and it is time to switch to main motion data.
MAIN_MOTION	MAIN	9	This event occurs when main motion is activated with the process running.
MOVE_STARTED	MAIN	10	This event occurs as soon as the robot starts moving along the process path. If using a *flying start* no event is distributed, because there is a TCP movement already. At a restart this event is distributed.
RESTART	MAIN	11	This event occurs when restart is ordered.

Continues on next page

4.1.14 ICap - connect CAP events to trap routines
Continuous Application Platform (CAP)
Continued

Events	Phase	Event number	Description
NEW_INSTR	MAIN	12	This event occurs when a new `CapL` or `CapC` instruction is fetched from the RAPID program.
AT_POINT	MAIN	13	This event occurs at every robtarget on the process path except the start and finish point.
AT_RESTARTPOINT	MAIN	14	This event occurs when the robot has jogged back, the restart distance, on the process path after a stop.
LAST_SEGMENT	MAIN	15	This event occurs at the starting point of the last segment.
PROCESS_END_POINT	MAIN	16	This event occurs when the robot reaches the end point of the process, that is, where the process is supposed to be ended. If using a *flying end* no event is distributed.
END_MAIN	END_MAIN	17	This event occurs at the point on the process path where supervision of the end sequence is started, that is, when the robot reaches the end point of the process.
MAIN_ENDED	END_MAIN	18	This event occurs when all conditions of the END_MAIN supervision list are fulfilled, that is, when the main process is considered ended.
PATH_END_POINT		19	This event occurs when the robot reaches the end point of the path, that is, the fine point or the middle of the zone (for *flying end*) in the last CAP instruction.
PROCESS_ENDED		20	This event occurs only when both the process is ended at the fine point or the middle of the zone (for *flying end*) in the last CAP instruction.
END_POST1	END_POST1	21	This event occurs when it is time to end the POST1 phase, that is, when it is time to change from the POST1 to the POST2-phase. If using a *flying end* no event is distributed.
POST1_ENDED	END_POST1	22	This event occurs when all the conditions of the END_POST1 supervision list are fulfilled, that is, when the POST1 phase is successfully ended and the POST2 phase is started. If using a *flying end* no event is distributed.
END_POST2	END_POST2	23	This event occurs when the POST2 phase is at an end, that is, when it is time to finally finish the process. If using a *flying end* no event is distributed.
POST2_ENDED	END_POST2	24	This event occurs when all the conditions of the END_POST2 supervision list are fulfilled, that is, when the POST2 phase, and thus the whole process, is successfully ended. If using a *flying end* no event is distributed.

Continues on next page

4.1.14 ICap - connect CAP events to trap routines

Continuous Application Platform (CAP)

Continued

Events	Phase	Event number	Description
CAP_STOP		25	This event is a required event. If any other event is used, this event must be defined too. The event/trap is executed as soon as possible after the controller is stopped due to an error or a program stop. An error can be a recoverable error detected in CAP, a fatal error detected in CAP or an internal error stopping the controller. The code executed in this trap should take all external equipment to a safe state, for example, reset all external I/O-signals.
CAP_PF_RESTART	MAIN	26	This event occurs when restart is ordered.
EQUIDIST	MAIN	27	This event is sent, if it is ordered with the instruction `CapEquiDist`.
AT_ERRORPOINT	MAIN	28	This event occurs after restart, when the TCP reaches the position of the supervision error.
FLY_START	MAIN	29	This event occurs when using *flying start*. This event is only available with *flying start*.
FLY_END	MAIN	30	This event occurs when using *flying end*. This event is only available with *flying end*.
LAST_INSTR_ENDED	MAIN	31	This event occurs when RAPID execution of the last CAP instruction is finished during *flying end*. This event is only available with *flying end*.
END_PRE	PRE	32	This event occurs when the supervision of the PRE-phase, if present, is activated. If using a *flying start* no event is distributed, because there is a TCP movement already. At a restart this event is distributed.
PRE_ENDED	PRE	33	This event occurs when the supervision of the PRE-phase, if present, is activated. If using a *flying start* no event is distributed, because there is a TCP movement already. At a restart this event is distributed.
START_POST1	POST1	34	This event occurs when the supervision of the POST1-phase, if present, is activated. If using a *flying start* no event is distributed, because there is a TCP movement already. At a restart this event is distributed.
POST1_STARTED	POST1	35	This event occurs when the supervision of the POST1-phase, if present, is activated. If using a *flying start* no event is distributed, because there is a TCP movement already. At a restart this event is distributed.

Continues on next page

Events	Phase	Event number	Description
START_POST2	POST2	36	This event occurs when the supervision of the POST1-phase, if present, is activated. If using a *flying start* no event is distributed, because there is a TCP movement already. At a restart this event is distributed.
POST2_STARTED	POST2	37	This event occurs when the supervision of the POST1-phase, if present, is activated. If using a *flying start* no event is distributed, because there is a TCP movement already. At a restart this event is distributed.

Limitations

The same variable for interrupt identity cannot be used more than once, without first deleting it. Interrupts should therefore be handled as shown in one of the alternatives below.

```
PROC setup_events ()
  VAR intnum start_intno;
  IDelete start_intno;
  CONNECT start_intno WITH start_trap;
  ICap start_intno, CAP_START;
ENDPROC
```

All activation of interrupts is done at the beginning of the program. These instructions are then kept outside the main flow of the program. The `ICap` instruction should be executed only once, for example, from the startup system event routine. A recommendation is that the traps should be placed in a background task.

Syntax

```
ICap
  [Interrupt ':='] < variable (IN) of intnum > ','
  [Event ':='] < variable (IN) of num > ';'
```

Related information

For information about	See

`CONNECT`
`IDelete`
`intnum`
	Technical reference manual - RAPID Instructions, Functions and Data types

4.1.15 InitSuperv - Reset all supervision for CAP

Usage

`InitSuperv` is used to initiate CAP supervision. This means that all supervision lists will be cleared and all I/O subscriptions will be removed.

Example

```
PROC main()
  InitSuperv;
  SetupSuperv diWR_EST, ACT,SUPERV_MAIN;
  SetupSuperv diGA_EST, ACT,SUPERV_MAIN;
  CapL p2, v100, cdata1, weavestart, weave,fine, tWeldGun;
ENDPROC
```

`InitSuperv` is used to clear all supervision lists before setting up new supervision.

Limitations

The `InitSuperv` instruction should be executed only once, for example, from the startup shelf.

Syntax

```
InitSuperv ';'
```

Related information

For information about	See
`SetupSuperv` instruction	*SetupSuperv - Setup conditions for signal supervision in CAP on page 101*
`RemoveSuperv` instruction	*RemoveSuperv - Remove condition for one signal on page 99*

4.1.16 IPathPos - Get center line robtarget when weaving

Usage

`IPathPos` is used to retrieve the position of the center line during weaving with CAP.

This function is mainly used together with the tracking functionality. It is necessary to activate weaving and the synchronization signals on both the left side and the right side.

Basic example

```
connect intpt, TRP_ipathpos IPathPos p_robt, sen_pos, intpt;
```

When `p_robt` gets a new calculated value, the interrupt `intpt` will be sent, and the TRAP `TRP_ipathpos` will be executed.

Arguments

```
IPathPos p_robt, sen_pos, intpt [\NoDispl] [\EOffs]
```

p_robt

Data type: `robtarget`

`p_robt` keeps the latest value of the calculated `robtarget`.

sen_pos

Data type: `pos`

`sen_pos` is not used.

intpt

Data type: `intno`

`intpt` specifies the interrupt that will be received each time a new value is assigned to `p_robt`.

[\NoDispl]

Data type: `switch`

If `\NoDispl` is specified, the value returned in the `PERS p_robt` will not include any displacement that might be specified using the RAPID instructions `PDispSet` and `PDispOn`.

[\EOffs]

Data type: `switch`

If `[\EOffs]` is specified, the value returned in the `PERS p_robt` will include any offset specified using the RAPID instruction `EOffsSet`.

Limitations

It is necessary to activate weaving and weave synchronization (with or without tracking).

Syntax

```
IPathPos
  [p_robt ':='] < persistent (PERS) of robtarget > ','
```

Continues on next page

4.1.16 IPathPos - Get center line robtarget when weaving

Continued

```
[sen_pos ':='] < persistent (PERS) of pos > ','
[Interrupt ':='] < variable (IN) of intnum >
['\' EOffs ]
['\' NoDispl ] ';'
```

Related information

For information about	See
`CapWeaveSync` instruction	*CapWeaveSync - set up signals and levels for weave synchronization on page 88*
`CapAPTrSetup` instruction	*CapAPTrSetup - Setup an At-Point-Tracker on page 43*
`CapLATrSetup` instruction	*CapLATrSetup - Set up a Look-Ahead-Tracker on page 78*

4.1.17 RemoveSuperv - Remove condition for one signal

Usage

`RemoveSuperv` is used to remove conditions added by `SetupSuperv` from supervision.

Basic example

```
PROC main()
  InitSuperv;
  SetupSuperv diWR_EST, ACT, SUPERV_MAIN \ErrIndSig:= do_WR_Sup;
  SetupSuperv diGA_EST, ACT, SUPERV_MAIN;
  CapL p2, v100, cdata1, weavestart, weave,fine, tWeldGun;
  RemoveSuperv di_Arc_Sup, ACT, SUPERV_START;
ENDPROC
```

Removes the signal *di_Arc_Sup* from the START list.

Arguments

```
RemoveSuperv Signal Condition Listtype
```

Signal

Data type: `signaldi`

Digital signal to remove from supervision list.

Condition

Data type: `num`

The name representing one of the following available conditions:

ACT:	Used for status supervision. Expected signal status during supervision: active. If the signal becomes passive, supervision triggers.
PAS:	Used for status supervision. Expected signal status during supervision: passive. If the signal becomes active, supervision triggers.
POS_EDGE:	Used for handshake supervision. Expected signal status at the end of supervision: active. If the signal does not become active within the chosen timeout, supervision triggers.
NEG_EDGE:	Used for handshake supervision. Expected signal status at the end of supervision: passive. If the signal does not become passive within the chosen timeout, supervision triggers.

Listtype

Data type: `num`

The name representing the number of the different lists (for example, phases in the process):

- SUPERV_PRE
- SUPERV_PRE_START
- SUPERV_END_PRE
- SUPERV_START
- SUPERV_MAIN
- SUPERV_END_MAIN

Continues on next page

4.1.17 RemoveSuperv - Remove condition for one signal

Continuous Application Platform (CAP)
Continued

- SUPERV_START_POST1
- SUPERV_POST1
- SUPERV_END_POST1
- SUPERV_START_POST2
- SUPERV_POST2
- SUPERV_END_POST2

Syntax

```
RemoveSuperv
  [Signal ':='] < variable (VAR) of signaldi > ','
  [Condition ':='] < variable (IN) of num > ','
  [Listtype ':='] < variable (IN) of num >';'
```

Related information

For information about	See
`InitSuperv` instruction	*InitSuperv - Reset all supervision for CAP on page 96*
`SetupSuperv` instruction	*SetupSuperv - Setup conditions for signal supervision in CAP on page 101*

4.1.18 SetupSuperv - Setup conditions for signal supervision in CAP

Usage

`SetupSuperv` is used to set up conditions for I/O signals to be supervised. The conditions are collected in different lists:

- PRE
- PRE_START
- END_PRE
- START
- MAIN
- END_MAIN
- START_POST1
- POST1
- END_POST1
- START_POST2
- POST2
- END_POST2

For more information about supervision lists see *Application manual - Continuous Application Platform*.

As an optional parameter an out signal can be specified. This out signal is set to high, if the given condition fails.

Basic example

```
PROC main()
  InitSuperv;
  SetupSuperv diWR_EST, ACT, SUPERV_MAIN \ErrIndSig:= do_WR_Sup;
  SetupSuperv diGA_EST, ACT, SUPERV_MAIN;
  CapL p2, v100, cdata1, weavestart, weave, fine, tWeldGun;
ENDPROC
```

`SetupSuperv` is used to set up supervision on signals. If signal *diWR_EST* fails during SUPERV_MAIN phase, the digital output signal *do_WR_Sup* is set high.

The `SetupSuperv` instruction should be executed only if supervision data is changed. If the supervision data is never changed, it is a good idea to put it into a module, that is executed from the startup shelf.

Arguments

```
SetupSuperv Signal Condition Listtype [\ErrIndSig]
```

Signal

Data type: `signaldi`

Digital signal to be supervised.

Continues on next page

4.1.18 SetupSuperv - Setup conditions for signal supervision in CAP

Continuous Application Platform (CAP)
Continued

Condition

Data type: num

The name representing one of the following available conditions:

ACT:	Used for status supervision. Expected signal status during supervision: active. If the signal becomes passive, supervision triggers.
PAS:	Used for status supervision. Expected signal status during supervision: passive. If the signal becomes active, supervision triggers.
POS_EDGE:	Used for handshake supervision. Expected signal status at the end of supervision: active. If the signal does not become active within the chosen timeout, supervision triggers.
NEG_EDGE:	Used for handshake supervision. Expected signal status at the end of supervision: passive. If the signal does not become passive within the chosen timeout, supervision triggers.

Listtype

Data type: num

The name representing the number of the different lists (for example, phases in the process):

- SUPERV_PRE
- SUPERV_PRE_START
- SUPERV_END_PRE
- SUPERV_START
- SUPERV_MAIN
- SUPERV_END_MAIN
- SUPERV_START_POST1
- SUPERV_POST1
- SUPERV_END_POST1
- SUPERV_START_POST2
- SUPERV_POST2
- SUPERV_END_POST2

[\ErrIndSig]

Data type: signaldo

Used to indicate which condition that failed if a failure has occurred. When the failure occurs the value on this signal is set to 1. This is an optional parameter.

Program execution

The given signal and its condition is added to the selected list. If a signal fails, the CapL/CapC instruction will report that a supervision error occurred during the specified phase and which signal(s) failed.

Errors

CAP_SPV_LIM

The maximum number of supervisions set up is exceeded.

Continues on next page

CAP_SPV_UNK_LST

The supervision list is unknown.

Limitations

Only digital input signals can be supervised.

Status supervision applies for a complete sequence of CAP instructions (see section *Supervision and process phases* in *Application manual - Continuous Application Platform*).

Syntax

```
SetupSuperv
  [Signal ':='] < variable (VAR) of signaldi > ','
  [Condition ':='] < variable (IN) of num > ','
  [Listtype ':='] < variable (IN) of num >
  [\ErrIndSig ':=' < variable (VAR) of signaldo >] ';'
```

Related information

For information about	See
`InitSuperv` instruction	*InitSuperv - Reset all supervision for CAP on page 96*
`RemoveSuperv` instruction	*RemoveSuperv - Remove condition for one signal on page 99*

4.2 Functions

4.2.1 CapGetFailSigs - Get failed I/O signals

Usage

`CapGetFailSigs` is used to return the names on the signal or signals that failed during supervision of `CapL` or `CapC`.

If supervision of one or several signals fails during the process a recoverable error will be returned from the `CapL`/`CapC` instruction. To determine which signal or signals that failed, `CapGetFailSigs` can be used in an error handler for all cases of supervision errors.

Basic example

```
Stringcopied := CapGetFailSigs(Failstring);
```

`Stringcopied` is assigned the value TRUE if the copy succeeds, and FALSE if it fails.

`Failstring` contains the signals that failed as text. If no string could be copied the string `EMPTY` is returned.

Return value

Data type: `bool`

TRUE or FALSE depending on if the fail string is modified.

Arguments

```
CapGetFailSigs (ErrorNames)
```

ErrorNames

Data type: `string`

`CapGetFailSigs` requires a string variable as input parameter.

Limitations

If many signals in a supervision list failed at the same time, only three of them are reported with `CapGetFailSigs`.

Syntax

```
CapGetFailSigs '('
[ErrorNames ':='] < variable (INOUT) of string >')'
```

A function with a return value of the data type `bool`.

Related information

For information about	See
`InitSuperv` instruction	*InitSuperv - Reset all supervision for CAP on page 96*
`SetupSuperv` instruction	*SetupSuperv - Setup conditions for signal supervision in CAP on page 101*

Continues on next page

For information about	See
`RemoveSuperv` instruction	*RemoveSuperv - Remove condition for one signal on page 99*

4.3 Data types

4.3.1 capaptrreferencedata - Variable setup data for At-Point-Tracker

Usage

`capaptrreferencedata` is used to setup the needed information for the At-Point-Tracker correction process setup by the `CapAPTrSetupAO`, `CapAPTrSetupAI`, and `CapAPTrSetupPERS` instructions.

Components

`reference_y`

Data type: `num`

Defines the reference for the Y position.

`reference_z`

Data type: `num`

Defines the reference for the Z position.

`threshold_y`

Data type: `num`

The difference between the input signal and the `reference_y` value must be greater than the `threshold_y` value for the regulator to react on the change.

`threshold_z`

Data type: `num`

The difference between the input signal and the `reference_z` value must be greater than the `threshold_z` value for the regulator to react on the change.

`gain_y`

Data type: `num`

The difference between the `reference_y` value and the input signal value is scaled with the `gain_y` value.

`gain_z`

Data type: `num`

The difference between the `reference_z` value and the input signal value is scaled with the `gain_z` value.

Structure

```
< data object of capaptrreferencedata >
  < reference_y of num >
  < reference_z of num >
  < threshold_y of num >
  < threshold_z of num >
  < gain_y of num >
  < gain_z of num >
```

Continues on next page

Related information

For information about	See
Instruction `CapAPTrSetupAI`	*CapAPTrSetupAI - Setup an At-Point-Tracker controlled by analog input signals on page 46*
Instruction `CapAPTrSetupAO`	*CapAPTrSetupAO - Setup an At-Point-Tracker controlled by analog output signals on page 49*
Instruction `CapAPTrSetupPERS`	*CapAPTrSetupPERS - Setup an At-Point-Tracker controlled by persistent variables on page 52*
Sensor Interface	*Application manual - Controller software IRC5*

4.3.2 capdata - CAP data

Usage

`capdata` contains all data necessary for defining the behavior of the CAP process.

Components

start_fly

Flying start

Data type: `bool`

Defines whether or not flying start is used:

Value	Consequence
TRUE	flying start is used
FALSE	flying start is NOT used

Flying start means that the robot motion is started before the process is started. The process is then started on the run (see *flypointdata - Data for flying start/end on page 130*).

end_fly

Flying end

Data type: `bool`

Defines whether or not flying end is used:

Value	Consequence
TRUE	flying end is used
FALSE	flying end is NOT used

Flying end means that the CAP process can be terminated before the robot reaches the end point, thus allowing the robot to leave the process path on the run that is, using a zone point (see *flypointdata - Data for flying start/end on page 130*).

first_instr

First instruction

Data type: `bool`

Defines whether or not a `CapL/CapC` instruction is the first instruction in a sequence of `CapL/CapC` instructions:

Value	Consequence
TRUE	this is the first instruction in a sequence of `CapL/CapC` instructions
FALSE	this is not the first instruction in a sequence of `CapL/CapC` instructions

last_instr

Last instruction

Data type: `bool`

Continues on next page

Defines whether or not a CapL/CapC instruction is the last instruction in a sequence of CapL/CapC instructions:

Value	Consequence
TRUE	this is the last instruction in a sequence of CapL/CapC instructions
FALSE	this is not the last instruction in a sequence of CapL/CapC instructions

restart_dist

Restart distance, unit: mm

Data type: num

Defines the distance the robot has to back along the path, when it is restarted after having encountered a stop when a CAP process was active.

In MultiMove systems all synchronized robots must use the same restart distance.

speed_data

Speed data for CAP

Data type: capspeeddata

Defines all CAP data concerning speed (see *capspeeddata - Speed data for CAP on page 116*).

start_fly_point

Data type: flypointdata

These data are only taken into account when start_fly is TRUE.

Defines flying start information for the CAP process (see *flypointdata - Data for flying start/end on page 130*.)

end_fly_point

Data type: flypointdata

These data are only taken into account when end_fly is TRUE.

Defines flying end information for the CAP process (see *flypointdata - Data for flying start/end on page 130*.)

sup_timeouts

Data type: supervtimeouts

Defines the timeouts used for all handshake supervision phases (see *supervtimeouts - Handshake supervision time outs on page 136* and section *Supervision* in *Application manual - Continuous Application Platform*).

proc_times

Data type: processtimes

Defines the timeouts used for the status supervision phases PRE, POST1, and POST2 (see *processtimes - process times on page 133* and section *Supervision and process phases* in *Application manual - Continuous Application Platform*).

block_at_restart

Data type: restartblkdata

Continues on next page

4.3.2 capdata - CAP data

Continuous Application Platform (CAP)

Continued

Defines the behavior of the CAP process during a restart (see *restartblkdata - blockdata for restart on page 134*).

Structure

```
< data object of capdata >
  < start_fly of bool >
  < end_fly of bool >
  < first_instr of bool >
  < last_instr of bool >
  < restart_dist of num >
  < speed_data of capspeeddata >
    < fly_start of num >
    < start of num >
    < startspeed_time of num >
    < startmove_delay of num >
    < main of num >
    < fly_end of num >
  < start_fly_point of flypointdata >
    < from_start of bool >
    < time_before of num >
    < distance of num >
  < end_fly_point of flypointdata >
    < from_start of bool >
    < time_before of num >
    < distance of num >
  < sup_timeouts of supervtimeouts >
    < pre_cond of num >
    < start_cond of num >
    < end_main_cond of num >
    < end_post1_cond of num >
    < end_post2_cond of num >
  < proc_times of processtimes >
    < pre of num >
    < post1 of num >
    < post2 of num >
  < block_at_restart of restartblkdata >
    < weave_start of bool >
    < motion_delay of bool >
    < pre_phase of bool >
    < startspeed_phase of bool >
    < post1_phase of bool >
    < post2_phase of bool >
```

Related information

	Described in:
`capspeeddata` **data type**	*capspeeddata - Speed data for CAP on page 116*
`flypointdata` **data type**	*flypointdata - Data for flying start/end on page 130*

Continues on next page

	Described in:
`supervtimeouts` data type	*supervtimeouts - Handshake supervision time outs on page 136*
`processtimes` data type	*processtimes - process times on page 133*
`block_at_restart` data type	*restartblkdata - blockdata for restart on page 134*
`CapL` instruction	*CapL - Linear CAP motion instruction on page 69*
`CapC` instruction	*CapC - Circular CAP motion instruction on page 55*

4.3.3 caplatrackdata - CAP Look-Ahead-Tracker track data

Usage

`caplatrackdata` contains data, with which the user can influence how the `CapL/CapC` instructions incorporate the path correction data generated by a Look-Ahead-Tracker (for example, Laser Tracker). `caplatrackdata` is part of the `captrackdata`.

Components

joint_no

Data type: `num`

Defines the joint type (expressed as a number) the sensor equipment shall use during tracking.

filter

Data type: `num`

Defines the time constant of a low pass filter applied to path corrections. The component may be set to values between 1 and 10 where 1 gives the fastest response (no filtering) to path errors detected by the sensor.

calibframe_no

Data type: `num`

Defines which calibration frame of the three frames defined in `CapLATrSetup`, that shall be used.

Value	Calibration frame	Description
1	calibframe	Mandatory in `CapLATrSetup`
2	calibframe2	Optional in `CapLATrSetup`
3	calibframe3	Optional in `CapLATrSetup`)

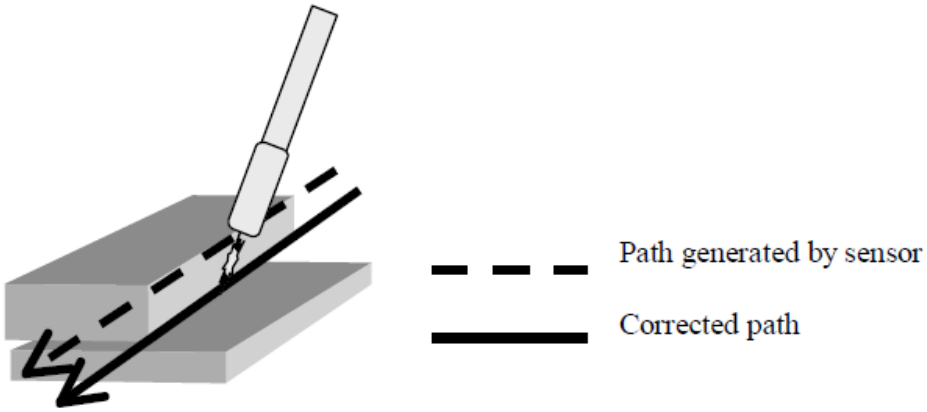
seamoffs_y, seamoffs_z

Data type: `num`

The seam offset components are used to add constant offsets to the sensor generated path (in mm). If for example the sensor considers the upper edge of a

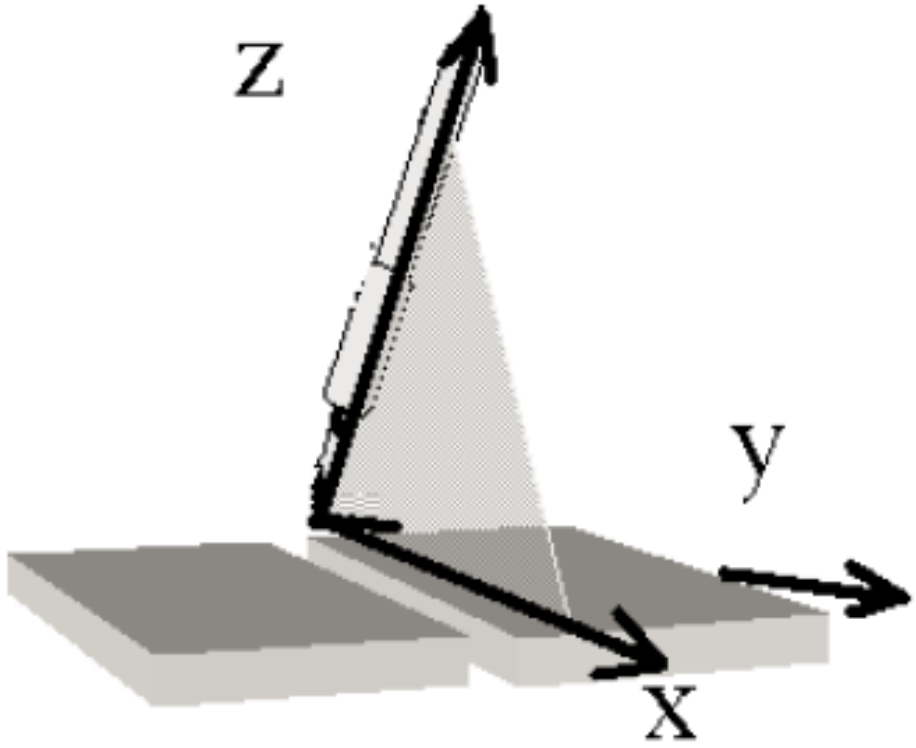
Continues on next page

lap joint to be the correct seam position, as indicated in the figure below, the seam offsets may be used to correct the path.



xx1200000199

The corrections are defined in the path coordinate system, which is right handed.



xx1200000200

- The x-direction is parallel to the path tangent.
- The z-direction is the tool z-vector.
- The x-direction is perpendicular to a plane through the x and z-directions.

seamadapt_y, seamadapt_z

Data type: `num`

Continues on next page

4.3.3 caplatrackdata - CAP Look-Ahead-Tracker track data
Continuous Application Platform (CAP)
Continued

The seam adapt components are similar to the seam offset components. The magnitudes of the offsets are however not given as fixed values. The offsets are calculated as the measured seam gap multiplied by the seam adapt values.

The components are used to adapt the tool offset with respect to the seam to optimize the process for different gap sizes.

track_mode

Data type: `num`

With the `track_mode` component it is possible to selectively influence the tracking behavior of a laser tracker.

Value	Track Mode
0	Normal tracking. y- and z-corrections are both taken into account
1	Tracking as if y-corrections sent by the Laser Tracker were zero. z-corrections are taken into account. [i]
2	Tracking as if z-corrections sent by the Laser Tracker were zero. y-corrections are taken into account. *i*
3	Tracking as if y- and z-corrections sent by the Laser Tracker were zero. *i*
4	y-correction switched off totally, that is, the correction of the y component is set to zero before it is sent to the robot. z-correction is taken into account. [ii]
5	z-correction switched off totally, that is, the correction of the z component is set to zero before it is sent to the robot. y-correction is taken into account. *ii*
6	y- and z-corrections are switched off totally, that is, the correction of the y and the z component is set to zero before it is sent to the robot. *ii*
7	y-correction is faded out, that is, the TCP returns ramped to the programmed y component of the path. z-correction is active.
8	z-correction is faded out, that is, the TCP returns ramped to the programmed z component of the path. y-correction is active.
9	y- and z-corrections are faded out, that is, the TCP returns ramped to the programmed path.
10	y-correction is faded in, that is, the TCP returns ramped to the programmed y component of the path. z-correction is active.
11	z-correction is faded in, that is, the TCP returns ramped to the programmed z component of the path. y-correction is active.
12	y- and z-corrections are faded in, that is, the TCP returns ramped to the programmed path.
13	Tracking as if y-corrections sent by the Laser Tracker were zero. z-corrections are taken into account. The difference to `track_mode` 1 is, that the mode starts at the robot TCP position and not at the sensor TCP position. *i*
14	Tracking as if z-corrections sent by the Laser Tracker were zero. y-corrections are taken into account. The difference to `track_mode` 2 is that the mode starts at the robot TCP position and not at the sensor TCP position. *i*

Continues on next page

Value	Track Mode
15	Tracking as if y- and z-corrections sent by the Laser Tracker were zero. The difference to `track_mode` 3 is that the mode starts at the robot TCP position and not at the sensor TCP position. [i]

i For `track_mode` 1, 2, or 3, the accumulated correction from the previous `CapL/CapC` instruction will be preserved for y or/and z and passed on to the next `CapL/CapC` instruction. This is the case during the hole lifetime of the CAP process. A new CAP process will be unaffected

ii For `track_mode` 4, 5, or 6, the sensor readings are accumulated even though y- and/or z-correction is set to zero before sending to the robot. That means, a 'dip' might occur in the beginning and in the end of the `CapL/CapC` instruction.

Basic examples

```
PERS captrackdata captrack := ["laser1:",50,[1,10,1,0,0,0,0,0]]
CapL p1, v200, cd, wsd, cwd, z20, tWeldGun \Track:=captrack;
```

Syntax

```
< data object of caplatrackdata >
  < joint_no of num >
  < filter of num >
  < calibframe_no of num >
  < seamoffs_y of num >
  < seamoffs_z of num >
  < seamadapt_y of num >
  < seamadapt_z of num >
  < track_mode of num >
```

Related information

	Described in:
`captrackdata` data type	*captrackdata - CAP track data on page 119*

4.3.4 capspeeddata - Speed data for CAP

Usage

`capspeeddata` is used to define all data concerning velocity for a CAP process - it is part of `capdata` and defines all velocity data and process times needed for a CAP process:

- velocity and how long this velocity shall be used at the start of the CAP process,
- delay for the movement of the robot relative the start of the CAP process,
- velocity for the CAP process,

The velocity is restricted by the performance of the robot. This differs, depending on the type of robot and the path of movement.

Components

fly_start

Data type: `num`

Not used.

start

Data type: `num`

Velocity (in mm/s) used at the start of the CAP process.

startspeed_time

Data type: `num`

The time (in seconds) to run at `start` velocity.

startmove_delay

Data type: `num`

The time (in seconds) that the robot movement is delayed relative the start of the CAP process.

main

Data type: `num`

The main CAP process velocity (mm/s).

fly_end

Data type: `num`

Not used.

Structure

```
< data object of capspeeddata >
  < fly_start of num >
  < start of num >
  < startspeed_time of num >
  < startmove_delay of num >
  < main of num >
  < fly_end of num >
```

Continues on next page

Related information

	Described in:
`capdata` data type	*capdata - CAP data on page 108*

4.3.5 capstopmode - Defines stop modes for CAP

Usage

`capstopmode` is used to define the available stop modes in CAP.

Basic examples

The following example illustrates the data type `capstopmode`.

Example 1

```
CAPSetStopMode SMOOTH_STOP_ON_PATH;
```

Predefined values

Value	Description
`SMOOTH_STOP_ON_PATH`	The robot movement is stopped smoothly without leaving the programmed path.
`QUICK_STOP_ON_PATH`	The robot movement is stopped as fast as possible without leaving the programmed path.
`EMERGENCY_STOP`	The robot movement is stopped as fast as possible regardless if it leaves the programmed path.

Characteristics

`capstopmode` is an alias data type for `num` and consequently inherits its characteristics.

Related information

	Described in:
Instruction `CAPSetStopMode`	*CAPSetStopMode - Set the stop mode for execution errors on page 87*

4.3.6 captrackdata - CAP track data

Usage

`captrackdata` provides the `CapL/CapC` instructions with all data necessary for path correction with a Look-Ahead- or At-Point-Tracker. The data is passed to the `CapL/C` instructions with use of the optional argument `\Track`.

The component `device` determines, which type of tracker is to be used. `captrackdata` cannot be changed within a sequence of `CapL/CapC` instructions. The component `device` is set by the first `CapL/C` instruction - if it is different in the remaining `CapL/C` instructions of the same sequence of `CapL/CapC` instructions, it will not have any effect.

To be able to change the `captrackdata` to be used in a `CapL/CapC` instruction, the sequence has to be terminated first by setting the component `last_inst` to `TRUE` in `capdata`.

If the `\Track` is not present in the first `CapL/C` instruction and all following in the same sequence of `CapL/CapC` instructions, no correction will be applied.

Components

device

Sensor device

Data type: `string`

Defines, to which device the sensor is connected, that shall be used in the `CapL/CapC` instructions to generate path corrections.

max_corr

Maximum allowed path correction

Data type: `num`

Defines the maximum path correction allowed.

For Look-Ahead trackers:

- If the TCP offset due to path corrections is more than `max_corr` and `\WarnMaxCorr` was specified in `CapLATrSetup`, the robot will continue its path but the applied path correction will not exceed `max_corr`.
- If `\WarnMaxCorr` was not specified, a track error is reported and program execution is stopped.

For At-Point trackers:

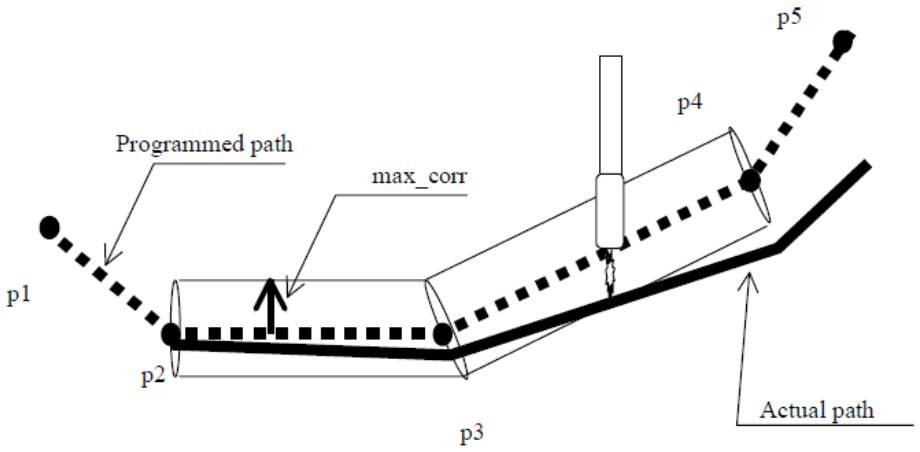
- If the TCP offset due to path corrections is more than `max_corr`, a track error is reported and program execution is stopped.

Continues on next page

4.3.6 captrackdata - CAP track data

Continuous Application Platform (CAP)
Continued

The figure shows the tool in a position relative to the programmed path where a `max_corr` track error would be reported. Unit: mm



xx1200000198

la_trackdata

Look-Ahead-Tracker track data

Data type: `caplatrackdata`

Defines tracking data, that are specific for Look-Ahead-Trackers (for example, laser trackers).

Basic examples

SIO.cfg:

```
COM_TRP:
-Name "SCOUT:" -Type "RTP1"
-Name "digi-ip:" -Type "SOCKDEV" -PhyChannel "LAN1" -RemoteAdress
     "192.168.125.5"
```

RAPID program:

```
PERS captrackdata captrack1 := ["digi-ip:",50,[1,10,1,0,0,0,0,0]];
CONST string laser := "digi-ip:";
PERS pose pose1 := [[137.867,-326.31,18.5],
     [0.640984,0.766438,0.0348674,0.0223137]];
PROC main()
  VAR pos sensorPos;
  CapLATrSetup laser, pose1, pos \SensorFreq:=10 \CorrFilter:=5
       \MaxBlind:=100 \MaxIncCorr:=2;
  WriteVar laser, 6, 1;
  ! sensor ON
  CapL p1, v200, cd, wsd_event, cwd, z20, tWeldGun
       \Track:=captrack1;
  CapC p2, p3, v200, cd2, wsd, cwd, z20, tWeldGun \Track:=captrack1;
  CapL p4, v200, cd3, wsd, cwd, fine, tWeldGun \Track:=captrack1;
  WriteVar laser, 6, 0;
  ! sensor OFF
ENDPROC
```

Continues on next page

Syntax

```
< data object of captrackdata >
  < device of string >
  < max_corr of num>
  < la_trackdata of caplatrackdata >
```

Related information

	Described in:
`caplatrackdata`	*caplatrackdata - CAP Look-Ahead-Tracker track data on page 112*
`CapAPTrSetup`	*CapAPTrSetup - Setup an At-Point-Tracker on page 43*
`CapLATrSetup`	*CapLATrSetup - Set up a Look-Ahead-Tracker on page 78*

4.3.7 capweavedata - Weavedata for CAP

Usage

`capweavedata` is used to define weaving for a CAP process during its MAIN phase (see *Application manual - Continuous Application Platform*).

Description of weaving

Weaving is superimposed on the basic path of the process. That means, that the process speed (defined in `capspeeddata`) is kept as defined, but the TCP speed is increased unless the physical robot limitations are reached.

Available weaving types:

- geometric weaving: most accurate shape
- wrist weaving: only robot axis 6 is used for weaving
- rapid weaving axis 1-3: only robot axis 1, 2 and 3 are used for weaving
- rapid weaving axis 4-6: only robot axis 4, 5 and 6 are used for weaving

Available weaving shapes:

- Zig-zag weaving
- V-shaped weaving
- Triangular weaving
- Circular weaving

All `capweavedata` components apply to the MAIN phase.

Components

The path coordinate system is defined by:

- X: path/movement direction
- Z: tool z-direction
- Y: perpendicular to both X and Z as to build a right-handed coordinate system

active

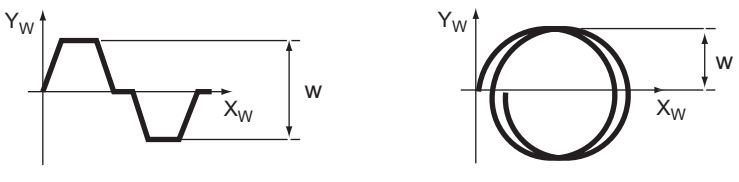
Data type: `bool`

Value	Description
TRUE	Perform weaving during the MAIN phase of the CAP process
FALSE	Do NOT perform weaving during the MAIN phase of the CAP process

width

Data type: `num`

For circular weaving, width is the radius of the circle (W in the following figure). For all other weaving shapes, width is the total amplitude of the weaving pattern.



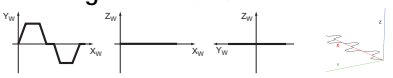
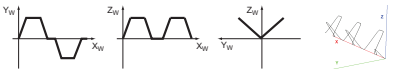
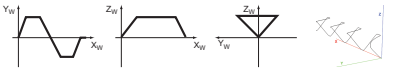
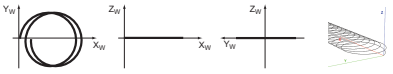
xx1200000721

Continues on next page

shape

Data type: num

The shape of the weaving pattern in the main phase.

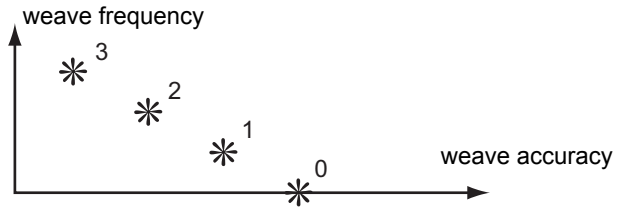
Value	Shape geometry	Result
0	No weaving	
1	Zig-zag weaving	Weaving horizontal to the seam  xx1200000714
2	V-shaped weaving	Weaving in the shape of a "V", vertical to the seam  xx1200000715
3	Triangular weaving	A triangular shape, vertical to the seam  xx1200000716
4	Circular weaving (Only available with geometric weaving, weaving type 0)	A circular shape, vertical to the seam  xx1200000717

type

Data type: num

Defines what axes are used for weaving during the MAIN phase

Specified value	Weaving type
0	Geometric weaving. All axes are used during weaving.
1	Wrist weaving.
2	Rapid weaving. Axes 1, 2, and 3 used.
3	Rapid weaving. Axes 4, 5, and 6 used.



xx1200000718

Continues on next page

4.3.7 capweavedata - Weavedata for CAP
Continuous Application Platform (CAP)
Continued

Note

All robots, that use TrueMove/QuickMove second generation have the following changed behavior for the different weaving types available in RW Arc and CAP, compared to TrueMove/QuickMove first generation:

- Geometric weaving: There is no change.
- Wrist weaving: uses mainly the wrist axes (4, 5, and 6) but small corrections can also be added to the main axes to be able to keep the pattern in the desired plane.
- Rapid weaving: In TrueMove/QuickMove second generation both geometric weaving and wrist weaving have highly improved performance. Therefore Rapid weaving (both types) is not necessary as a special weaving type any more.

  Rapid weaving axis 1, 2, and 3 is the same as geometric weaving.

  Rapid weaving axis 4, 5, and 6 is the same as wrist weaving.

  The weaving types are still available for backward compatibility.

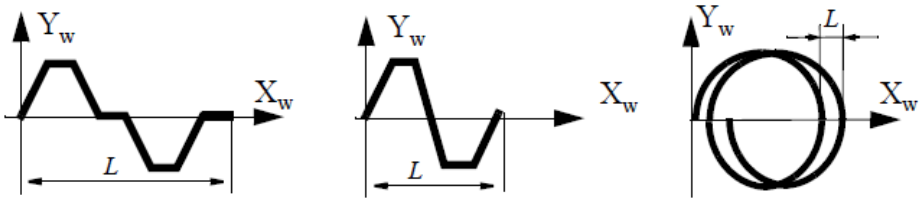
The system uses TrueMove/QuickMove second generation, if there is a switch `dyn_ipol_type 1` in MOC.cfg in the *MOTION_PLANNER* data.

length

Data type: `num`

Defines the length of the weaving cycle in the MAIN phase for geometric weaving (type = 0) and wrist weaving (type = 1). The length argument is not used for the other weaving types.

For circular weaving the `length` component defines the distance between two successive circles (L) if the `cycle_time` argument is set to 0. If `cycle_time` has a value then the length can be displaced. The TCP rotates left with a positive length value, and right with a negative length value.



xx1200000187

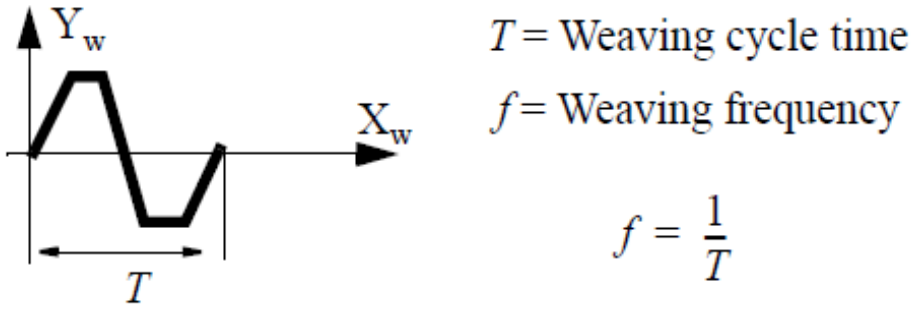
cycle_time

Data type: `num`

Defines the weaving frequency (in Hz) in the MAIN phase for both types of Rapid weaving (type = 2 or 3) and for circular weaving. The `cycle_time` argument is not used for the other weaving types.

Continues on next page

For circular weaving the `cycle_time` argument defines the number of circles per second. The TCP rotates left with a positive `cycle_time` value, and right with a negative `cycle_time` value.



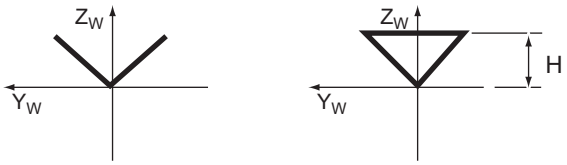
xx1200000188

height

Data type: `num`

Defines the height of the weaving pattern (in mm) during V-shaped and triangular weaving.

Not available for circular weaving.

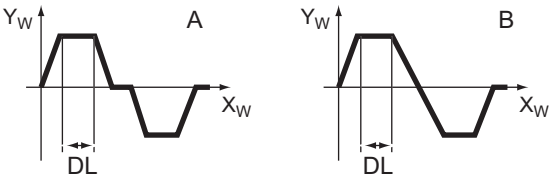


xx1200000722

dwell_left

Data type: `num`

The length of the dwell (DL) used to force the TCP to move only in the direction of the seam at the left turning point of the weave. Not available for circular weaving.



xx1200000723

A	Zigzag and V-shaped weaving
B	Triangular weaving

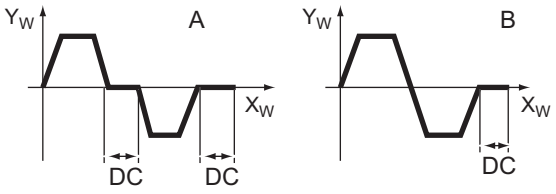
Continues on next page

4.3.7 capweavedata - Weavedata for CAP
Continuous Application Platform (CAP)
Continued

dwell_center

Data type: num

The length of the dwell (DC) used to force the TCP to move only in the direction of the seam at the center point of the weave. Not available for circular weaving.



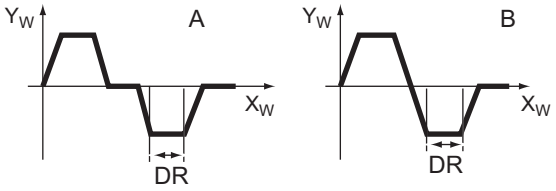
xx1200000724

A	Zigzag and V-shaped weaving
B	Triangular weaving

dwell_right

Data type: num

The length of the dwell (DR) used to force the TCP to move only in the direction of the seam at the right turning point of the weave. Not available for circular weaving.



xx1200000725

A	Zigzag and V-shaped weaving
B	Triangular weaving

dir

Data type: num

The weave direction angle horizontal to the seam. An angle of zero degrees results in a weave vertical to the seam.

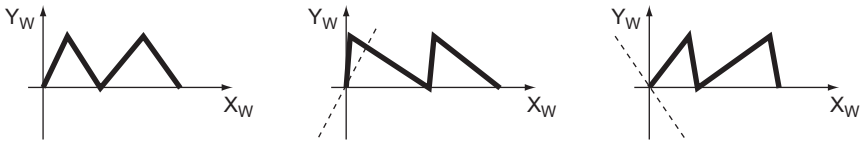
xx1200000726

Continues on next page

tilt

Data type: `num`

The weave tilt angle, vertical to the seam. An angle of zero degrees results in a weave which is vertical to the seam.

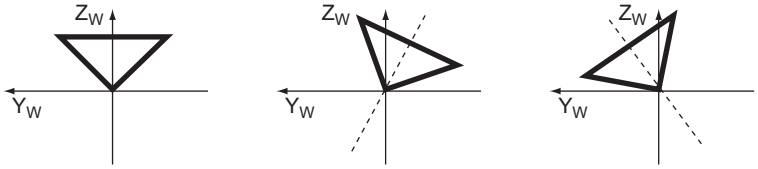


xx1200000727

rot

Data type: `num`

The weave orientation angle, horizontal-vertical to the seam. An angle of zero degrees results in symmetrical weaving.



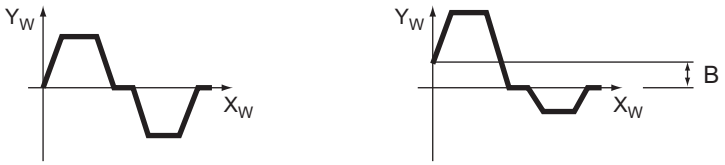
xx1200000728

bias

Data type: `num`

The bias horizontal to the weaving pattern. The bias can only be specified for zig-zag weaving and may not be greater than half the width of the weave. Not available for circular weaving.

The following figure shows zigzag weaving with and without bias (B).



xx1200000729

ptrn_sync_on

Data type: `bool`

Value	Description
TRUE	Send synchronization pulses at the right and left turning points of the weave pattern
FALSE	Do NOT send synchronization pulses at the right and left turning points of the weave pattern

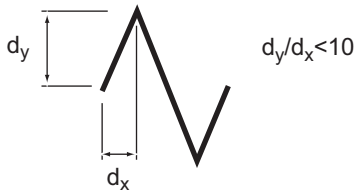
Continues on next page

4.3.7 capweavedata - Weavedata for CAP
Continuous Application Platform (CAP)
Continued

Limitations

The maximum weaving frequency is 2 Hz.

The inclination of the weaving pattern must not exceed the ratio 1:10 (84 degrees). See the following figure.



xx1200000730

Change of `weave_type` in `weavedata` is not possible in zone points, only in fine points. This is the behavior for both *TrueMove* and *QuickMove*, first and second generation.

All robots, that use *TrueMove* or *QuickMove* second generation have the following changed behavior for the different weaving types available in RW Arc, compared to *TrueMove* or *QuickMove* first generation:

- Geometric weaving - There is no change.
- Wrist weaving - uses mainly the wrist axes (4, 5, and 6) but small corrections can also be added to the main axes to be able to keep the pattern in the desired plane.
- Rapid weaving - In *TrueMove* or *QuickMove* second generation both geometric weaving and wrist weaving have highly improved performance. Therefore Rapid weaving (both types) is not necessary as a special weaving type any more.

  Rapid weaving axis 1, 2, and 3 is the same as geometric weaving.

  Rapid weaving axis 4, 5, and 6 is the same as wrist weaving.

  The weaving types are still available for backward compatibility.

The system uses *TrueMove* or *QuickMove* second generation, if there is a switch `dyn_ipol_type 1` in MOC.cfg in the MOTION_PLANNER data (system parameters).

Syntax

```
< data object of capweavedata >
  < active of bool>
  < width of num >
  < shape of num >
  < type of num >
  < length of num >
  < cycle_time of num >
  < height of num >
  < dwell_left of num >
  < dwell_center of num >
  < dwell_right of num >
  < dir of num >
  < tilt of num >
```

Continues on next page

```
< rot of num >
< bias of num >
< ptrn_sync_on of bool >
```

Related information

	Described in:
`capdata` **data type**	*capdata - CAP data on page 108*

4.3.8 flypointdata - Data for flying start/end

Usage

`flypointdata` is used to define all data of flying start or flying end for a CAP process - it is part of `capdata` for both flying start and flying end.

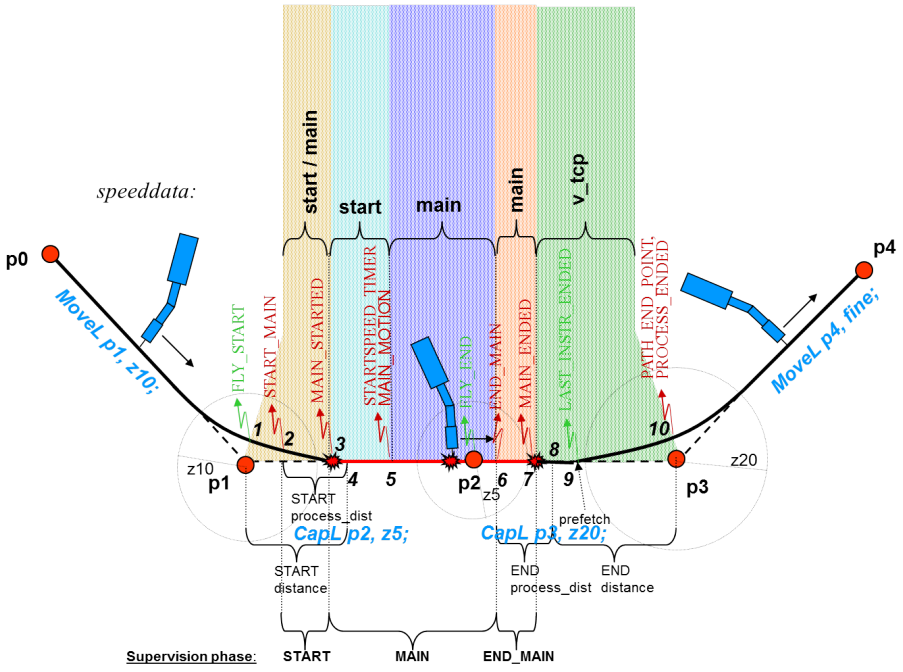
Definitions

`flypointdata` defines data for both flying start and flying end:

- This functionality is only available for CAP.
- Flying start is triggered by the combination of *first instruction = TRUE* and zone point.
- Flying end is triggered by the combination of *last_instr = TRUE* and zone point.
- Weavestart will be ignored.
- If the starting point is a fine point, no flying start will be performed.
- If the end point is a fine point, no flying end will be performed.
- Motion delay will be ignored.
- Restart after an error will work in the same way as usual: there are no specific features for flying start, scrape start is available, if the application process was active, when the error occurred.
- If weaving is activated, the transition in the zone is made by ramping in the weaving pattern starting at the entrance to the zone until the full pattern is reached when the TCP leaves the zone.
- Supervision is active during START phase (with moving TCP), MAIN phase and END_MAIN phase (with moving TCP).
- Backing on the path will be limited to backing to position 4 (see the following figure).
- The user has to adapt distance and the approach and leaving angle to the application process: for example, for arc welding at the point where the arc shall be established (point 4 in the figure) has to be selected in such a way, that it is possible to ignite.
- The distance between position 4 and 6 must not be = 0.
- The `START process_dist` must be equal to or shorter than `START distance`.
- If program execution is stopped and the application process is active (between positions 3 and 6), CAP will behave as usual, that is, backing on path (only if pos. 4 had been passed), weave start, motion delay and movement start timeout are available.
- If program execution is stopped between positions 1 and 3 or between positions 7 and 10, the `CapX` instruction will behave like a `TrigX` instruction.
- The first CAP segment with flying start is recommended to be at least `START distance` long.

Continues on next page

- If the first segment is shorter than `START distance`, but longer than `START process_dist`, the positions 2 and 4 will be moved towards position 1.
- If the first segment is shorter than or equal `START process_dist`, positions 1 and 2 will coincide and position 4 will be at the end of the segment.
- The last CAP segment with flying end is recommended to be at least `END distance + END process_dist` long.
- If the last segment is shorter than `END distance + END process_dist`, but longer than `END process_dist`, the positions 7 and 9 will be moved towards position 10.
- If the last segment is shorter than or equal `END process_dist`, positions 8 and 10 will coincide and position 6 will be at the start of the segment.
- The START phase timeout specified in `capdata` will only be used at restart of the application process.
- If a process error occurs after the prefetch request from motion has arrived at the last CAP instruction (after position 9), that is, PGM is released from the CAP instruction and may continue with the next instruction, an error log message is sent, the process is stopped, *but* the robot movement continues.



xx1200000180

Components

from_start

Data type: `bool`

Not used.

process_dist

Data type: `num`

Continues on next page

4.3.8 flypointdata - Data for flying start/end

Continuous Application Platform (CAP)
Continued

The distance (in mm) within which the process is started (for *flying start*) or ended (for *flying end*).

distance

Data type: `num`

Sets the start/end of the supervision of the CAP process as a distance (in mm) from the start/end point.

Structure

```
< databases of flypointdata >
  < from_start of bool >
  < process_dist of num >
  < distance of num >
```

Related information

	Described in:
`capdata` data type	*capdata - CAP data on page 108*

4.3.9 processtimes - process times

Usage

`processtimes` is used to define the duration times for all status supervision phases in CAP, except phase MAIN, which is defined by the robot movement (see section *Supervision* in *Application manual - Continuous Application Platform*).

`processtimes` is a component of `capdata` and defines the timeout times for the following status supervision phases in CAP:

- PRE_START
- POST1
- POST2

The specified timeout time has to be larger than zero, if supervision should be used during the corresponding status supervision phase in CAP (see section *Supervision and process phases* in *Application manual - Continuous Application Platform*).

Components

pre

Data type: `num`

Defines the duration of the phase PRE_START in seconds. During that time all conditions defined for that phase have to be fulfilled.

post1

Data type: `num`

Defines the duration of the phase POST1 in seconds. During that time all conditions defined for that phase have to be fulfilled.

post2

Data type: `num`

Defines the duration of the phase POST2 in seconds. During that time all conditions defined for that phase have to be fulfilled.

Syntax

```
< data object of processtimes >
  < pre of num >
  < post1 of num >
  < post2 of num >
```

Related information

	Described in:
`capdata` data type	*capdata - CAP data on page 108*

4.3.10 restartblkdata - blockdata for restart

Usage

`restartblkdata` is used to define the behavior of a CAP process at restart.

`restartblkdata` is a component of `capdata` and defines the following for a CAP process at restart, if:

- The robot should execute/block weaving stationary during process restart (`weave_start`).
- Robot movement restart should be delayed or not relative process restart (`motion_delay`).
- PRE phase and PRE_START phase should be executed/blocked (`pre_phase`).
- A velocity different from main velocity should be used or not during start of the process (`startspeed_phase`).
- START_POST1 phase and POST1 phase should be executed/blocked (`post1_phase`).
- START_POST2 phase and POST2 phase should be executed/blocked (`post2_phase`).

Components

weave_start

Data type: `bool`

Value	Description
FALSE	Stationary weaving at restart until the process has started
TRUE	No stationary weaving at restart until the process has started

motion_delay

Data type: `bool`

Value	Description
FALSE	Delay of robot movement at restart after the process has started
TRUE	No delay of robot movement at restart after the process has started

pre_phase

Data type: `bool`

Value	Description
FALSE	Execute PRE phase and PRE_START phase at restart
TRUE	Do NOT execute PRE phase and PRE_START phase at restart

startspeed_phase

Data type: `bool`

Value	Description
FALSE	Move the robot with start speed in the beginning of a restart

Continues on next page

Value	Description
TRUE	Do NOT move the robot with start speed in the beginning of a restart, use main speed directly

post1_phase

Data type: `bool`

Value	Description
FALSE	Execute START_POST1 phase and POST1 phase at restart
TRUE	Do NOT execute START_POST1 phase and POST1 phase at restart

post2_phase

Data type: `bool`

Value	Description
FALSE	Execute START_POST2 phase and POST2 phase at restart
TRUE	Do NOT execute START_POST2 phase and POST2 phase at restart

Syntax

```
< data object of restartblkdata >
  < weave_start of bool >
  < motion_delay of bool >
  < pre_phase of bool >
  < startspeed_phase of bool >
  < post1_phase of bool >
  < post2_phase of bool >
```

Related information

	Described in:
`capdata` data type	*capdata - CAP data on page 108*

4.3.11 supervtimeouts - Handshake supervision time outs

Usage

`supervtimeouts` is used to define timeout times for handshake supervision in CAP.

`supervtimeouts` is a component of `capdata` and defines the timeout times for the following handshake supervision phases in CAP:

- PRE
- END_PRE and START
- END MAIN and START_POST1
- END_POST1 and START_POST2
- END_POST2

If the parameter is set to 0, there is no timeout.

Components

pre_cond

Data type: `num`

Timeout time (in seconds) for the PRE phase conditions to be fulfilled.

start_cond

Data type: `num`

Timeout time (in seconds) for the END_PRE and START phase conditions to be fulfilled.

end_main_cond

Data type: `num`

Timeout time (in seconds) for the END_MAIN and START_POST1 phase conditions to be fulfilled.

end_post1_cond

Data type: `num`

Timeout time (in seconds) for the END_POST1 and START_POST2 phase conditions to be fulfilled.

end_post2_cond

Data type: `num`

Timeout time (in seconds) for the END_POST2 phase conditions to be fulfilled.

Syntax

```
< data object of supervtimeouts >
  < pre_cond of num >
  < start_cond of num >
  < end_main_cond of num >
  < end_post1_cond of num >
  < end_post2_cond of num >
```

Continues on next page

Related information

	Described in:
`capdata` data type	*capdata - CAP data on page 108*

4.3.12 weavestartdata - weave start data

Usage

`weavestartdata` is used to control stationary weaving during start and restart of a process in CAP.

`weavestartdata` is a component of `capdata` and defines the properties of stationary weaving at start or restart of a CAP process:

- if there shall be stationary weaving at start (`active`)
- width of stationary weaving (`width`)
- direction relative path direction (`dir`)
- frequency of stationary weaving (`cycle_time`)

Stationary weaving uses always geometric weaving with zig-zag pattern, see *capweavedata - Weavedata for CAP on page 122*.

Components

active

Data type: `bool`

Value	Description
TRUE	Perform stationary weaving at start of a CAP process
FALSE	Do NOT perform stationary weaving at start of a CAP process

width

Data type: `num`

Defines the amplitude of stationary weaving (mm).

dir

Data type: `num`

Defines the direction of stationary weaving relative to the path direction (degrees). Zero degrees means weaving perpendicular to both the path and the z-coordinate of the tool.

cycle_time

Data type: `num`

Defines the total time (in seconds) for a complete cycle of stationary weaving, that is, it defines the weaving frequency. The stationary weaving will last until the process has started, that is, the supervision criteria of the START_MAIN phase are fulfilled.

Syntax

```
< data object of weavestartdata >
  < active of bool >
  < width of num >
  < dir of num >
  < cycle_time of num >
```

Continues on next page

Related information

	Described in:
`capdata` data type	*capdata - CAP data on page 108*

This page is intentionally left blank

Index

ABB AB
Robotics & Discrete Automation
S-721 68 VÄSTERÅS, Sweden
Telephone +46 (0) 21 344 400

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

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

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

abb.com/robotics

3HAC050990-001, Rev D, en