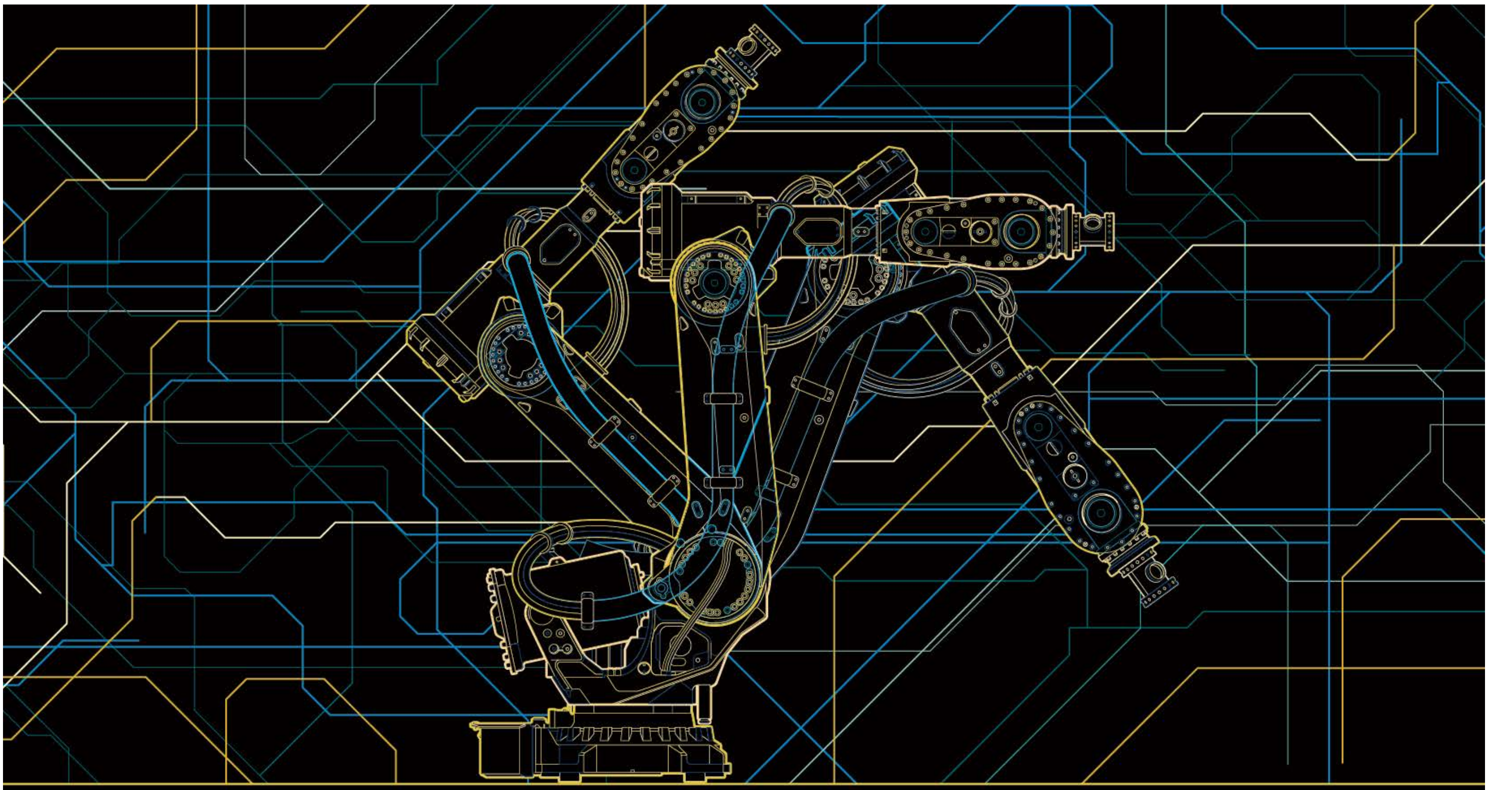


Application manual
Discrete application platform

Power and productivity
for a better world™

Trace back information:
Workspace RW 6-0 version a3
Checked in 2014-11-11
Skribenta version 4.1.349

Application manual
Discrete application platform

Document ID: 3HAC050994-001

Revision: -

ABB AB
Robotics Products
Se-721 68 Västerås
Sweden

Table of contents

This page is intentionally left blank

1 Discrete application summary

Overview

The option Discrete Application Platform (DAP) provides a software framework for application software engineers.

The package is an optimal tool for fast and straight forward development by providing a setup of specialized methods and datatypes in RAPID. It encapsulates motion and process execution in one RAPID-instruction call (see `EG1ML`/`EG1MJ`).

The use of the package reduces application development costs and ensures a high quality level and optimal use of the IRC5-system.

The Discrete application is tailored for applications similar to SpotWelding which with the following environment:

- Discrete Application combines finepoint positioning with execution of up to four parallel processes.
- The process is specialized for monitoring an external process device.
- Supports encapsulation of the process and motion in shell-routines provided to the end user.

The package is designed to have an internal kernel administrating the fast and quality secured process sequence skeleton. It calls RAPID routines which the application writer has to prepare to fulfill his specific task. It is up to the writer of the application how much flexibility to leave to the end user.

It is possible to use the application in a MultiMove system with up to four robots using the application.

Continues on next page

1 Discrete application summary

1.1 Summary (DAP)

Discrete application features

The Discrete Application package contains the following features:

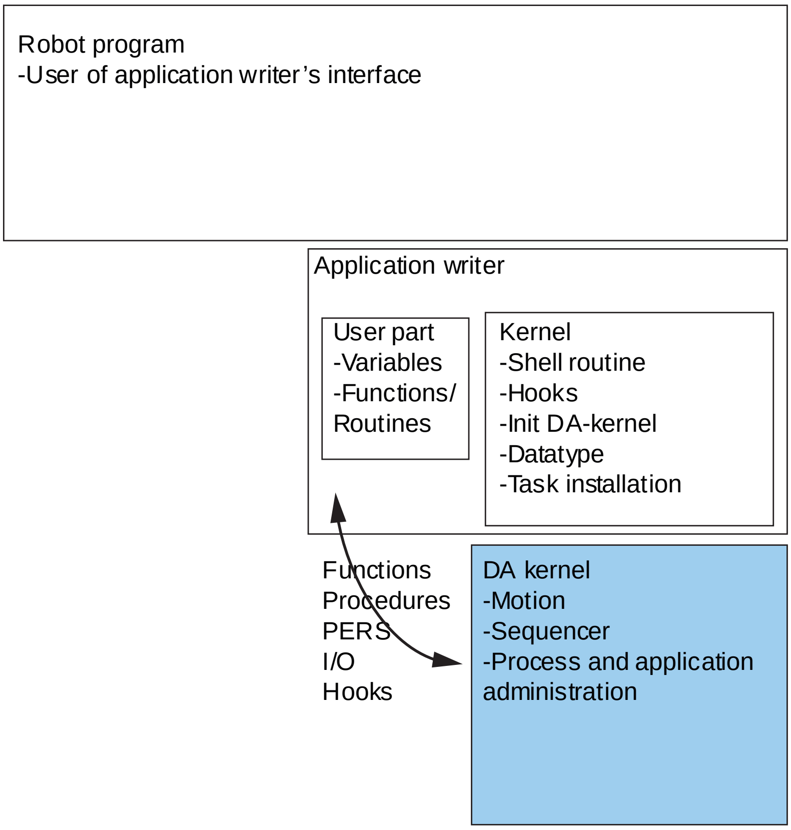
- Installation of one process instance of a Discrete Application per robot in the system
- Installation of max four processes running independently in parallel in the system
- Dynamic configuration of one RAPID task per process
- Dynamic installation of application modules
- Minimized RAPID-memory requirement
- Fast and accurate fine positioning
- Precalculation of the next position resulting in quick start after a process completion
- Free naming of I/O-signals used by the kernel.
- Setting of program number for an external device
- Setting of external start signal
- Subscribing for external ready signal
- Subscribing for external stop signal
- Dual/single tool
- Time and sequence related events calling RAPID actions hooks
- Exception event RAPID hooks such as Process Hold / Release and Abort
- Automatic process retry
- Process simulation
- External process simulation
- Return to the process position
- Process tool counters
- Supports both program and start triggered external devices
- Process current data setting and retrieving
- Manual process execution
- Possible to start external process disregarding the in position event
- Individual process abort
- Cancelling of all processes at instruction abortion

Principles of discrete applications

The scope of the Discrete Application is limited to RAPID, I/O-configuration and system configuration.

Continues on next page

Layers of a discrete application



xx1400002241

Figure 1.1: Layers and interfaces

DAP is based on a separate handling of motion and processes. The motion acts as trigger and synchronisation towards the processes. On its way towards the programmed position, the motion task will trigger actions in the process tasks.

The triggers are activated by virtual digital signals. Their names are fix and predefined. They are not multiplied by additional process installations.

Each process provides storage for three current data of anytype which are updated with the begin of the process, i.e. it's content is stable during process execution. The data have different purposes:

- process data: information altering with each instruction
- process tool data: information connected to the four equipment, i.e. equipment config data
- internal process data: information needed by the application shell.

Calls to hooks offer application writer's tools to shape the application processes. All the RAPID PERS data is used to customize the internal process sequence.

A program stop will only stop the motion task execution. The process and supervision does by default carry on their tasks until they come to a well defined process stop. A process hold may though very well be activated through the use of the shelf routines.

The application may run independently of the motion if manually triggered.

Continues on next page

Supported equipment:

- Up to four external process device monitoring with parallel interface. The device may be of two types - program schedule or start signal triggered. The process monitoring is interrupted by either process ready, timeout or external stop.
- Any type of process tool which can be controlled through RAPID-code and I/O interface is applicable.

Programming principles

Both the robot’s movement and the process control are supposed to be embedded in one shell instruction of free format and name.

The application “EG1” is specified by (see the example code that follows with the DAP option, RobotwareXX/options/dap):

- process data
- process tool data
- internal process data
- The system modules *EG1BAS.SYS*, *EG1PRC.SYS* and *EG1TOL.SYS* containing RAPID shell routine, data types, data definitions and routines.
- System parameters: the kernel I/O configuration.

Discrete application instructions

Instruction	Description
`DaActProc`	Activate a process.
`DaDeactAllProc`	Deactivate all installed process.
`DaDeactProc`	Deactivate a specific process.
`DaDefExtSig`	Define I/O-signals interfacing the external device.
`DaDefProcData`	Define three data which shall be used as current data at process start.
`DaDefProcSig`	Define I/O-signals for the process execution information.
`DaDefUserData`	Define process user data which enables the application writer to influence the framework behaviour.
`DaGetCurrData`	Retrieve the content of the current data of the types defined by `DaDefProcData`.
`DaProcML`	Initiator of motion and process. Order time event calculation. Move the TCP along a linear path and perform n processes.
`DaProcMJ`	Initiator of motion and process. Order time event calculation. Move the TCP along a non-linear path and perform n processes.
`DaSetCurrData`	Change the content of the current data of the types defined by `DaDefProcData`.
`DaSetupAppBehav`	Deactivate one or more of the five user hooks: `DaPrepPrcEG1, DaTmEvt1EG1, DaTmEvt2EG1, DaTmEvt3EG1, DaStartEG1`
`DaStartManAction`	The application runs independently of the motion, i.e. a manual trigg of the application.
`DaGetAppDescr`	Retrieve the application descriptors (one descriptor per robot).

Continues on next page

Instruction	Description
`DaGetNumOfProcs`	Retrieve the number of precesses in the system.
`DaGetNumOfRob`	Retrieve the number of robots (application descriptors) in the system.
`DaGetPrcDescr`	Retrieve the process descriptors.
`DaGetAppIndex`	Retrieve index of current application descriptor.

Discrete application functions

Instruction	Description
`DaGetFstTimeEvt`	Retrieve the first event time of all active processes in the current application descriptor.
`DaCheckMMSOpt`	Checks if any MultiMove option is installed.
`DaGetMP`	Retrieve the motion planner for current application descriptor.
`DaGetRobotName`	Retrieve the robot name for current application descriptor.
`DaGetRobotName`	Retrieve the name of the of the task that uses a specific application descriptor.

Discrete application data types

Data type	Description
`dadescapp`	Application descriptor.
`dadescprc`	Process descriptor.
`daintdata`	Type of required first element of `eg1intdata`.

Discrete application user hooks

The application name is added to the name of the hook. The following shows the hooks for the example application "EG1".

Hook	Description
`DaCalcEvtEG1`	Called before motion start.
`DaPrepPrcEG1`	Motion start.
`DaTmEvt1EG1`	First time event delta time T1 in advance of inpos.
`DaTmEvt2EG1`	Second time event delta time T2 in advance of inpos.
`DaTmEvt3EG1`	Third time event delta time T3 in advance of inpos.
`DaStartEG1`	Inpos (or immediately after `DaTmEvt3`) before setting external start signal.
`DaEndPrcEG1`	Called after receiving the ready signal.
`DaExtStopEG1`	Called after receiving the external device stop signal.
`DaTimoutEG1`	Called after timeout has passed without getting either ready or stop.
`DaHoldPrcEG1`	Called at process hold.
`DaRlsPrcEG1`	Called at process release after a hold.
`DaAbortPrcEG1`	Called at process abortion.

This page is intentionally left blank

2 Programming discrete application

2.1 Programming summary

Overview

The option Discrete Application supports creating new applications with a discrete behaviour, see *Discrete application summary on page 7*. The writer of an application will gain from the use of the framework in terms of:

- Development time
- Run time execution time
- RAPID-program memory need
- Similar look and feel between applications
- Tested kernel software
- MultiMove system adaption

Continues on next page

2.1.1 Designing a discrete application

About this section

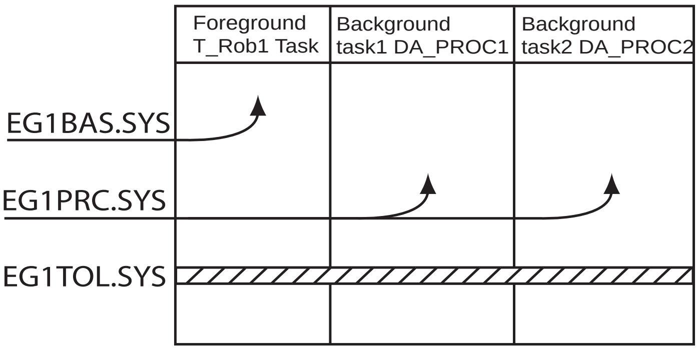
This is a description of the required steps to follow when writing a discrete application. You can find example files for designing a discrete application in the folder *options\dap* in your system.

Modules

There are three modules required for each application named “EG1”:

- Base module: *EG1BAS.SYS*
- Process module: *EG1PRC.SYS*
- Tool module: *EG1TOL.SYS*

These three modules will run in different RAPID tasks. If we, for example, have one application and two processes it will look like the following figure:



xx1400002242

Figure 2.1:

The figure above shows that module *EG1BAS.SYS* will be running in the `T_ROB1` task. Module *EG1PRC.SYS* will be running in a background (process) task. There can be as many process tasks started as the maximum number of processes allowed. Today maximum number of processes are four. There will be at least one process task attached to each robot that runs the application. In a MultiMove system it is possible to have four robots connected to the same controller, and the four processes can be distributed between the robots. If all of the robots in the system run the application, each robot can only have one process task attached to it. But if two robots run the application they can, for example, have two processes each. It is only possible to have ONE discrete application in one MultiMove system, i.e. all robots in the cell must run the same discrete application.

The figure above also shows that all installed RAPID-tasks will share code and data declared in module *EG1TOL.SYS*.

Base module

The base module shall contain code and data which is accessed in the T_ROB1 task. It shall at least contain (see *EG1BAS.SYS on page 20*):

- init code for the framework
- application shell routine

Continues on next page

- time event calculation hook
- a power on shelf routine named EG1ShPowerOn( ) where the initialization of the application and processes is sited
- further shelf routines: The framework will call shelf routines at the appropriate event given a name of the following convention:
    - EG1ShStart
    - EG1ShReStart
    - EG1ShStop
    - EG1ShQStop

Process module

The process module shall contain (see *EG1PRC.SYS on page 21*):

- the sequence hooks

Tool module

The tool module shall contain:

- common datatypes, notably process data, process tool data and internal process data
- common PERS data
- common code

See also *Installation on page 24*.

Application name

The name of the application must be defined in *eg1tol.sys* as

```
CONST string EG1_APP_NAME := "EG1";
```

The string length of the name, in this case “EG1”, is limited to 5 characters.

There must also be a routine, `DefAppName`, in *EG1bas.sys* where the application name is retrieved:

```
PROC DefAppName(INOUT string name)
  name := EG1_app_name;
ENDPROC
```

The routine `DefAppName` is called when the system is starting up, so it is very important that the routine exists in *EG1bas.sys*.

Process task

It is very important that the names of the process tasks begins with “DA_PROC” (DA_PROC1, DA_PROC2...). Look in the example code file *eg1sys.cfg*.

Initialization

The following instructions shall be used in the `EG1ShPowerOn`-routine (in *eg1bas.sys*) to initialize the application and it’s processes. Putting it in `EG1ShPowerOn` ensures the installation of the application automatically at warm start and a proper Power Failure support by the frame work.

Initialization of application and processes

`DaGetAppDescr`	returns an array containing the configured application descriptors.

Continues on next page

`DaGetPrcDescr`	returns an array containing the configured process descriptors.
`DaGetNumOfProcs`	returns how many processes that are configured in the system.
`DaGetNumOfRob`	returns how many robots in the system that run a discrete application (in a MultiMove system there can be more than one robot -> more than one application descriptor).
`EG1GetRobNo`	returns the index in the application descriptor that have the same task number as current RAPID task.

Process transfer data definition

`DaDefProcData` defines three essential data for the application. Their content will be stored by the framework as current data at each process start. The current data remains stable during the complete process.

- process data
- process tool data
- internal process data

This data has to be defined for each process. They have to be defined as `PERS` variables (see *eg1tol.sys*). The process data and process tool data shall be known to the end user. The internal process data may serve the application writer such as to make data coming from the instruction parameters accessible in the sequence hooks without showing them to the end user.

The data type shall be defined by the `RECORD` statement. It is the application writer’s choice if it shall alterable to the end user. The internal process data is the only data type with the restriction that the first element has to be of type `daintdata` and named `internal`.

```
RECORD myprocintdat
  !Required element, because it’s used by the kernel..;
  daintdata internal;
ENDRECORD
```

Current data of these three data types may be extracted or changed in the sequence hooks by `DaGetCurrData` and `DaSetCurrData`.

User variables

`DaDefUserData` defines data which enables the application writer to influence the framework behaviour. The framework will access the persistent data directly, i.e. a change of the content of such a user data is immediately recognized by the framework. This kind of data is of installation type and it is not supposed to be updated between or in the shell routine unless a `NoConc`-order was given. If a user data is not installed the framework will use it’s default value.

Example:

```
PERS num my_max_prog_no := 63;
DaDefUserData proc_desc, my_max_prog_no, DA_PROG_MAX;
```

The following table brings up all available user data. For detailed description of the palette of available user data, see *Process sequence on page 19*.

user data selector	type
DA_PROC_TIMEOUT	num

Continues on next page

user data selector	type
DA_SIMULATE_PROC	bool
DA_SIM_TIME	num
DA_AUTO_RESTART	bool
DA_PROG_MAX	num
DA_PARITY	num
DA_ASYNC_START	bool
DA_START_TYPE	num
DA_FORCED_SEQ	bool

External device connection signals

`DaDefExtSig` defines I/O-signals connected to an external device such as a weld timer. If an optional signal is omitted, the framework will not use it. For further details, see *DaDefExtSig - Discrete application - definition of the external signals on page 39*.

Process signals

`DaDefProcSig` defines I/O-signals used by the framework such as information about process status. If an optional signal is omitted, the framework will not use it. See Instructions for further details.

Designing the shell-routine

The shell routine is the end users method to run the application with the motion. The prototype-format of the shell-routine is free to be designed by the application writer. Some guidelines should however be considered.

The shell routine shall encapsulate a call of the routine `DaProcML/DaProcMJ`. The routine moves the robot to the assigned position and at the same time executes the process sequence. The movement is by default concurrent.

The module where the shell routine is declared has to be defined in the task `T_ROB1` as `NOSTEPIN`.

Required elements of the shell routine are:

- deactivation / activation of the processes (in a MultiMove system all processes should not be deactivated)
- preparation of the transfer data
- running `DaProcML`
- error clause
- backward clause

A template of the shell routine and the time event calculation hook is described on the following pages.

Continues on next page

Template of a master routine

The master shell routine should at least have the robtarget, speed data and wobjdata in the parameter list. How the parameters are gathered and if they are optional or not is decided by the application writer.

Observe that the descriptors, number of processes and so on have been fetched in the Power On routine (see *EG1BAS.SYS on page 20* and *Power On on page 25*).

```
PROC EG1ML (robtarget ToPoint \identno ID, speeddata Speed, num
     EquipNo, PERS tooldata Tool \PERS wobjdata WObj \switch InPos)

VAR bool found := FALSE;

! Check if THIS task has a running process, if any deactivate
! it. In a MultiMove system every application descriptor
! uses different motion planners, "connected" processes use
! the same motion planner). See eg1sys_mms.cfg.
FOR i FROM 1 TO EG1_NOF_ROB DO
  IF EG1_app_desc{rob_no}.MotPlan = EG1_prc_desc{i}.MotPlan
    DaDeactProc EG1_prc_desc{i};
ENDFOR

! Activate the process/processes that are connected to THIS
! motion task. See eg1sys.cfg/eg1sys_mms.cfg.
FOR j FROM 1 TO EG1_NOF_PROC DO
  IF EG1_app_desc{rob_no}.MotPlan = EG1_prc_desc{j}.MotPlan
    AND EG1_prc_desc{j}.Active = FALSE THEN
      found := TRUE;

      ! Save the equipment number for this process
      ! descrip tor
      EG1_prc_desc{j}.EquipNo := EquipNo;

      ! Activate the first inactive process belonging to
      ! current application descriptor
      DaActProc EG1_prc_desc{j};
  ENDIF
ENDFOR

IF found = FALSE THEN
  TPWrite "No process were configured for this task. Check the
       configuration.";
  Stop;
ELSE
  IF (XX_err_no = XX_NO_ERR) THEN
    ! Move to the work position and start the processes
    DaProcML ToPoint, Speed, Tool \WObj?WObj \InPos?InPos \ID?ID;
  ELSE
    DaProcML ToPoint, Speed, Tool \WObj:=WObj \InPos?InPos \ID?ID
         \PreconError;
  ENDIF
```

Continues on next page

```
ENDIF

BACKWARD
  ! Perform backward actions
  ...;
  ! Move to the weld position.
  MoveL ToPoint \ID?ID, Speed, FINE, Tool \WObj?WObj;

ERROR
  ! Perform error actions before raising the error
  ...;
  RAISE;
ENDPROC

! Before DaProcML/DaProcMJ moves the TCP it will call the
! time event calculation hook DaCalcEvtXX. Here must all the
! event times be initiated for each process.
PROC DaCalcEvtXX (num EquipNo, VAR num EventTimes{*})
! Calculate the event times or extract them from the parameters
  EventTimes{1} := ...;
  EventTimes{2} := ...;
  ...
ENDPROC
```

Process sequence

The discrete application framework encapsulates a sequence execution in connection to a fine point motion. It is typically used to monitor an external process device. It takes care of:

- Setting the program number for the process controller device including parity bit
- Starting the external device process by either a start signal or the program number.
- Waiting for a ready, timeout or external stop signal after process start
- Resetting the start signal after receiving the ready/timeout/external stop signal
- Calling application writer’s RAPID hooks.
- Logical sequence jumps (hook retry)
- Process restart after power failure
- Process canceling when moving the program pointer
- Interrupting and resuming the process at program stop/restart before the main action has started.

Each active process has it’s own independent sequence run. All sequences are started at the same time by the `DaProcML/DaProcMJ`-instruction. When all have successfully finished their tasks this is reported back to the application master of the framework which decides that the entire application has finished. The sequence is synchronized with the motion and the event times. On request (see *Sequence parameters on page 20*) the time delays may be omitted when the motion is no

Continues on next page

longer synchronizing, i.e. in case of a retry of the sequence when the end position is already reached.

Sequence parameters

The sequence may be influenced by parameters controlled from the RAPID shell, notably the user PERS data. The following list shows existing parameters, the related user data selector and the default value if not defined by the user:

Parameter function	User data selector	Description	Default
Process timeout	`DA_PROC_TIMEOUT`	Time out for waiting for the process ready signal. The time is started when the start signal is set to the external device	1 s
Process simulation	`DA_SIMULATE_PROC`	Simulation of the process. If simulation is TRUE the start signal is not set. After the simulation time (defined by `DA_SIM_TIME`) has passed on the ready signal is set	No simulation
Process simulation time	`DA_SIM_TIME`	Time to simulate the process	1 s
Automatic restart	`DA_AUTO_RESTART`	Number of times the complete process should re-run after ready signal timeout before stopping by calling the timeout hook	0, i.e. no auto restart
Maximum program number	`DA_PROG_MAX`	Maximum allowed program number. The value should match the length of the external program schedule. (The maximum value that can be used here is 8388607, e.g a 23 bit group.)	63
Program parity	`DA_PARITY`	Weld schedule parity calculation. Possible values: `DA_NONE, DA_EVEN, DA_ODD`	None
Asynchronous start	`DA_ASYNC_START`	TRUE value: The inpos event hook and the following start of the process is not waiting for inpos but immediately executes as soon as the last time event has executed	Wait for inpos
External device start type	`DA_START_TYPE`	The external device may initiate the process by setting either the start signal (=`DA_START_TRIG`) or the program number (=`DA_PROG_TRIG`)	Start signal initiator
Skipping delays	`DA_FORCED_SEQ`	The sequence delays are omitted if the motion is no longer synchronizing, notably after a retry	No forced sequence

Application writer's hooks

The application writer's hooks are the code entries where the application specific code is defined. The name has to follow the below description where again “EG1” is the application name (see *Application name on page 15*).

EG1BAS.SYS

The following hook shall be defined in *eg1bas.sys*.

`DaCalcEvtEG1`	`(num EquipNo, VAR num EventTimes{*})`

Continues on next page

`EquipNo`	Equipment number, which is an extra information to make it easier to find data if stored in arrays
`EventTimes`	Time is an array where the time events 1 through 3 shall be returned from the calculation. The order has to be: Time{1} >= Time{2} >= Time{3} else this order will be forced by the framework.

EG1PRC.SYS

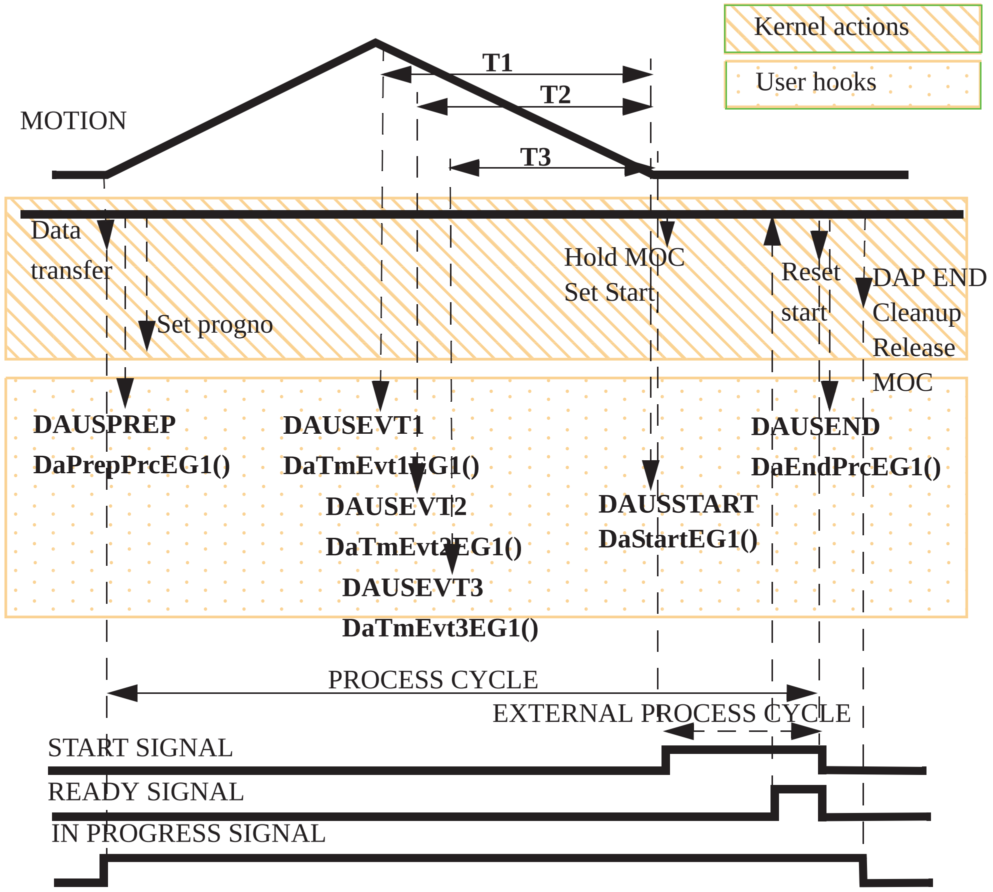
The following hooks shall be defined in *eg1prc.sys*. Each sequence hook is called once for each process. The routine parameter format is the same for all procedures:

`ProcNo`	Process number, which is used to get the correct process descriptor in the process descriptor array.
`Status`	Contains the execution result and information about where to resume the sequence. For possible values see *Sequence control on page 22*.
`Par1` and `Par2`	Dummy parameters currently not used.

They are called in the following moments of the sequence:

`DaPrepPrcEG1`	`(PERS num Status, num ProcNo, bool Par1, string Par2)` Called at the start of the motion
`DaTmEvt1EG1`	`(PERS num Status, num ProcNo, bool Par1, string Par2)` Called at the first time event of the motion
`DaTmEvt2EG1`	`(PERS num Status, num ProcNo, bool Par1, string Par2)` Called at the second time event of the motion
`DaTmEvt3EG1`	`(PERS num Status, num ProcNo, bool Par1, string Par2)` Called at the third time event of the motion
`DaStartEG1`	`(PERS num Status, num ProcNo, bool Par1, string Par2)` Called before the start signal is set by the kernel. This event is either executed at inposition (default) or immediately after the third time event.
`DaEndPrcEG1`	`(PERS num Status, num ProcNo, bool Par1, string Par2)` Called when receiving the process ready signal. This indicates a successful end of the process and should be the last process event hook.
`DaExtStopEG1`	`(PERS num Status, num ProcNo, bool Par1, string Par2)` Called when receiving the process external stop signal
`DaTimoutEG1`	`(PERS num Status, num ProcNo, bool Par1, string Par2)` Called when process timeout has passed without receiving neither the ready signal not the stop signal.
`DaHoldPrcEG1`	`(PERS num Status, num ProcNo, bool Par1, string Par2)` Called when process hold signal is set. Trigger on positive flange
`DaRlsPrcEG1`	`(PERS num Status, num ProcNo, bool Par1, string Par2)` Called when process hold signal is reset after a hold. Trigger on negative flange
`DaAbortPrcEG1`	`(PERS num Status, num ProcNo, bool Par1, string Par2)` Called when process abort signal is set. Trigger on positive flange

Continues on next page



xx1400002243

Figure 2.2: Example: Successful application sequence

Sequence control

The framework allows the user hooks to influence where to resume the sequence through the status parameter. The following values are possible:

- `DAOK`
- `DACANCEL`
- `DAUSPREP/DAUSEVT1/DAUSEVT2/DAUSEVT3/DAUSSTART/DAUSEND` offers the possibility to redo part of the sequence by entering the assigned hook. Only backwards jumps are allowed, otherwise the return value is treated as `DAOK`.

Sequence influence

The sequence may be influenced by the instruction `DaSetupAppBehav`. The instruction can affect five of the eleven sequence hooks - `DaPrepPrcEG1, DaTmEvt1EG1, DaTmEvt2EG1, DaTmEvt3EG1` and `DaStartEG1`. With help of the instruction `DaSetupAppBehav` these five sequence hooks can be deactivated, and thereby time will be saved. The instruction must be called before calling the routine `DaProcML/DaProcMJ`. For further details, see *DaSetupAppBehav - Discrete application - sets up application behaviour on page 57*.

Continues on next page

Exceptions

Process abortion

Each process may be aborted individually. The process is then reported back to the application master as finished. A process abortion kills any ongoing RAPID-execution even if for instance waiting for a user interaction in a `TPReadFK`. `DaAbortPrcEG1` is called as last user hook.

- Initiator for a process abortion may be:
- Process abort signal
- User hook returned `DACANCEL`
- Application abortion

Application abortion

The entire process may be aborted. That may be the case when the user-PP is moved, i.e. the shell routine is abandoned. It will cause a process abortion for each active process. See above.

Initiator of an application abortion is:

- Application shell routine was given up by moving the PP

Process hold

A process hold interrupts a running hook and calls `DaHoldPrcEG1`. If a hold occurs while the start signal is on the start signal is reset.

Initiator of a process hold is:

- Program execution stop before the start of the main action.
- Process hold signal goes high. This may be done in a stop/qstop-shelf if desired.

Process release

A process release is always run after a process hold if the process was not aborted during the hold. `DaRlsPrcEG1` is called and the interrupted event hook is resumed. If the hold occurred while the start signal was high the sequence is resumed where the start signal was set and timeout, stop and ready is subscribed for.

Initiator of a process release is:

- Program execution is restarted.
- Process hold signal goes low which may be done in a restart-shelf if desired.

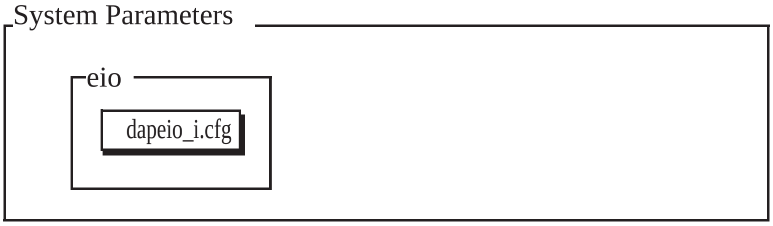
Utilities

`DaGetAppDescr`	Returns the descriptor of an installed application.	
`DaGetProcDescr`	Returns the descriptor of an installed process.	
`DaGetCurrData`	Retrieves currently valid data from the framework.	The data is valid from the moment `DaProcML/DaProcMJ` was called and the motion has started i.e. when the earlier process has finished and released the motion.
`DaSetCurrData`	changes the currently used data.	The same time span as described for `DaGetCurrData.`

2.1.2 Installation

I/O configuration

The I/O-configuration contains required internal virtual signals which are only known and used by the discrete application framework.



xx1400002244

Figure 2.3: The parameter configuration

RAPID system configuration

The installation of the discrete application is done when the system is starting up.

Task installation

After a cold start in a single system there will be two tasks installed. One motion task, `T_ROB1`, that will run the application, and one background task, `DA_PROC1`, that will run one process. Observe that if only the DAP option is included in the system (and no Spot option), the option MultiTasking also must be included. Then it is possible to add process task via RobotStudio.

If there is a MultiMove system with for example four motion robots in the system, the motion tasks will be named `T_ROB1... T_ROB4`, but there will still only be two process tasks, `DA_PROC1` and `DA_PROC2`, installed from start (if you use the example file *eg1sys.cfg*). If more robots will run the discrete application, process tasks must be added via RobotStudio. The option MultiTasking is not needed because it is included in the MultiMove option.

Task addition

In RobotStudio it is possible to look at and configure the tasks. Under the tab **Configuration**/**Controller**/**Mechanical Unit Groups** (only if you have a MultiMove system) you can see how the configuration is done. It is also possible to change the configuration. New background tasks (not motion tasks) will be added in **Configuration**/**Controller**/**Tasks.** If the configuration file (*sys.cfg*) is saved an example how part of it will look like will be like this:

```
CAB_TASKS:
-Name "T_ROB1" -Type "NORMAL" -UseMechanicalUnitGroup "rob1"
     -MotionTask
-Name "T_ROB2" -Type "NORMAL" -UseMechanicalUnitGroup "rob2"
     -MotionTask
-Name "DA_PROC1" -TrustLevel "SysHalt" -UseMechanicalUnitGroup
     "rob1"
-Name "DA_PROC2" -TrustLevel "SysHalt" -UseMechanicalUnitGroup
     "rob1"
```

Continues on next page

```
-Name "DA_PROC3" -TrustLevel "SysHalt" -UseMechanicalUnitGroup
     "rob2"

MECHANICAL_UNIT_GROUP:
-Name "rob1" -Robot "ROB_1" -UseMotionPlanner "motion_planner_1"
-Name "rob2" -Robot "ROB_2" -UseMotionPlanner "motion_planner_2"
```

The example above shows two motion task “connected” to process tasks via the mechanical unit group. Motion task `T_ROB1` will use two processes and task `T_ROB2` will use one process. Look also in the example code for DAP, *eg1sys.cfg/eg1sys_mms.cfg*.

Power On

The instruction, `DaShelfPowerOn`, is called by every task that will run the discrete application, when the system is starting up. It is not possible to look into the code because it is cryptated, but what happens is that the application and processes are set up. The first motion task that calls `DaShelfPowerOn` does the initiation. A check is done how many motion task in the system that will work as discrete application robots, and how many processes every application robot will use. In a single system there is only one motion task, but in a MultiMove system there can be up to four robots that can act as application robots. A process is “connected” to a motion task through the `MECHANICAL_UNIT_GROUP`. In a single system all tasks use the same mechanical unit group, but in a MultiMove system every motion task uses different mechanical unit groups. It is through the “connection” motion task/process task the system can discern which motion task will act as a discrete application task. The process task MUST be named like `DA_PROC1`, `DA_PROC2`... because that is how the system recognize the processes.

A maximum number of four discrete application descriptors may be installed for the hole system, i.e. there can be four robots that run a discrete application. It is only possible to have one discrete application configured in the system. It can be up to four processes installed, divided between the robots.

Template of a power on routine

The routine is called by all application tasks when the system is starting up. A check is done which of the application descriptors that corresponds to this task. The application descriptors are saved in an array and the index of the descriptor is saved in a persistent variable and is later on used in other routines, among others, `EG1ML`.

```
PROC EG1ShPowerOn()
  ! Init EG1 PERS
  ! Get process descriptors
  DaGetPrcDescr EG1_prc_desc;

  ! Get application descriptor
  DaGetAppDescr EG1_app_desc;

  ! Get number of processes
```

Continues on next page

```
  DaGetNumOfProcs EG1_NOF_PROC;

  ! Get number of robots (In a MultiMove system there can
  ! be more than one robot -> more than one application
  ! descriptor)
  DaGetAppIndex rob_no;
  ! Define the process data
  FOR j FROM 1 TO EG1_NOF_PROC DO
    DaDefProcData EG1_prc_desc{j}, EG1_prc_data{j},
    EG1_tool_data{j}, EG1_int_data{j};

    ! Define the user data
    DaDefUserData EG1_prc_desc{j}, EG1_prc_time_out,
    DA_PROC_TIMEOUT;

    TEST j
    CASE 1:
      ! Define the external signals
      DaDefExtSig EG1_prc_desc{1}, doStart1, diReady1, goProgNo1;
      ! Define the process signals
      DaDefProcSig EG1_prc_desc{1}, doInProgress1, doProcFault1,
           doExtFault1;
    CASE 2:
      ....
    ENDTEST
  ENDFOR
ENDPROC
```

Module

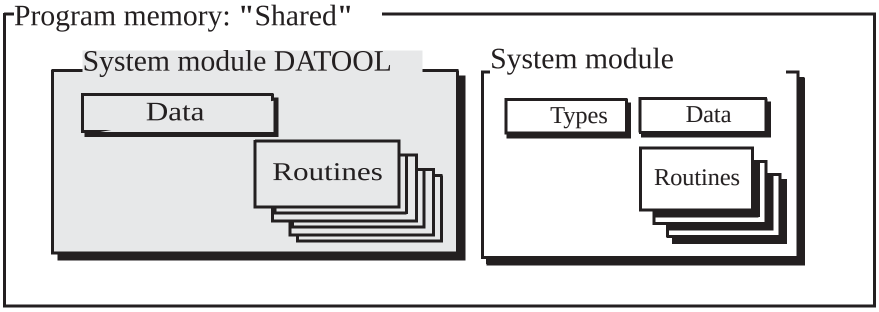
The framework will allocate encoded modules with predefined names in the tasks. It will also allocate the application specific modules provided by the application writer. Those three modules must follow the rules below:

- The three system modules (a base, process and tool module) must be loaded into the directory *HOME:/dap*. Then make a warmstart.
- Name convention: *EG1BAS.SYS*, *EG1PRC.SYS* and *EG1TOL.SYS* where "EG1" is the name of the application used in `DaDefAppName` (see *Application name on page 15*).

Continues on next page

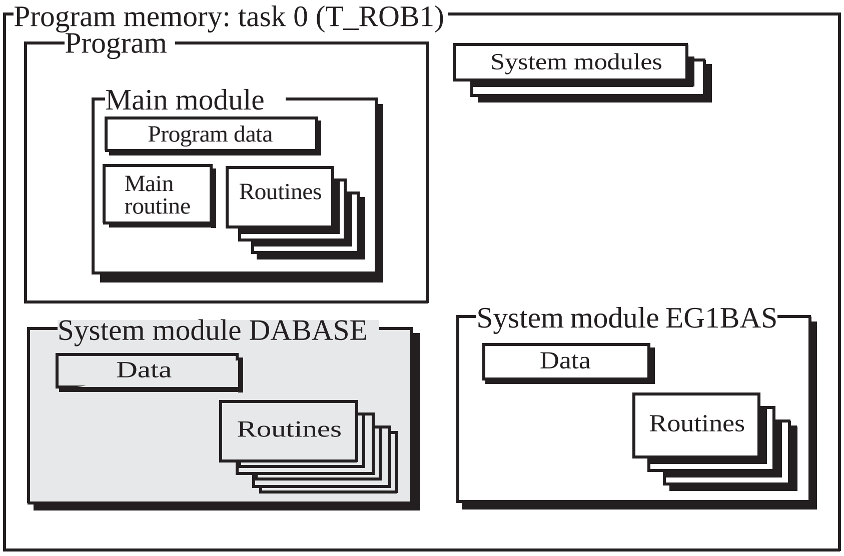
RAPID task and module setup example

The following description is in accordance to the example with the application "EG1" in the initialization chapter. It shows one task that runs the application, `T_ROB1`, and three processes connected to it, `DA_PROC1`, `DA_PROC2` and `DA_PROC3`.



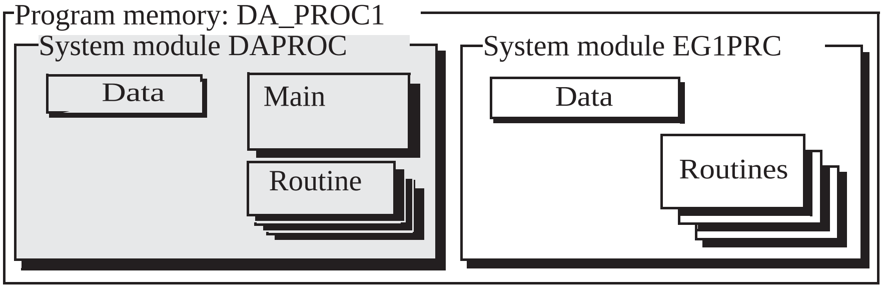
xx1400002245

Figure 2.4: Module Allocation for Discrete application



xx1400002246

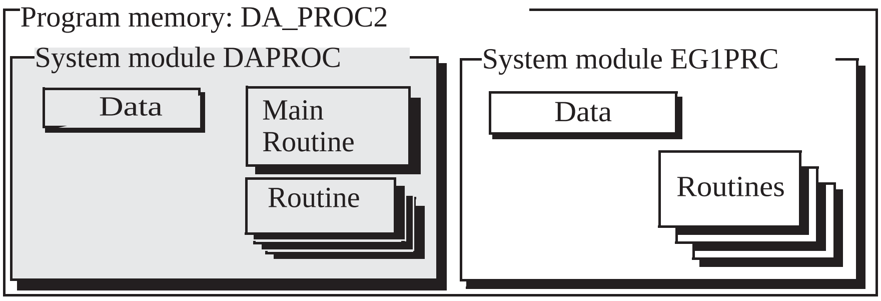
Figure 2.5:



xx1400002247

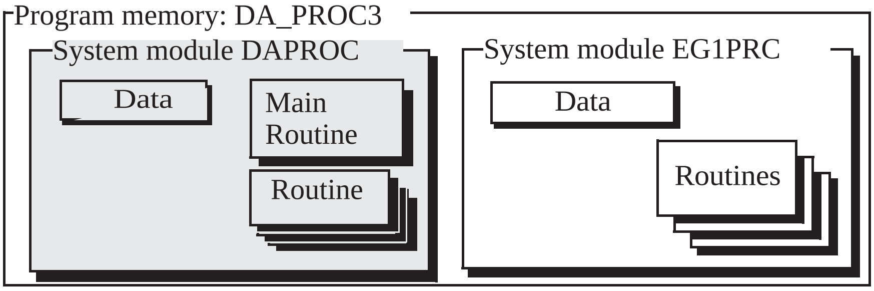
Figure 2.6:

Continues on next page



xx1400002248

Figure 2.7:



xx1400002249

Figure 2.8:

With the DAP option it follows a executable application and a framework of the three system modules. There are six files connected to the executable application, namely:

- *EG1.PRG*
- *EG1BAS.SYS*
- *EG1PRC.SYS*
- *EG1TOL.SYS*
- *EG1_EIO.CFG*
- *READ_EG1.TXT*

Before running this application read the file *READ_EG1.TXT*. The name of the three system modules is as follows:

- *EG1BAS.SYS*
- *EG1PRC.SYS*
- *EG1TOL.SYS*

3 RAPID Reference

3.1 RAPID Data types

3.1.1 dadescapp - Discrete application - application descriptor

Description

dadescapp (Discrete Application - Application descriptor) is used to describe an application within the discrete application.

Overview

Data of the type dadescapp contains a reference to an installed application within the discrete application. It is linked during the power on sequence of the system, where the instruction DaShelfPowerOn is called. Every motion task that is configured (i.e. has a process “connected”) to run a discrete application will create an instance of an application descriptor.

In a MultiMove system it is possible to have a maximum of four instances of an application descriptor, i.e. only four robots can run run a discrete application.

Example

```
! The new application name. The string length of the name
! is limited to 5 characters.
CONST string EG1_APP_NAME := "EG1";
PERS string DaAppName := "";
! Number of possible robots running an application. In a MultiMove
! system there will be possible to have four intances of an
! application, in a single system one.
CONST num EG1_MAX_NOF_ROB := 4;
! Application descriptor
PERS dadescapp EG1_app_desc{EG1_MAX_NOF_ROB} := [[0, 0, 0, 0, 0,
     0, 0, ""], ...];
...
! Get application descriptor
DaGetAppDescr EG1_app_desc;
```

This data can then be used as shown in the example below.

```
IF EG1_app_desc{1}.taskno = 1 THEN
  ...;
ENDIF
```

A new application EG1 will be installed and the descriptors of this new application will be the allocated data EG1_app_desc.

The declarations above must exist in the file *eg1tol.sys*. And it is very important that the instruction DefAppName exist in *EG1bas.sys*, so the system will know the name of the application.

Continues on next page

The application name is declared by the variable `EG1_APP_NAME` and is retrieved during the start up sequence, by the routine `DefAppName`. A new application EG1 will be installed and instances of the descriptor of this new application will be the allocated data `EG1_app_desc`. If it is a MultiMove system, an instance per motion task that runs the discrete application will be installed.

When the system is starting up the application descriptors are installed and can be “picked up” with the instruction `DaGetAppDesr`.

Components

`ipm`

ipm number

Data type: `num`

Internal use

`id`

identification

Data type: `num`

Internal use

`taskno`

task number

Data type: `num`

The task running this instance of application

`motplan`

motion planner

Data type: `num`

The motion planner this instance of appliction is using

`noofprocs`

number of processes

Data type: `num`

Number of processes this instance of application has “connected”

`dadamno`

damaster number

Data type: `num`

Internal use

`robotname`

robot name

Data type: `string`

Name of the robot that runs this instance of the application

`taskname`

task name

Continues on next page

Data type: `string`

Name of the task that runs this instance of the application

Related information

For information about	See
Process descriptor	*dadescprc - Discrete application - process descriptor on page 32*
Characteristics of non-value data types	*Technical reference manual - RAPID overview* *Discrete application summary on page 7*

3.1.2 dadescprc - Discrete application - process descriptor

Description

dadescprc (Discrete Application - Process descriptor) is used to describe an process within

the discrete application.

Overview

Data of the type dadescprc contains a reference to an installed process in an already installed application within the discrete application.

It is linked to a new process during the power on sequence of the system. For every process task (DA_PROCX) that is configured in the system, there will be a new process.

In a MultiMove system, it is possible to have a maximum of four instances of process descriptors, i.e. only four equipments can be active in the system at the same time (every equipment “uses” one process descriptor).

Example

```
! Possible number of processes in the system.
CONST num NOF_POSS_PROCS := 4;

! Allocate descriptors for the new processes
PERS dadescprc proc_desc{NOF_POSS_PROCS} := [[0, 0, 0, 0, 0, 0, 0,
     0, FALSE], ...];
...
! Get process descriptors
DaGetPrcDescr proc_desc;
```

This data can then be used as shown in the example below.

```
IF proc_desc{1}.taskno = 1 THEN
  ...;
ENDIF
```

When the system is starting up, the processes are installed. The process descriptors can be “picked up” with the instruction DaGetPrcDescr and will be the allocated data proc_desc.

Components

ipm

ipm number

Data type: num

Internal use

id

identification

Data type: num

Continues on next page

Internal use

`taskno`

task number

Data type: `num`

Number of the task that uses this process descriptor.

`motplan`

motion planner

Data type: `num`

Number of the motion planner that uses this process descriptor.

`procno`

process number

Data type: `num`

Number of processes “connected” to currrent application descriptor. Up to four processes can be used in a system, divided between the application descriptors.

`equipno`

equipment number

Data type: `num`

Number of the equipment

`daprocno`

process number

Data type: `num`

Number of process, i.e if the process name is “DA_PROC1”, then `daprocno` = 1

`active`

active

Data type: `bool`

Tells if the process is active or not

Related information

For information about	See
Application descriptor	*dadescapp - Discrete application - application descriptor on page 29*
Characteristics of non-value data types	*Technical reference manual - RAPID overview* *Discrete application summary on page 7*

3.1.3 daintdata - Discrete application - internal data

Description

`daintdata` (Discrete Application - Internal data) is used to define internal data within the discrete application.

Overview

Discrete application - Internal data is a data type used for internal data transfer between the developer of the application and the discrete application framework. The data is setup before process start and it shall be used in the user hooks to gain information from the current process.

Components

`prog_no`

Program Number

Data type: `num`

The program number for the external device.

`noconc`

No Concurreny

Data type: `bool`

No concurrency information for the process execution. If this flag is set to TRUE the process will be executed in no concurrency mode.

`equip_act`

Equipment Active

Data type: `bool`

Process belong to the assigned equipment is active if this flag is set to TRUE.

`start_no`

Start Number

Data type: `num`

The subprocess (e.g. dual tool) number information to the external device.

1: Start1 Ready1 -> Subprocess1

2: Start2 Ready2 -> Subprocess2

12: Start1 Ready1 Start2 Ready2 -> Subprocess1 first, Subprocess2 second

21: Start2 Ready2 Start1 Ready1 -> Subprocess2 first, Subprocess1 second

`act_start_no`

Active Start Number

Data type: `num`

The active start number information (see `start_no`), the value is set by the discrete application framework and shall not be changed.

Continues on next page

`counter1`

Data type: `num`

The counter of the execution for the subprocess 1.

`counter2`

Data type: `num`

The counter of the execution for the subprocess 2.

`prog_name`

Program Name

Data type: `string`

The program name for the external device. This component is not yet implemented. When `daintdata` is initiated then give this component the value of an empty string.

Example

```
! Definition of the intdata
RECORD swintdata
  daintdata internal;
  num component2;
  ...;
ENDRECORD

PERS swintdata internal_data1 := [ [1, FALSE, TRUE, 1, 1, 0, 0,
     ""], 1, ... ];
...
! Setup the internal data
internal_data1.internal.prog_no := 1;
internal_data1.internal.noconc := FALSE;
internal_data1.internal.euip_act := TRUE;
internal_data1.internal.start_no := 1;
internal_data1.internal.act_start_no := 1;
internal_data1.internal.counter1 := 0;
internal_data1.internal.counter2 := 0;
internal_data1.internal.prog_name := "";
...
```

Structure

```
<dataobject of daintdata>
  <prog_no of num>
  <noconc of bool>
  <equip_act of bool>
  <start_no of num>
  <act_start_no of num>
  <counter1 of num>
  <counter2 of num>
  <prog_name of string>
```

3.2 RAPID Instructions

3.2.1 DaActProc - Discrete application - activate process

Description

DaActProc is used to activate a connected process in the application within the discrete application framework.

Examples

```
! Possible number of processes in the system
CONST num NOF_POSS_PROCS := 4;

! Allocate descriptors for the new processes
PERS dadescprc proc_desc{NOF_POSS_PROCS};
...
! Get process descriptors
DaGetPrcDescr proc_desc;
! Activate process
DaActProc proc_desc{1};
```

The first process will be activated after the DaActProc... execution.

Arguments

```
DaActProc ProcDesc
```

ProcDesc

Process Descriptor

Data type: dadescprc

The descriptor of the connected process to be activated.

Limitations

The number of active processes at the same time is limited to 4.

If no application is active, the program execution will result in a fatal RAPID user error.

Syntax

```
DaActProc [ ProcDesc':=' ] < persistent array {*} (PERS) of
     dadescprc > ';'
```

Related information

For information about	See
Application descriptor	*dadescapp - Discrete application - application descriptor on page 29*
Process descriptor	*dadescprc - Discrete application - process descriptor on page 32*

3.2.2 DaDeactAllProc - Discrete application - deactivate all processes

Description

`DaDeactAllProc` is used to deactivate all active processes in the application within the discrete application framework.

Examples

```
! Possible number of processes in the system.
CONST num NOF_POSS_PROCS := 4;

! Allocate descriptors for the new processes
PERS dadescprc proc_desc{NOF_POSS_PROCS};
...
! Get process descriptors
DaGetPrcDescr proc_desc;
! Deactivate all processes
DaDeactAllProc;
```

All active processes will be deactivated after the `DaDeactAllProc` ... execution.

Limitations

When trying to deactivate all processes, be sure that a minimum of one process is already active. Otherwise the program execution will result in a fatal RAPID user error.

Syntax

```
DaDeactAllProc ';'
```

Related information

For information about	See
Application descriptor	*dadescapp - Discrete application - application descriptor on page 29*
Process descriptor	*dadescprc - Discrete application - process descriptor on page 32*

3.2.3 DaDeactProc - Discrete application - deactivate process

Description

DaDeactProc is used to deactivate a connected process in the application within the discrete application framework.

Examples

```
! Possible number of processes in the system
CONST num NOF_POSS_PROCS := 4;

! Allocate descriptors for the new processes
PERS dadescprc proc_desc{NOF_POSS_PROCS};
...
! Get process descriptors
DaGetPrcDescr proc_desc;
! Activate process
DaDeactProc proc_desc{1};
```

The first process will be deactivated after the DaDeactProc... execution.

Arguments

```
DaDeactProc ProcDesc
```

ProcDesc

Process Descriptor

Data type: dadescprc

The descriptor of the connected process to be deactivated.

Limitations

If no application is active, the program execution will result in a fatal RAPID user error.

Syntax

```
DaDeactProc [ ProcDesc':=' ] < persistent array {*} (PERS) of
     dadescprc > ';'
```

Related information

For information about	See
Application descriptor	*dadescapp - Discrete application - application descriptor on page 29*
Process descriptor	*dadescprc - Discrete application - process descriptor on page 32*

3.2.4 DaDefExtSig - Discrete application - definition of the external signals

Description

DaDefExtSig is used to define the external signals of the connected process within the discrete application.

Examples

```
! Possible number of processes in the system
CONST num NOF_POSS_PROCS := 4;

! Allocate the desriptor for the new processes
VAR dadescprc proc_desc{NOF_POSS_PROCS};

! The event times of the processes
VAR num evt_time_prc1{3} := [2.5, 1.8, 1.0];
VAR num evt_time_prc2{3} := [2.2, 1.7, 0.8]

! The first time event
VAR num first_time_event;
...
! Get process descriptors
DaGetPrcDescr proc_desc;

! Define the external signals for process one
DaDefExtSig proc_desc{1}, doStart1, diReady1, goProgNo1
```

The external signals will be defined as specified after DaDefExtSig ... execution.

Note

Those signals must be already configurated in the system.

Arguments

```
DaDefExtSig ProcDesc Start1 [\Start2] Ready1 [\Ready2] [\Reset]
     [\Stop] ProgNo [\ProgParity]
```

ProcDesc

Process Descriptor

Data type: dadescprc

The descriptor of the connected process.

Start1

Data type: signaldo

The start signal one of the connected process. This signal is used to start the process of the external device. Start1 is set if the value of start_no and act_start_no in daintdata is 1.

[\Start2]

Data type: signaldo

Continues on next page

The start signal two of the connected process (optional). If this signal is defined, the optional argument `Ready2` must also be in use. The signal is used if `start_no` or `act_start_no` in `daintdata` is 2. If this optional signal is not defined in the instruction `Start1` will be used.

`Ready1`

Data type: `signaldi`

The ready signal one of the connected process. This signal is used to subscribe for the end of the external process. `Ready1` is subscribed for if `start_no` or `act_start_no` in `daintdata` is 1. When the signal is received the main action ready hook is executed.

`[\Ready2]`

Data type: `signaldi`

The ready signal two of the connected process (optional). If this signal is defined, the optional argument `Start2` must also be in use. The signal is used if `start_no` or `act_start_no` in `daintdata` is 2. If this optional signal is not defined in the instruction `Ready1` will be used.

`[\Reset]`

Data type: `signaldo`

The reset signal of the connected process. The output is pulsed (10ms) after the execution of the main action timeout or stop hook. If the signal is not defined, it will not be used.

`[\Stop]`

Data type: `signaldi`

The stop signal of the connected process. This signal is used to subscribe for a stop signal from the external device. When the signal is received, the main action stop hook is executed. If the signal is not defined, it will not be used.

`ProgNo`

Program Number

Data type: `signalgo`

The program number signals of the connected process.

`[\ProgParity]`

Program Parity

Data type: `signaldo`

The program parity of the program number. The different parities are:

- None parity if this optional argument is omitted.
- Odd parity if this optional argument is in use and the output signal is 0.
- Even parity if this optional argument is in use and the output signal is 1.

Limitations

Make sure that the signals are configured. Otherwise the program execution will result in a fatal RAPID user error.

Continues on next page

Syntax

```
DaDefExtSig
  [ ProcDesc':=' ] < persistent array {*} (PERS) of dadescprc >
       ','
  [ Start1':=' ] < variable (VAR) of signaldo >
  [ '\' Start2 ':=' < variable (VAR) of signaldo > ] ','
  [ Ready1':=' ] < variable (VAR) of signaldi >
  [ '\' Ready2 ':=' < variable (VAR) of signaldi > ]
  [ '\' Reset':=' < variable (VAR) of signaldo > ]
  [ '\' Stop ':=' < variable (VAR) of signaldi > ] ','
  [ ProgNo':=' ] < variable (VAR) of signalgo >
  [ '\' ProgParity ':=' < variable (VAR) of signaldo > ]';'
```

Related information

For information about	See
Application descriptor	*dadescapp - Discrete application - application descriptor on page 29*
Process descriptor	*dadescprc - Discrete application - process descriptor on page 32*

3.2.5 DaDefProcData - Discrete application - definition of the process data

Description

`DaDefProcData` is used to define the data of a connected process within the discrete application.

Examples

Sequence for define data for one process:

```
! Possible number of processes in the system
CONST num NOF_POSS_PROCS := 4;
! Allocate descriptors for the new processes
PERS dadescprc proc_desc{NOF_POSS_PROCS};

! Definition of the procdata
RECORD procdata
  string string_comp;
ENDRECORD
! Definition of the tooldata
RECORD tooldata
  string string_comp;
  num time_event1;
  num time_event2;
  num time_event3;
ENDRECORD
! Definnition of the intdata
RECORD intdata
  daintdata internal;
  string string_comp;
ENDRECORD

! Allocate a procdata, a tooldata and a intdata
PERS procdata prc_data{NOF_POSS_PROCS} := [["PROCDATA1], ...];
PERS tooldata tool_data{NOF_POSS_PROCS} := [["TOOLDATA1", 0.20,
     0.1, 0.05], ...];
PERS intdata int_data{NOF_POSS_PROCS} := [[[5, TRUE, TRUE, 1, 1,
     0, 0,""], "INTDATA1"], ...];
...
! Get process descriptors
DaGetPrcDescr proc_desc;
! Define the process data
DaDefProcData proc_desc{1}, prc_data{1}, tool_data{1}, int_data{1};
```

The process data will be defined as specified after `DaDefProcData` ... execution.

Note

Those data must be predefined as persistents in a defined module.

Continues on next page

Arguments

```
DaDefProcData ProcDesc ProcData ToolData IntProcData
```

`ProcDesc`

Process Descriptor

Data type: `dadescprc`

The descriptor of the connected process.

`ProcData`

Process Data

Data type: `anytype`

The process data of the connected process.

`ToolData`

Tool Data

Data type: `anytype`

The tool data of the connected process.

`IntProcData`

Internal Process Data

Data type: `anytype`

The internal process data of the connected process.

Limitations

When defining process data, the process connected to the current application must be already installed. Otherwise the program execution will result in a fatal RAPID user error.

If the specified data are not PERS, the program execution will result in a fatal RAPID user error.

Syntax

```
DaDefProcData
  [ ProcDesc’:=’ ] < persistent array {*} (PERS) of dadescprc >
       ’,’
  [ ProcData’:=’ ] < persistent (PERS) of anytype > ‘,’
  [ ToolData’:=’ ] < persistent (PERS) of anytype > ‘,’
  [ IntProcData’:=’ ] < persistent (PERS) of anytype > ’;’
```

Related information

For information about	See
Application descriptor	*dadescapp - Discrete application - application descriptor on page 29*
Process descriptor	*dadescprc - Discrete application - process descriptor on page 32*
Internal data	*daintdata - Discrete application - internal data on page 34*

3.2.6 DaDefProcSig - Discrete application - definition of the process signals

Description

`DaDefProcSig` is used to define the process signals of the connected process within the discrete application.

Examples

```
! Possible number of processes in the system
CONST num NOF_PROCS := 4;
! Allocate descriptors for the new processes
PERS dadescprc procdesc{NOF_PROCS};
...
! Get process descriptors
DaGetPrcDescr proc_desc;
! Define the process signals for process one
DaDefProcSig proc_desc{1}, doInProgress1, doProcFault1, doExtFault1;
```

The process signals will be defined as specified after `DaDefProcSig` ... execution.

Note

Those signals must be already configurated in the system.

Arguments

```
DaDefProcSig ProcDesc InProgress ProcFault ExtFault [\Cancel]
     [\Hold]
```

`ProcDesc`

Process Descriptor

Data type: `dadescprc`

The descriptor of the connected process.

`InProgress`

In Progress

Data type: `signaldo`

The in progress signal of the connected process. This signal is set when the process is running.

`ProcFault`

Process Fault

Data type: `signaldo`

The process fault signal of the connected process. This signal is set when a process fault occured.

`ExtFault`

External Fault

Data type: `signaldo`

Continues on next page

The external fault signal of the connected process. This signal is set when an external fault occured.

`[\Cancel]`

Data type: `signaldi`

The cancel signal of the connected process. If this argument is specified and the input is set to 1, the process will be aborted an reset.

`[\Hold]`

Data type: `signaldi`

The hold signal of the connected process. If this argument is specified and set to 1, the process will be hold untill the signal is set to 0 again.

Limitations

Make sure that the signals are configured. Otherwise the program execution will result in a fatal RAPID user error.

Syntax

```
DaDefProcSig
  [ ProcDesc':=' ] < persistent array {*} (PERS) of dadescprc >
       ','
  [ InProgress':=' ] < variable (VAR) of signaldo > ','
  [ ProcFault':=' ] < variable (VAR) of signaldo > ','
  [ ExtFault':=' ] < variable (VAR) of signaldo >
  [ '\' Cancel ':=' < variable (VAR) of signaldi > ]
  [ '\' Hold ':=' < variable (VAR) of signaldi > ] ';'
```

Related information

For information about	See
Application descriptor	*dadescapp - Discrete application - application descriptor on page 29*
Process descriptor	*dadescprc - Discrete application - process descriptor on page 32*

3.2.7 DaDefUserData - Discrete application - define user data

Description

DaDefUserData is used to define process user data within the discrete application.The instruction transmits the location of the data which gives the framework the possibility to access the same data location as the RAPID-program, i.e. changing the content of such a PERS data is immediately affecting the framework.

Examples

```
! Possible number of processes in the system
CONST num NOF_POSS_PROCS := 4;

! Allocate descriptors for the new processes
PERS dadescprc proc_desc{NOF_POSS_PROCS};

! Process ready timeout
PERS num timeout := 2;
...
! Get process descriptors
DaGetPrcDescr proc_desc;
! Define timeout user data
DaDefUserData proc_desc{j}, timeout, DA_PROC_TIMEOUT;
```

The specified user data will be defined as specified for the selected process after DaDefUserData ... execution. Note that all processes may very well share the same PERS data of a certain user data type if it shall be valid for the entire application.

Arguments

```
DaDefUserData ProcDesc UserData Selector
```

ProcDesc

Process Descriptor

Data type: dadescprc

The descriptor of the connected process.

UserData

User Process Data

Data type: anytype

User process data of any type. The type however has to match the intended user data. See table below.

user data selector	type
DA_PROC_TIMEOUT	num
DA_SIMULATE_PROC	bool
DA_SIM_TIME	num

Continues on next page

user data selector	type
DA_AUTO_RESTART	bool
DA_PROG_MAX	num
DA_PARITY	num
DA_ASYNC_START	bool
DA_START_TYPE	num
DA_FORCED_SEQ	bool

`Selector`

User Process Data Selector

Data type: `num`

Selector that describes the type of user data.

Note

For further details, see *Programming discrete application on page 13*.

Syntax

```
DaDefUserData
  [ ProcDesc':=' ] < persistent array {*} (PERS) of dadescprc >
       ','
  [ UserData':=' ] < persistent (PERS) of anytype > ','
  [ Selector':=' ] < expression (IN) of num> ';'
```

Related information

For information about	See
Application descriptor	*dadescapp - Discrete application - application descriptor on page 29*
Process descriptor	*dadescprc - Discrete application - process descriptor on page 32*

3.2.8 DaGetCurrData - Discrete application - get current data

Description

`DaGetCurrData` is used to get a selected data of the connected process within the discrete application.

Examples

Sequence for define data for one process:

```
Sequence for define data for one process:
! Possible number of processes in the system
CONST num NOF_POSS_PROCS := 4;
! Allocate descriptors for the new processes
PERS dadescprc proc_desc{NOF_POSS_PROCS};
! User defined data types for the process

RECORD procdata
  string string_comp;
ENDRECORD
RECORD tooldata
  string string_comp;
  num time_event1;
  num time_event2;
  num time_event3;
ENDRECORD
RECORD intdata
  daintdata internal;
  string string_comp;
ENDRECORD

! The allocated data objects
PERS procdata prc_data{NOF_POSS_PROCS} := [["PROCDATA1], ...];
PERS tooldata tool_data{NOF_POSS_PROCS} := [["TOOLDATA1", 0.20,
     0.1, 0.05], ...];
PERS intdata int_data{NOF_POSS_PROCS} := [[[5, TRUE, TRUE, 1, 1,
     0, 0,""], "INTDATA1"], ...];

VAR tooldata cur_tool_data;
...
! Get process descriptors
DaGetPrcDescr proc_desc;
! Define the users data of the connected process
DaDefProcData proc_desc{1}, prc_data{1}, tool_data{1}, int_data{1};
...
! Get the current tool data of the connected process
DaGetCurrData prc_desc{1}, cur_tool_data, DA_TOOL_DATA;
```

Continues on next page

The allocated data object cur_tool_data will be get the current tool data (`DataSelect = DA_TOOL_DATA`) of the connected process `prc_desc`. This data can then be used as shown in the example below.

```
IF cur_tool_data.component1 = 1 THEN
  ...;
ENDIF
```

Arguments

```
DaGetCurrData ProcDesc Data DataSelect
```

`ProcDesc`

Process Descriptor

Data type: `dadescprc`

The descriptor of the connected process.

`Data`

Data type: `anytype`

The allocated data object to be updated with the selected current data.

`DataSelect`

Data Selector

Data type: `num`

The type of data to be get. The available data types are:

1	DA_PROC_DATA	Discrete application process data
2	DA_TOOL_DATA	Discrete application tool data
3	DA_INTPROC_DATA	Discrete application internal process data

Note

These data selectors are predefined in the system.

Limitations

If the data selector not valid, the program execution will result in a fatal RAPID user error.

Syntax

```
DaGetCurrData
  [ ProcDesc':=' ] < persistent array {*} (PERS) of dadescprc >
       ','
  [Data ':='] <variable (VAR) of anytype>
  [DataSelect ':='] <expression (IN) of num>
  ';
```

Related information

For information about	See
Application descriptor	*dadescapp - Discrete application - application descriptor on page 29*

Continues on next page

For information about	See
Process descriptor	*dadescprc - Discrete application - process descriptor on page 32*

3.2.9 DaProcML/MJ - Discrete Application - multiple processes

Description

`DaProcML` and `DaProcMJ` is used in discrete applications to control the motion and a set of up to 4 processes. `DaProcML` moves the TCP lineary to the target position. `DaProcMJ` moves the TCP non-lineary to the target position. Both instructions is calling the process RAPID user hooks during motion.

Examples

```
DaProcML p100, vmax, tool5;
```

The TCP of `tool5` is moved on a linear path to the position `p100` with the speed given in `vmax` and a set of up to 4 processes might be in preparation.

The process position is always a stop (discrete) position since the processes are always performed while the manipulator is standing still. The tools of the processes can be in preparation on the way to the position, that depends on the setup of the application processes. The processes are started and supervised until finished and the tools are in the home position.

```
DaProcMJ p100, vmax, tool5 \PreconError;
```

The TCP of `tool5` is moved on a non-linear path to the position `p100` with the speed given in `vmax` and no process is performed.

Arguments

```
DaProcML ToPoint Speed Tool [\WObj] [\InPos] [\PreconError] [\ID]
     [\TLoad]
```

`ToPoint`

Data type: `robtarget`

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

`Speed`

Data type: `speeddata`

The speed data that applies to movements. Speed data defines the velocity for the tool centre point, the tool reorientation and external axes.

`Tool`

Data type: `tooldata`

The tool in use when the robot moves. The tool centre point is the point moved to the specified destination position, and should be the position of the process tools.

`[\WObj]`

Work Object

Data type: `wobjdata`

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

Continues on next page

This argument can be omitted, and if it is, the position is related to the world coordinate system by using the default work object `wobj0`.

If, a stationary TCP or coordinated external axes are used, this argument must be specified in order to perform a movement relative to the work object.

Data type:

`[\InPos]`

In Position

Data type: `switch`

The optional switch argument `\InPos` inhibits the preactions of the connected processes. That means, if this argument is specified, the event times will be set internal to 0 for all the connected processes. The events will then be generated when the manipulator is in the target position.

`[\PreconError]`

Precondition Error

Data type: `switch`

The optional switch argument `\PreconError` indicates a precondition error of the connected processes. If this argument is specified, the manipulator will move to the target position without perfoming a process.

`[\ID]`

Synchronization id

Data type: `identno`

This argument must be used in a `MultiMove` system, if coordinated synchronized movement, and is not allowed in any other cases.

The specified id number must be the same in all cooperating program tasks. The id number gives a guarantee that the movements are not mixed up at runtime.

`[[\TLoad]`

Data type: `loaddata`

The `\TLoad` argument 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 `\TLoad` argument is used, then the `loaddata` in the current `tooldata` is not considered.

If the `\TLoad` argument is set to `load0`, then the `\TLoad` argument is not considered and the `loaddata` in the current `tooldata` is used instead. For a complete description of the `TLoad` argument, see the MoveL instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Continues on next page

Program execution

Internal sequence in a `DaProcML/DaProcMJ` instruction:

Sequence	Action	Information
If a precondition error is indicated:	Move to the target position without performing a process.	The used work object, tool and destination position is stored in: • da_current_wobj • da_current_tool • da_current_point and can be reused for some service functions etc
End of the `DaProcML/DaProcMJ` instruction		
If no precondition error is indicated:	Calculate the event times, if the argument `\InPos` is omitted, for all processes by calling the RAPID user hook `DaCalcEvtXX` (XX = Application name) and setup the time events.	Retrieve the calculated first time event from the discrete application framework.

Note

If the argument `\InPos` is defined, the RAPID user hook `DaCalcEvtXX` will not be called, instead all the event times will be setup with 0.

- Setup the three different I/O trigg actions to activate the RAPID process user hooks.
- Execute the movement towards the destination position with the trigg events on the path. If the argument `\InPos` is used, all the events will be generated when the manipulator has reached his destination position.
- The process sequences will be started and the RAPID user hooks will be called as described in *Programming discrete application on page 13*.
- Wait until the processes are ready or canceled.
- The default program execution is the concurrency mode, that means the next movement will be precalculated, but the manipulator will be hold (the next movement instruction is prepared). The manipulator will be released and carry on with the already precalculated movement after the processes are ready or canceled. The user can change the execution mode by setting the internal `daintdata` component `noconc` to TRUE. If the component `noconc` is set to TRUE, the program execution stops and waits for the ready signal of every process without precalculating the next movement.
- The current in use work object, tool and the destination position is stored in:

  A `da_current_wobj`

  B `da_current_tool`

  C `da_current_point` and can be reused for some service functions etc.
- End of the `DaProcML/DaProcMJ` instruction.

Continues on next page

Syntax

```
DaProcML/DaProcMJ
  [ ToPoint’:=’ ] < expression (IN) of robtarget > ’,’
  [ Speed’:=’ ] < expression (IN) of speeddata > ’,’
  [ Tool’:=’ ] < persistent (PERS) of tooldata >
  [ ’\’ WObj ’:=’ < persistent (PERS) of wobjdata > ]
  [ ’\’ InPos ]
  [ ’\’ PreconError ]
  [ ’\’ ID ’:=’ < expression (IN) of identno > ]
  [ '\' TLoad ':=' ] < persistent (PERS) of loaddata > ] ';'
```

Related information

For information about	See
Definition of velocity	Data type `speeddata` in *Technical reference manual - RAPID Instructions, Functions and Data types*.
Definition of zonedata	Data type `zonedata` in *Technical reference manual - RAPID Instructions, Functions and Data types*.
Definition of tool	Data type `tooldata` in *Technical reference manual - RAPID Instructions, Functions and Data types*.
Definition of work objects	Data type `wobjdata` in *Technical reference manual - RAPID Instructions, Functions and Data types*.
Definition of loaddata	Data type `loaddata` in *Technical reference manual - RAPID Instructions, Functions and Data types*.
`MoveL`	Instruction `MoveL` in *Technical reference manual - RAPID Instructions, Functions and Data types*.

3.2.10 DaSetCurrData - Discrete application - set current data

Description

`DaSetCurrData` is used to set a selected data of the connected process within the discrete application .

Examples

Sequence for define data for one process:

```
! Possible number of processes in the system
CONST num NOF_POSS_PROCS := 4;

! Allocate descriptors for the new processes
PERS dadescprc procdesc{NOF_POSS_PROCS};

! User defined data types for the process
RECORD procdata
  string string_comp;
ENDRECORD
RECORD tooldata
  string string_comp;
  num time_event1;
  num time_event2;
  num time_event3;
ENDRECORD
RECORD intdata
  daintdata internal;
  string string_comp;
ENDRECORD

! The allocated data objects
PERS procdata prc_data{NOF_POSS_PROCS} := [["PROCDATA1], ...];
PERS tooldata tool_data{NOF_POSS_PROCS} := [["TOOLDATA1", 0.20,
     0.1, 0.05], ...];
PERS intdata int_data{NOF_POSS_PROCS} := [[[5, TRUE, TRUE, 1, 1,
     0, 0,""], "INTDATA1"], ...];

VAR tooldata cur_tool_data;
...
! Get process descriptors
DaGetPrcDescr proc_desc;
! Define the users data of the connected process
DaDefProcData prc_desc{1}, prc_data{1}, tool_data{1},int_data{1};
...
! Get the current tool data of the connected process
DaGetCurrData proc_desc{1}, cur_tool_data, DA_TOOL_DATA;
...
cur_tool_data.string_comp := TOOLDATA2;
DaSetCurrData proc_desc, cur_tool_data, DA_TOOL_DATA;
```

Continues on next page

The tool data (`DataSelect = DA_TOOL_DATA`) of the connected process `proc_desc{1}` will be set to the new defined user tool data `cur_tool_data`.

Arguments

```
DaSetCurrData ProcDesc Data DataSelect
```

`ProcDesc`

Process Descriptor

Data type: `dadescprc`

The descriptor of the connected process.

`Data`

Data type: `anytype`

The data to be setup in the connected process.

`DataSelect`

Data Selector

Data type: `num`

The type of data to be get. The available data types are:

1	DA_PROC_DATA	Discrete application process data
2	DA_TOOL_DATA	Discrete application tool data
3	DA_INTPROC_DATA	Discrete application internal process data

Note

These data selectors are predefined in the system.

Limitations

If the data selector not valid, the program execution will result in a fatal RAPID user error.

Syntax

```
DaSetCurrData
  [ ProcDesc’:=’ ] < persistent array {*} (PERS) of dadescprc >
       ‘,’
  [ Data’:=’ ] < variable (VAR) of anytype > ‘,’
  [ DataSelect’:=’ ] < expression (IN) of num > ’;’
```

Related information

For information about	See
Application descriptor	*dadescapp - Discrete application - application descriptor on page 29*
Process descriptor	*dadescprc - Discrete application - process descriptor on page 32*

3.2.11 DaSetupAppBehav - Discrete application - sets up application behaviour

Description

DaSetupAppBehav enables the application writer to influence the framework. Usually the framework will call six of the eleven sequence hooks once. Five of them can be deactivated with aid of the instruction DaSetupAppBehav, namely DaPrepPrcXX, DaTmEvt1XX, DaTmEvt2XX, DaTmEvt3XX, DaStartXX. This will save time as each hook takes at least 30 ms to execute.

DaSetupAppBehav will affect all the active processes. A call to DaSetupAppBehav without arguments will activate all the deactivated sequence hooks, i.e. the framework will call all the five sequence hooks once.

Examples

```
! There is no code written in the both sequence hooks -
! DaTmEvt2XX and DaTmEvt3XX, so they will be deactivated.
DaSetupAppBehav \Exclude1:=TmEvt2 \Exclude2:=TmEvt3;
```

In this example the internal kernel won't make a call to neither DaTmEvt2XX or DaTmEvt3XX. This two sequence hooks won't be called for the activated processes.

Arguments

```
DaSetupAppBehav [\Exclude1] [\Exclude2] [\Exclude3] [\Exclude4]
     [\Exclude5]
```

[\Exclude1]

Data type: action_num

A selector connected to one of the five possible sequence hooks. The selector will deactivate the belonging sequence hook. The following table shows the possible selector constants.

sequence hook selector	sequence hook
DaPrepPrcXX	PrepPrc
DaTmEvt1XX	TmEvt1[\Exclude2]
DaTmEvt2XX	TmEvt2
DaTmEvt3XX	TmEvt3
DaStartXX	Start

[\Exclude2]

Same as \Exclude1.

[\Exclude3]

Same as \Exclude1.

[\Exclude4]

Same as \Exclude1.

[\Exclude5]

Same as \Exclude1.

Continues on next page

Limitations

The instruction must be called before calling the routine `DaProcML/DaProcMJ`.

Syntax

```
DaSetupAppBehav

[ ’\’ Exclude2 ’:=’ < expression (IN) of action_num > ]
[ ’\’ Exclude3 ’:=’ < expression (IN) of action_num > ]
[ ’\’ Exclude4 ’:=’ < expression (IN) of action_num > ]
[ ’\’ Exclude5 ’:=’ < expression (IN) of action_num > ]
```

3.2.12 DaStartManAction - Discrete application - execute an application manually

Description

`DaStartManAction` is used to run an application independently of the motion.

If no argument is used, the processes that are already active will run. If arguments are used, all other processes will be stopped and only the specified processes will run.

Examples

Example 1

```
! Execute the application independently of the motion
DaStartManAction;
```

Example 2

```
! Execute the application independently of the motion
! with process 1 and 3 running and the other processes stopped
DaStartManAction \Proc1 \Proc3;
```

Arguments

```
DaStartManAction [\Proc1] [\Proc2] [\Proc3] [\Proc4]
```

`[\Proc1]`

Data type: `switch`

Is used to run process 1 and stop all processes not specified as argument in the `DaStartManAction` instruction.

`[\Proc2]`

Data type: `switch`

Is used to run process 2 and stop all processes not specified as argument in the `DaStartManAction` instruction.

`[\Proc3]`

Data type: `switch`

Is used to run process 3 and stop all processes not specified as argument in the `DaStartManAction` instruction.

`[\Proc4]`

Data type: `switch`

Is used to run process 4 and stop all processes not specified as argument in the `DaStartManAction` instruction.

Syntax

```
DaStartManAction
  [ \Proc1 ]
  [ \Proc2 ]
  [ \Proc3 ]
  [ \Proc4 ]
```

Continues on next page

Related information

For information about	See
Application descriptor	*dadescapp - Discrete application - application descriptor on page 29*
Process descriptor	*dadescprc - Discrete application - process descriptor on page 32*

3.2.13 DaGetAppDescr - Discrete application - get application descriptors

Description

`DaGetAppDescr` is used to get the array of application descriptors from the application within the discrete application.

Examples

```
! Number of possible robots running an application. In a MultiMove
! system there will be possible to have four intances of an
! application, in a single system one.
CONST num MAX_NOF_ROB := 4;

! Application descriptor
PERS dadescapp app_desc{MAX_NOF_ROB};
...
! Get application descriptors
DaGetAppDescr app_desc;
```

This data can then be used as shown in the example below.

```
IF app_desc{1}.taskno = 1 THEN
  ...;
ENDIF
```

The descriptors of the application will be given to the allocated data object `app_desc`.

Arguments

```
DaGetAppDescr AppDesc
```

`AppDesc`

Application Descriptor

Data type: `dadescapp`

An allocated data object to get the application descriptor.

Limitations

The application name must not haave more than 5 characters. Otherwise the program execution will result in a fatal RAPID user error.

Syntax

```
DaGetAppDescr
  [ AppDesc':=' ] < persistent array {*} (PERS) of dadescapp > ';'
```

Related information

For information about	See
Application descriptor	*dadescapp - Discrete application - application descriptor on page 29*

3.2.14 DaGetAppIndex - Discrete application - index of application array

Description

`DaGetAppIndex` is used to find out what application descriptor current RAPID task uses.

Examples

```
! Number of possible robots running an application. In a MultiMove
! system there will be possible to have four intances of an
! application, in a single system one.
CONST num MAX_NOF_ROB := 4;
! Application descriptor
PERS dadescapp app_desc{MAX_NOF_ROB} := [[0, 0, 0, 0, 0, 0, 0, ""],
     ...];
...

! Index of the application descriptor array
VAR num index;

! Get which RAPID task is running now
DaGetAppIndex index;
```

This data can then be used as shown in the example below.

```
IF app_desc{index}.taskno = 1 THEN
  ...;
ENDIF

! In a MultiMove system there can be more than
! one robot -> more than one application descriptor
```

The application descriptors are saved in an array. The array is filled in when the system is starting up. To find out which application descriptor THIS task uses, the instruction `DaGetAppIndex` can be used. This instruction is only useful in a MultiMove system, where more than one task can run the application.

Arguments

```
DaGetAppIndex index
```

`index`

Data type: `num`

The index of the array of application descriptors.

Limitations

If no application is active, the program execution will result in a fatal RAPID user error.

Syntax

```
DaGetAppIndex [ index’:=’ ] < variable (VAR) of num> ’;’
```

Continues on next page

Related information

For information about	See
Application descriptor	*dadescapp - Discrete application - application descriptor on page 29*
Process descriptor	*dadescprc - Discrete application - process descriptor on page 32*

3.2.15 DaGetNumOfProcs - Discrete application - get number of processes

Description

`DaGetNumOfProcs` is used to find out how many processes that are installed in the system.

Examples

```
! Number of processes
VAR num NOF_PROCS;
...
! Get number of processes
DaGetNumOfProcs NOF_PROCS
```

Number of processes depends on how many `DA_PROC` tasks that are configured for the system. Two `DA_PROC` tasks installed means that `NOF_PROCS` will be two.

Arguments

```
DaGetNumOfProcs numofprocs
```

`numofprocs`

number of processes

Data type: `num`

Number of processes installed in the system.

Limitations

If no application is active, the program execution will result in a fatal RAPID user error.

Syntax

```
DaGetNumOfProcs [ numofprocs':=' ] < variable (VAR) of num> ';'
```

Related information

For information about	See
Application descriptor	*dadescapp - Discrete application - application descriptor on page 29*
Process descriptor	*dadescprc - Discrete application - process descriptor on page 32*

3.2.16 DaGetNumOfRob - Discrete application - number of robots

Description

DaGetNumOfRob is used to find out how many robots (i.e tasks) running the application, that are installed in the system.

Examples

```
! Number of robots
VAR num NOF_ROB;
...
! Get number of robots
DaGetNumOfRob NOF_ROB;
```

Number of robots depends on how many motion tasks in the system that are configured to run the application. A motion task (T_ROB1, T_ROB2..) runs an application if at least one process task (DA_PROC1, DA_PROC2...) is connected to the same mechanical unit group. In a single system all tasks use the same mechanical unit group, but in a MultiMove system that differs. For more information, see *Task installation on page 24*.

Arguments

```
DaGetNumOfRob numofrob
```

numofrob

number of robots

Data type: num

Number of application tasks installed in the system.

Limitations

If no application is active, the program execution will result in a fatal RAPID user error.

Syntax

```
DaGetNumOfRob [ numofrob':=' ] < variable (VAR) of num> ';'
```

Related information

For information about	See
Application descriptor	*dadescapp - Discrete application - application descriptor on page 29*
Process descriptor	*dadescprc - Discrete application - process descriptor on page 32*

3.2.17 DaGetPrcDescr - Discrete application - get process descriptor

Description

DaGetPrcDescr is used to get the array of all connected process descriptors of the application within the discrete application.

Examples

```
! Possible number of processes in the system.
CONST num NOF_POSS_PROCS := 4;
! Number of processes installed
PERS num NOF_POSS_PROCS := 1;
! Allocate descriptors for the new processes
PERS dadescprc proc_desc{NOF_POSS_PROCS};
...
! Get process descriptors
DaGetPrcDescr proc_desc;
```

This data can then be used as shown in the example below.

```
IF proc_desc{1}.taskno = 1 THEN
  ...;
ENDIF
```

The descriptors of the application will be given to the allocated data object proc_desc.

Arguments

```
DaGetPrcDescr ProcDesc AppDesc [\ProcName] | [\ProcNo]
```

ProcDesc

Process Descriptor

Data type: dadescprc

An allocated data object to get the process descriptor.

AppDesc

Application Descriptor

Data type: dadescapp

The descriptor of the connected application.

[\ProcName]

Process Name

Data type: string

The name of the connected process. If this argument is omitted, the connected process descriptor which refers to the process number will be retrieved.

[\ProcNo]

Process Number

Data type: num

Continues on next page

The number of the connected process. If this argument is omitted, the connected process descriptor which refers to the process name will be retrieved.

Limitations

One of the two optionals arguments (`\ProcName`, `\ProcNo`) must be specified, otherwise the program execution will result in an fatal RAPID user error.

Error handling

If a process, referenced either by the process name or process number, cannot be found, the system variable `ERRNO` is set to `ERR_DA_UNKPROC`. This error can then be handled in the RAPID error handler (see example below).

Example

```
...
VAR dadescapp app_desc;
VAR dadescprc prc_desc{4};
VAR string app_name;
VAR num proc_no;
  ...
  DaGetActApp app_desc, app_name;
  ...
  FOR i FROM 1 TO 4 DO
    proc_no := i;
    DaGetPrcDescr prc_desc{i}, app_desc \ProcNo:=proc_no;
  ENDFOR
  ...
ERROR
  IF (ERRNO = ERR_DA_UNKPROC) THEN
    TPWrite "Can’t find the process " \Num:=proc_no;
    TRYNEXT;
  ENDIF
```

If any of the processes cannot be found, the user will get a message about which process does not exist.

Syntax

```
DaGetPrcDescr
  [ ProcDesc’:=’ ] < variable (VAR) of dadescprc > ’,’
  [ AppDesc’:=’ ] < variable (VAR) of dadescapp >
  [’\’ProcName’:=’ ] < expression (IN) of string >
  | [ ‘\’ProcNo’:=’ ] < expression (IN) of num > ’;’
```

Related information

For information about	See
Application descriptor	*dadescapp - Discrete application - application descriptor on page 29*
Process descriptor	*dadescprc - Discrete application - process descriptor on page 32*

3.3 RAPID Functions

3.3.1 DaGetFstTimeEvt -Discrete application - get the first time event

Description

`DaGetFstTimeEvt` is used to get the first time event of all activated processes within the discrete application.

Examples

Sequence for define data for one process:

```
! Possible number of processes in the system
CONST num NOF_POSS_PROCS := 4;

! Allocate descriptors for the new processes
PERS dadescprc proc_desc{NOF_POSS_PROCS};

! The event times of the processes
VAR num evt_time_prc1{3} := [2.5, 1.8, 1.0];
VAR num evt_time_prc2{3} := [2.2, 1.7, 0.8]

! The first time event
VAR num first_time_event;
...
! Get process descriptors
DaGetPrcDescr proc_desc;

! Get number of processes
DaGetNumOfProcs NOF_PROCS;
! Setup the time events in DaCalcEvtXX
...
! Activate all processes
FOR i FROM 1 TO NOF_PROCS
  DaActProc proc_desc{i};
  ...
ENDFOR
! Get first time event
first_time_event := DaGetFstTimeEvt();
```

The content of the variable `first_time_event` will be 2.5 (the first time event which is specified in the current running processes: `evt_time_prc1{1}`) after the `DaGetFstTimeEvt` execution.

Return value

Data type: `num`

The first time event in seconds.

Continues on next page

Limitations

When using `DaGetFstTimeEvt` the processes must be activated. It will always return the first time event from the current activated processes.

If no process is active, the program execution will result in a fatal RAPID user error.

Syntax

```
DaGetFstTimeEvt '(' ')' ';'
```

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

Related information

For information about	See
Application descriptor	*dadescapp - Discrete application - application descriptor on page 29*
Process descriptor	*dadescprc - Discrete application - process descriptor on page 32*

3.3.2 DaCheckMMSOpt - Discrete application - Check if MultiMove system

Description

`DaCheckMMSOpt` is used to find out if this is a Single or MultiMove system.

Examples

```
IF (DaCheckMMSOpt()) THEN
  ...
ENDIF
```

If an option for MultiMove is installed, `DaChecMMSOpt` returns `TRUE`, otherwise `FALSE` (single system).

Return value

Data type: `bool`

TRUE: MultiMove system

FALSE: Single system

Limitations

If no application is active, the program execution will result in a fatal RAPID user error.

Syntax

```
DaCheckMMSOpt ´(´ ´) ´;´
```

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

Related information

For information about	See
Application descriptor	*dadescapp - Discrete application - application descriptor on page 29*
Process descriptor	*dadescprc - Discrete application - process descriptor on page 32*

3.3.3 DaGetMP - Discrete application - Get motion planner

Description

DaGetMP is used to get the motion planner that a specific application descriptor is configured for.

Examples

```
! Number of possible robots running an application. In a MultiMove
! system there will be possible to have four intances of an
! application, in a single system one.
CONST num MAX_NOF_ROB := 4;
! Application descriptor
PERS dadescapp app_desc{MAX_NOF_ROB} := [[0, 0, 0, 0, 0, 0, 0, ""],
     ...];
...
VAR num mp;
mp:= DaGetMP(1);
```

The application descriptors are saved in an array. The index of the array for a particular application descriptor is sent to DaGetMP. The motion planner that is configured for the descriptor is returned. This function is only useful in a MultiMove system, where all motion tasks uses different motion planners. For more information, see *Application manual - MultiMove*.

Return value

Data type: num

Number of motion planner

Arguments

```
DaGetMP(index)
```

Index

Data type: num

The index of the array of application descriptors.

Limitations

If no application is active, the program execution will result in a fatal RAPID user error.

Syntax

```
DaGetMP ´(´[ index ’:=’ ] < variable (VAR) of num> ´) ´;´
```

A function with a return value of the data type num.

Related information

For information about	See
Application descriptor	*dadescapp - Discrete application - application descriptor on page 29*

Continues on next page

For information about	See
Process descriptor	*dadescprc - Discrete application - process descriptor on page 32*

3.3.4 DaGetRobotName - Discrete application - Get Robot name

Description

`DaGetRobotName` is used to get the name of the robot that uses a specific application descriptor.

Examples

```
! Number of possible robots running an application.
! In a MultiMove system, it is possible to have four intances
! of an application, in a single system one.
CONST num MAX_NOF_ROB := 4;
! Application descriptor
PERS dadescapp app_desc{MAX_NOF_ROB} := [[0, 0, 0, 0, 0, 0, 0, ""],
     ...];
...
VAR string rob_name;
rob_name := DaGetRobotName(1);
```

The application descriptors are saved in an array. The index of the array for a particular application descriptor is sent to `DaGetRobotName`. The name of the robot that uses the descriptor is returned.

Return value

Data type: `string`

Name of robot

Arguments

```
DaGetRobotName(index)
```

`Index`

Data type: `num`

The index of the array of application descriptors.

Limitations

If no application is active, the program execution will result in a fatal RAPID user error.

Syntax

```
DaGetRobotName
  ´(´[ index ’:=’ ] < variable (VAR) of num> ´) ´;´
```

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

Related information

For information about	See
Application descriptor	*dadescapp - Discrete application - application descriptor on page 29*

Continues on next page

For information about	See
Process descriptor	*dadescprc - Discrete application - process descriptor on page 32*

3.3.5 DaGetTaskName - Discrete application - Get Task name

Description

`DaGetTaskName` is used to get the name of the of the task, that uses a specific application descriptor.

Examples

```
! Number of possible robots running an application.
! In a MultiMove system, it is possible to have four intances
! of an application, in a single system one.
CONST num MAX_NOF_ROB := 4;
! Application descriptor
PERS dadescapp app_desc{MAX_NOF_ROB} := [[0, 0, 0, 0, 0, 0, 0, ""],
     ...];
...
VAR string task_name;
task_name := DaGetTaskName(1);
```

The application descriptors are saved in an array. The index of the array for a particular application descriptor is sent to `DaGetTaskName`. The name of the task that uses the descriptor is returned.

Return value

Data type: `string`

Name of motion task.

Arguments

```
DaGetTaskName(index)
```

`Index`

Data type: `num`

The index of the array of application descriptors.

Limitations

If no application is active, the program execution will result in a fatal RAPID user error.

Syntax

```
DaGetTaskName
  ´(´[ index ’:=’ ] < variable (VAR) of num> ´) ´;´
```

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

Related information

For information about	See
Application descriptor	*dadescapp - Discrete application - application descriptor on page 29*

Continues on next page

For information about	See
Process descriptor	*dadescprc - Discrete application - process descriptor on page 32*

Contact us

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

ABB AS, Robotics
Discrete Automation and Motion
Nordlysvegen 7, N-4340 BRYNE, Norway
Box 265, N-4349 BRYNE, Norway
Telephone: +47 51489000

ABB Engineering (Shanghai) Ltd.
5 Lane 369, ChuangYe Road
KangQiao Town, PuDong District
SHANGHAI 201319, China
Telephone: +86 21 6105 6666

www.abb.com/robotics

3HAC050994-001, Rev -, en

Power and productivity
for a better world™