ABB Robotics

Technical reference manual
RAPID kernel

Power and productivity
for a better world™

Technical reference manual - RAPID kernel
3HAC16585-1
Revision G

Controller software IRC5

RobotWare 5.12

Table of contents

RAPID Kernel reference manual

Index

ABB AB
Robotics Products
SE-721 68 Västerås
Sweden

Contents

Contents

1 Introduction

This manual contains a formal description of the ABB Robotics robot programming language RAPID.

1.1 Design objectives

The RAPID language is aimed to support a levelled programming concept where new routines, data objects and data types may be installed at a specific IRB site. This concept makes it possible to customize (extend the functionality of) the programming environment and must be fully supported by the programming language.

In addition, with RAPID a number of powerful features are introduced:

- Modular Programming - Tasks/Modules
- Procedures and Functions
- Type definitions
- Variables, Persistents, and Constants
- Arithmetic
- Control Structures
- Backward Execution Support
- Error Recovery
- Undo Execution Support
- Interrupt Handling
- Placeholders

1.2 Language summary

Task - modules

A RAPID application is called a *task*. A task is composed of a set of *modules*. A module contains a set of data and routine declarations. The *task buffer* is used to host modules currently in use (execution, development) on a system.

RAPID distinguishes between *task modules* and s*ystem modules*. A task module is considered to be a part of the task/application while a system module is considered to be a part of the “system”. System modules are automatically loaded to the task buffer during system start-up and are aimed to (pre)define common, system specific data objects (tools, weld data, move data..), interfaces (printer, log file ..) etc.

While small applications usually are contained in a single task module (besides the system module/s), larger applications may have a "main" task module that in turn references routines and/or data contained in one or more other, "library" task modules.

One task module contains the *entry* procedure of the task. *Running* the task really means that the entry routine is executed. Entry routines must be parameterless.

Routines

There are three types of routines - *functions*, *procedures* and *traps*. A function returns a value of a specific type and is used in expression context. A procedure does not return any value and is used in statement context. Trap routines provide a means to respond to interrupts. A trap routine can be associated with a particular interrupt and is then later automatically executed if that interrupt occurs.

User (defined) routines are defined using RAPID declarations while *predefined* routines are supplied by the system and always available.

User routines

A RAPID routine declaration specifies the routine name, routine parameters, data declarations, statements, and possibly a backward handler and/or error handler and/or undo handler.

Predefined routines

There are two types of predefined routines - *built-in* routines and *installed* routines. Built-in routines (like arithmetic functions) are a part of the RAPID language while installed routines are application/equipment dependent routines used for the control of the robot arm, grippers, sensors etc. Note that from the users point of view there is no difference between built-in routines and installed routines.

Data objects

There are four types of data objects - *constants*, *variables*, *persistents* and *parameters*. A persistent (data object) can be described as a "persistent" variable. While a variable value is lost (re-initialized) at the beginning of each new session - at module load (module variable) or routine call (routine variable) - a persistent keeps its value between sessions. Data objects can be structured (record) and dimensioned (array, matrix etc.).

Statements

A statement may be *simple* or *compound*. A compound statement may in turn contain other statements. A *label* is a "no operation" statement that can be used to define named (goto-) positions in a program. Statements are executed in succession unless a *goto*, *return*, *raise*, *exit*, *retry*, or *trynext* statement, or the occurrence of an *interrupt* or *error* causes the execution to continue at another point.

The *assignment* statement changes the value of a variable, persistent, or parameter.

A *procedure call* invokes the execution of a procedure after associating any *arguments* with corresponding *parameters* of the procedure. RAPID supports *late binding* of procedure names.

The *goto* statement causes the execution to continue at a position specified by a label.

The *return* statement terminates the evaluation of a routine.

The *raise* statement is used to raise and propagate errors.

The *exit* statement terminates the evaluation of a task.

The *connect* statement is used to allocate an interrupt number and associate it with a trap (interrupt service) routine.

The *retry* and *trynext* statements are used to resume evaluation after an error.

The *if* and *test* statements are used for selection. The if statement allows the selection of a statement list based on the value of a condition. The test statement selects one (or none) of a set of statement lists, depending on the value of an expression.

The *for* and *while* statements are used for iteration. The for statement repeats the evaluation of a statement list as long as the value of a loop variable is within a specified value range. The loop variable is updated (with selectable increment) at the end of each iteration. The while statement repeats the evaluation of a statement list as long as a condition is met. The condition is evaluated and checked at the beginning of each iteration.

Backward execution

RAPID supports stepwise, backward execution of statements. Backward execution is very useful for debugging, test and adjustment purposes during RAPID program development. RAPID procedures may contain a *backward handler* (statement list) that defines the backward execution "behavior" of the procedure.

Error recovery

The occurrence of a runtime detected error causes suspension of normal program execution. The control may instead be passed to a user provided *error handler.* An error handler may be included in any routine declaration. The handler can obtain information about the error and possibly take some actions in response to it. If desirable, the error handler can return the control to the statement that caused the error (retry) or to the statement after the statement that caused the error (trynext) or to the point of the call of the routine. If further execution is not possible, at least the error handler can assure that the task is given a graceful abortion.

Undo execution

A routine can be aborted at any point by moving the program pointer out of the routine. In some cases, when the program is executing certain sensitive routines, it is unsuitable to abort. Using a undo handler it is possible to protect such sensitive routines against an unexpected program reset. The undo handler is executed automatically if the routine is aborted. This code should typically perform clean-up actions, for example closing a file.

Interrupts

Interrupts occur as a consequence of a user defined (interrupt) condition turning true. Unlike errors, interrupts are not directly related to (synchronous with) the execution of a specific piece of the code. The occurrence of an interrupt causes suspension of normal program execution and the control may be passed to a *trap routine*. After necessary actions have been taken in response to the interrupt the trap routine can resume execution at the point of the interrupt.

Data types

Any RAPID object (value, expression, variable, function etc.) has a *data type*. A data type can either be a *built-in* type or an *installed* type (compare installed routines) or a *user-defined* (defined in RAPID) type. Built-in types are a part of the RAPID language while the set of installed or user-defined types may differ from site to site. From the users point of view there is no difference between built-in, installed and user-defined types. There are three different kinds of types - *atomic* types, *record* types and *alias* types. The definition of an atomic type must be built-in or installed, but a record or alias type could also be user-defined.

Atomic types are "atomic" in the sense that they are not defined upon any other type and they cannot be divided into parts or components. Record types are built up by a set of named, ordered components. An alias type is by definition equal to another type. Alias types make it possible to classify data objects.

In addition to the atomic, record or alias classification of types, each type has a *value class*. There are three value classes of types - *value* types, *nonvalue* types and *semivalue* types. An object of value type is simply considered to represent some form of "value" (for example 3.55 or "John Smith"**)**. A nonvalue (type) object instead represents a hidden/encapsulated description of some physical or logical object, for example a file. Semivalue objects are special. They have two types, one "basic" nonvalue type and one *associated* value type that may be used to represent some property of the nonvalue type.

Built-in data types

The built-in atomic types are *bool*, *num*, *dnum*, and *string*. Bool is an enumerated type with the value domain {**TRUE**, **FALSE**} and provides a means of performing logical and relational computations. The num type supports exact and approximate arithmetic computations. The string type represents character sequences.

The built-in record types are *pos*, *orient*, *a*nd *pose*. The pos type represents a position in space (vector). The orient type represents an orientation in space. The pose type represents a coordinate system (position/orientation combination).

The built-in alias types are *errnum* and *intnum*. Errnum and intnum are both aliases for num and are used to represent error and interrupt numbers.

Operations on objects of built-in types are defined by means of arithmetic, relational and logical operators, and predefined routines.

Installed data types

The concept of installed types supports the use of installed routines by making it possible to use appropriate parameter types. An installed type can be either an Atomic, Record, or Alias type.

User-defined data types

The user-defined types make it easier to customize an application program. They also make it possible to write a RAPID program which is more readable.

Placeholders

The concept of *placeholders* supports structured creation and modification of RAPID programs. Placeholders may be used by offline and online programming tools to temporarily represent "not yet defined" parts of a RAPID program. A program that contains placeholders is syntactically correct and may be loaded to (and saved from) the task buffer. If the placeholders in a RAPID program do not cause any semantic errors (see 1.4), such a program can even be executed, but any placeholder encountered causes an execution error (see 1.4).

1.3 Syntax notation

The context-free syntax of the RAPID language is described using a modified variant of the Backus-Naur Form - EBNF:

- Boldface, upper case words denote *reserved words* and *placeholders*, for example **WHILE**
- Quoted strings denote other *terminal symbols*, for example ’+’
- Strings enclosed in angle brackets denote syntactic categories - *nonterminals*, for example <constant expression>
- The symbol "::=" means *is defined as*, for example <dim> ::= <constant expression>
- A list of terminals and/or nonterminals denotes a sequence, for example **GOTO**<identifier> ’;’

- Square brackets enclose optional items. The items may occur zero or one time, for example <return statement> ::= **RETURN** [<expression>] ';'
- The vertical bar separates alternative items, for example **OR** | **XOR**
- Braces enclose repeated items. The items may appear zero or more times. For example <statement list> ::= { <statement> }
- Parentheses are used to hierarchically group concepts together, for example (**OR**|**XOR**)<logical term>

1.4 Error classification

Based on the time of detection errors may be divided into:

- *Static* errors
- *Execution* errors

Static errors are detected either when a module is loaded into the task buffer (see 9) or before program execution after program modification:

- Lexical errors - illegal lexical elements. for example b := 2E52786; Exponent out of range
- Syntax errors - violation of the syntax rules. for example **FOR** i 5 **TO** 10 **DO** - Missing **FROM** keyword
- Semantic errors - violation of semantic rules - typically type errors, for example **VAR** num a; a := "John"; - Data type mismatch
- Fatal (system resource) errors, for example Program to complex (nested)

Execution errors occur (are detected) during the execution of a task:

- Arithmetic errors. for example Division by zero
- IO-errors. for example No such file or device
- Fatal (system resource) errors, for example Execution Stack overflow

The error handler concept of RAPID makes it possible to recover from nonfatal execution errors. See 7.

2 Lexical elements

This chapter defines the lexical elements of RAPID.

2.1 Character set

Sentences of the RAPID language are constructed using the standard ISO 8859-1 (Latin-1) character set. In addition *newline*, *tab*, and *formfeed* control characters are recognized.

<character> ::= -- *ISO 8859-1 (Latin-1)*--
<newline> ::= -- *newline control character* --
<tab> ::= -- *tab control character* --
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<hex digit> ::= <digit> | A | B | C | D | E | F | a | b | c | d | e | f
<letter> ::=
<upper case letter>
| <lower case letter>
<upper case letter> ::=
A | B | C | D | E | F | G | H | I | J
| K | L | M | N | O | P | Q | R | S | T
| U | V | W | X | Y | Z | À | Á | Â | Ã
| Ä | Å | Æ | Ç | È | É | Ê | Ë | Ì | Í
| Î | Ï | [1] | Ñ | Ò | Ó | Ô | Õ | Ö | Ø
| Ù | Ú | Û | Ü | [2] | [3] | ß
<lower case letter> ::=
a | b | c | d | e | f | g | h | i | j
| k | l | m | n | o | p | q | r | s | t
| u | v | w | x | y | z | ß | à | á | â | ã
| ä | å | æ | ç | è | é | ê | ë | ì | í
| î | ï | [1] | ñ | ò | ó | ô | õ | ö | ø
| ù | ú | û | ü | [2] | [3] | ÿ

1) Icelandic letter eth.
2) Letter Y with acute accent.
3) Icelandic letter thorn.

2.2 Lexical units

A RAPID sentence is a sequence of lexical units - tokens. The RAPID tokens are:

- identifiers
- reserved words
- literals
- delimiters
- placeholders
- comments

Tokens are indivisible. Except for string literals and comments, space must not occur within tokens.

An identifier, reserved word, or numeric literal must be separated from a trailing, adjacent identifier, reserved word, or numeric literal by one or more space, tab, formfeed, or newline characters. Other combinations of tokens may by separated by one or more space, tab, formfeed, or newline characters.

2.3 Identifiers

Identifiers are used for naming objects.

<identifier> ::=
<ident>
| **<ID>**
<ident> ::= <letter> {<letter> | <digit> | '_'}

The maximum length of an identifier is 32 characters. All characters of an identifier are significant. Identifiers differing only in the use of corresponding upper and lower case letters are considered the same. The placeholder **<ID>** (see *Placeholders on page 5* and *2.9 Placeholders on page 11*) can be used to represent an identifier.

2.4 Reserved words

The 56 identifiers listed below are reserved words. They have a special meaning in the RAPID language. They may not be used in any context not specially stated by the syntax.

ALIAS	AND	BACKWARD
CASE	CONNECT	CONST
DEFAULT	DIV	DO
ELSE	ELSEIF	ENDFOR
ENDFUNC	ENDIF	ENDMODULE
ENDPROC	ENDRECORD	ENDTEST
ENDTRAP	ENDWHILE	ERROR
EXIT	FALSE	FOR
FROM	FUNC	GOTO
IF	INOUT	LOCAL
MOD	MODULE	NOSTEPIN
NOT	NOVIEW	OR
PERS	PROC	RAISE
READONLY	RECORD	RETRY
RETURN	STEP	SYSMODULE
TEST	THEN	TO
TRAP	TRUE	TRYNEXT
UNDO	VAR	VIEWONLY
WHILE	WITH	XOR

2.5 Numerical literals

A numerical literal represents a numeric value.

<num literal> ::=
<integer> [<exponent>]
| <decimal integer>) [<exponent>]
| <hex integer>
| <octal integer>
| <binary integer>
| <integer> ’.’ [<integer>] [<exponent>]

| [<integer>] '.' <integer> [<exponent>]
<integer> ::= <digit> {<digit>}
<decimal integer> ::= '0' ('D' | 'd') <integer>
<hex integer> ::= '0' ('X' | 'x') <hex digit> {<hex digit>}
<octal integer> ::= '0' ('O' | 'o') <octal digit> {<octal digit>}
<binary integer> ::= '0' ('B' | 'b') <binary digit> {<binary digit>}

<exponent> ::= ('E' | 'e') ['+' | '-'] <integer>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<hex digit> ::= <digit> | A | B | C | D | E | F | a | b | c | d | e | f
<octal digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
<binary digit> ::= 0 | 1

A numerical literal must be in the range specified by the ANSI IEEE 754 Standard for Floating-Point Arithmetic.

For example: 7990 23.67 2E6 .27 2.5E-3 38.

2.6 Bool literals

A bool literal represents a logical value.

<bool literal> ::= **TRUE** | **FALSE**

2.7 String literals

A string literal is a sequence of zero or more characters enclosed by the double quote (") character.

<string literal> ::= '"' { <character> | <character code> } '"'
<character code> ::= '\' <hex digit> <hex digit>

The possibility to use character codes provides a means to include non printable characters (binary data) in string literals. If a back slash or double quote character should be included in a string literal it must be written twice.

For example: "A string literal"
"Contains a "" character"
"Ends with BEL control character\07"
"Contains a \\ character"

2.8 Delimiters

A delimiter is one of the following characters:

{ } ( ) [ ] , . = < > + - * / : ; ! \ ?

or one of the following compound symbols:

:= <> >= <=

2.9 Placeholders

Placeholders can be used by offline and online programming tools to temporarily represent "not yet defined" parts of a RAPID program. A program that contains placeholders is syntactically correct and can be loaded to (and saved) from the task buffer. If the placeholders in a RAPID program does not cause any semantic errors (see *1.4 Error classification on page 6*), such a program can even be executed, but any placeholder encountered causes an execution error (see *1.4 Error classification on page 6*). RAPID recognizes the following 13 placeholders:

<TDN> - (represents a) data type definition
<DDN> - (represents a) data declaration
<RDN> - routine declaration
<PAR> - parameter declaration
<ALT> - alternative parameter declaration
<DIM> - array dimension
<SMT> - statement
<VAR> - data object (variable, persistent or parameter) reference
<EIT> - else if clause of if statement
<CSE> - case clause of test statement
<EXP> - expression
<ARG> - procedure call argument
<ID> - identifier

2.10 Comments

A comment starts with an exclamation mark and is terminated by a newline character. A comment can never include a newline character.

<comment> ::= ’!’ { <character> | <tab> } <newline>

Comments have no effect on the meaning of a RAPID code sequence, their sole purpose is to clarify the code to the reader.

Each RAPID comment occupies an entire source line and may occur either as:

- an element of a type definition list (see bullet 3 below),
- an element of a record component list,
- an element of a data declaration list (see *5.3 Procedure declarations on page 52*),
- an element of a routine declaration list (see *9.1 Module declarations on page 78*) or
- an element of a statement list (see *4.2 Statement lists on page 38*).

For example:

```
! Increase length
length := length + 5;
IF length < 1000 OR length > 14000 THEN
  ! Out of bounds
  EXIT;
ENDIF
...
```

Comments located between the last data declaration (see *2.18 Data declarations on page 20*) and the first routine declaration (see *5 Routine declarations on page 49*) of a module are regarded to be a part of the routine declaration list. Comments located between the last data declaration and the first statement of a routine are regarded to be a part of the statement list (see *4.2 Statement lists on page 38*).

2.11 Data types

A RAPID data type is identified by its name and can be either *built-in, installed,* or *user-defined* (defined in RAPID).

<data type> ::= <identifier>

Built-in types are part of the RAPID language while the set of installed or user-defined types may differ from site to site. Please see site specific documentation. The concept of installed types supports the use of installed routines by making it possible to use appropriate parameter types. The user-defined types make it possible to prepare understandable and easy programmable application packets for the application engineer. From the users' point of view there is no difference between built-in, installed, and user-defined types.

There are three different types - *atomic* types, *record* types, and *alias* types.

A *type definition* introduces an alias or a record by associating an identifier with a description of a data type. A type definition can be represented by the placeholder **<TDN>**.

<type definition> ::=
[**LOCAL**] (<record definition>
| <alias definition>)
| <comment>
| <TDN>

Type definitions can occur in the heading section of modules (see *9 Task modules on page 77*).

The optional local directive classifies the data object being *local*, otherwise *global* (see *2.20 Scope rules on page 21*).

For example: **LOCAL RECORD** object - record definition
num usecount;
string name;
ENDRECORD

ALIAS num another_num; - alias definition

<TDN>- definition placeholder

2.12 Scope rules

The *scope* of a type definition denotes the area in which the type is *visible* and is determined by the context and position of its declaration.

The scope of a predefined type comprises any RAPID module.

A user-defined type is always defined inside a module. The following scope rules are valid for module type definitions:

- The scope of a local module type definition comprises the module in which it is contained.
- The scope of a global module type definition in addition comprises any other module in the task buffer.
- Within its scope a module type definition hides any predefined type with the same name.
- Within its scope a local module type definition hides any global module type with the same name.
- Two module objects declared in the same module may not have the same name.
- Two global objects declared in two different modules in the task buffer may not have the same name.

2.13 Atomic types

Atomic types are "atomic" in the sense that they are not defined upon any other type and cannot be divided into parts or components. The internal structure (implementation) of an atomic type is hidden. The built-in atomic types are the numeric types *num* and *dnum*, the logical type *bool*, and the text type *string*.

Num type

A num object represents a numeric value. The num type denotes the domain specified by the ANSI IEEE 754 Standard for Floating-Point Arithmetic.

Within the subdomain -8388607 to (+)8388608, num objects may be used to represent integer (exact) values. The arithmetic operators +, -, and * (see *3.11 Operators on page 34*) preserves integer representation as long as operands and result are kept within the integer subdomain of num.

Examples of num usage;

```
VAR num counter;      - variable declaration
counter := 250;       - num literal usage
```

Dnum type

A dnum object represents a numeric value. The num type denotes the domain specified by the ANSI IEEE 754 Standard for Floating-Point Arithmetic.

Within the subdomain -4503599627370496 to (+)4503599627370496, dnum objects may be used to represent integer (exact) values. The arithmetic operators +, -, and * (see *3.11 Operators on page 34*) preserves integer representation as long as operands and result are kept within the integer subdomain of dnum.

Examples of dnum usage;

```
VAR dnum value;       - variable declaration
value := 2E+43;       - dnum literal usage
```

Bool type

A bool object represents a logical value.

The bool type denotes the domain of twovalued logic - { **TRUE**, **FALSE** }.

Examples of bool usage;

```
VAR bool active;      - variable declaration
active := TRUE;       - bool literal usage
```

String type

A string object represents a character string.

The string type denotes the domain of all sequences of graphical characters (ISO 8859-1) and control characters (non ISO 8859-1 characters in the numeric code range 0 .. 255). A string may consist of 0 to 80 characters (fixed 80 characters storage format).

Examples of string usage;

```
VAR string name;        - variable declaration
name := "John Smith"; - string literal usage
```

2.14 Record types

A record data type is a composite type with named, ordered components. The value of a record type is a composite value consisting of the values of its components. A component can have atomic type or record type. Semivalue types cannot be included in a record type. The built-in record types are *pos*, *orient*, and *pose*. The available set of installed and user-defined record types is by definition not bound by the RAPID specification. Please see site specific documentation.

A record type is introduced by a record definition.

```
<record definition> ::=
        RECORD <identifier>
            <record component list>
        ENDRECORD
<record component list> ::=
        <record component definition> |
        <record component definition> <record component list>
<record component definition> ::=
        <data type>  <record component name> ';'
```

For example:
```
RECORD newtype
  num x;
ENDRECORD
```

A record value can be expressed using an *aggregate* representation.

For example:
```
[ 300, 500, depth ]      - pos record aggregate value
```

A specific component of a record data object can be accessed by using the name of the component.

For example:
```
p1.x := 300;- assignment of x-component of pos variable p1
```

Unless otherwise stated the domain of a record type is the cartesian product of the domains of its components.

Pos type

A pos object represents a vector (position) in 3D space. The pos type has three components:

[x, y, z]

name	data type	comment
x	num	x-axis component of position
y	num	y-axis component of position
z	num	z-axis component of position

Examples of pos usage:

```
VAR pos p1;              - variable declaration
p1 := [ 10, 10, 55.7 ];  - aggregate usage
p1.z := p1.z + 250;      - component usage
p1 := p1 + p2;           - operator usage
```

Orient type

An orient object represents an orientation (rotation) in 3D space. The orient type has four components:

[q1, q2, q3, q4]

name	data type	comment
q1	num	first quarternion component
q2	num	second quarternion component
q3	num	third quarternion component
q4	num	fourth quarternion component

The quarternion representation is the most compact way to express an orientation in space. Alternate orientation formats (for example euler angles) can be specified using predefined functions available for this purpose.

Examples of orient usage:

```
VAR orient o1;              - variable declaration
o1 := [ 1, 0, 0, 0 ];       - aggregate usage
o1.q1 := -1;                - component usage
o1 := euler( a1, b1, g1);- function usage
```

Pose type

A pose object represents a 3D frame (coordinate system) in 3D-space**.** The pose type has two components:

[trans, rot]

name	data type	comment
trans	pos	origin translation
rot	orient	rotation

Examples of pose usage:

```
VAR pose p1;                        - variable declaration
p1 := [[ 100, 100, 0 ], o1 ];       - aggregate usage
p1.trans := homepos;                - component usage
```

2.15 Alias types

An alias data type is defined as being equal to another type. Alias types provide a means to classify objects. The system may use the alias classification to look up and present type related objects.

An alias type is introduced by an alias definition.

<alias definition> ::=
ALIAS <type name> <identifier> ’;’

For example: **ALIAS** num newtype;

For example: Usage of alias type ’level’ (alias for num):

```
CONST level low := 2.5;
CONST level high := 4.0;
```

Note that one alias type cannot be defined upon another alias type.

The built-in alias types are *errnum* and *intnum* - both aliases for num.

Errnum type

The errnum type is an alias for num and is used for the representation of error numbers.

Intnum type

The intnum type is an alias for num and is used for the representation of interrupt numbers.

2.16 Data type value classes

With respect to the relation between object data *type* and object *value*, data types can be classified as being either:

- *value* data type
- *nonvalue* (private) data type
- *semivalue* data type.

An object of value type is simply considered to represent some form of "value" (for example 5, [10, 7, 3.25], "John Smith", **TRUE)**. A nonvalue (type) object instead represents a hidden/encapsulated description (descriptor) of some physical or logical object, for example the *iodev* (file) type.

The content ("value") of nonvalue objects can only be manipulated using installed routines ("methods"). Nonvalue objects may in RAPID programs only be used as arguments to *var* or *ref* parameters.

For example: Use of nonvalue object ’logfile’

```
VAR iodev logfile;
  ...
! Open logfile
Open "flp1:LOGDIR" \File := "LOGFILE1.DOC ", logfile;
  ...
! Write timestamp to logfile
Write logfile, "timestamp = " + GetTime();
```

Semivalue objects are special. They have two types, one "basic" nonvalue type and one *associated* (value) type that may be used to represent some property of the nonvalue type. RAPID views a semivalue object as a value object when used in value context (se table below) and a nonvalue object otherwise. The semantics (meaning/result) of a read or update (value) operation performed upon a semivalue type is defined by the type itself.

For example: Use of semivalue object ’sig1’ in value context (the associated type of signaldi is num).

```
VAR signaldi sig1;
  ...
! use digital input sig1 as value object
IF sig1 = 1 THEN ...
  ...
! use digital input sig1 as nonvalue object
IF DInput(sig1) = 1 THEN ...
```

Note that a semivalue object (type) can reject either reading or updating "by value".

For example, the following assignment will be rejected by sig1 since sig1 represents an input device.

```
VAR signaldi sig1;
  ...
sig1 := 1;
```

The table below shows which combinations of object usage and type value class that are possibly legal and which are impossible/illegal:

XLegal/Possible
---Illegal/Impossible

Data object usage

Object Declarations	value	nonvalue	semivalue
Constant	X	---	
Persistent	X	---	
Variable with initialization	X	---	
Variable without initialization	X	X	X
Routine Parameter			
in	X	---	---
var	X	X	X
pers	X	---	---
ref (only installed routines)	X	X	X
inout:			
var	X	---	---
pers	X	---	---
Function return value	X	---	---
Object Reference	value	nonvalue	semivalue
Assignment[2)]	X	---	X[1)]
Assignment	X	X[3]	X[3)]
Assignment[4)]	X	---	X[1)]

1) The associated type (value) is used.
2) Assignment target (see 4.4), Connect target (see 4.12).
3) Argument to var or ref parameter.
4) Object used in expression.

The basic value types are the built-in atomic types num, dnum, bool, and string. A record type with all components being value types is itself a value type, for example the built-in types pos, orient, and pose. An alias type defined upon a value type is itself a value type, for example the built-in types errnum and intnum.

A record type having at least one semivalue component and all other components have value type is itself a semivalue type. An alias type defined upon a semivalue type is itself a semivalue type.

All other types are nonvalue types, for example record types with at least one nonvalue component and alias types defined upon nonvalue types.

Arrays have the same value class as the element value class.

2.17 Equal types

The types of two objects are *equal* if the objects have the same *structure* (degree, dimension and number of components) and either:

- Both objects have the same type name (any alias type name included is first replaced by its definition type).
- One of the objects is an aggregate (array or record) and the types of (all) corresponding elements/components are *equal*.
- One of the objects has a value type, the other object has a semivalue type and the type of the first object and the associated type of the semivalue object are *equal*. Note that this is only valid in value context.

2.18 Data declarations

There are four kinds of data objects - *constants, variables, persistents*, and *parameters*. Except for *predefined data objects* (see *2.19 Predefined data objects on page 21* and appendix C) and for loop variables (see *4.15 For statement on page 45*) all data objects must be declared. A *data declaration* introduces a constant, a variable, or a persistent by associating an identifier with a data type. See *5.1 Parameter declarations on page 50* for information on parameter declarations. A data declaration can be represented by the placeholder **<DDN>**.

<data declaration> ::=
[**LOCAL**] (<variable declaration>
| <persistent declaration>
| <constant declaration>)
| **TASK** (<variable declaration>
| <persistent declaration>
| <comment>
| **<DDN>**

A persistent (data object) can be described as a "persistent" variable. While a variable value is lost (re-initialized) at the beginning of each new session - at module load (module variable) or routine call (routine variable) - a persistent keeps its value between sessions. This is accomplished by letting an update of the value of a persistent automatically lead to an update of the initialization value of the persistent declaration. When a module (or task) is saved, the initialization value of any persistent declaration reflects the current value of the persistent. In addition, persistents are stored in a system public "database" and can be accessed (updated, referenced) by other components of the control system.

Data declarations can occur in the heading section of modules (see *9 Task modules on page 77*) and routines (see *5 Routine declarations on page 49*).

The optional local directive classifies the data object being *local*, otherwise *global* (see *2.20 Scope rules on page 21*). Note that the local directive only may be used at module level (not inside a routine).

The optional task directive classifies persistent data objects and variable data objects being task *global* as opposed to system *global*. In the scope rules there is no difference between the two global types.

However the current value of a task global persistents will always be unique to the task and not shared among other tasks. System global persistents in different tasks share current value if they are declared with the same name and type.

Declaring a variable as task global will only be effective in a module that is installed shared. System global variables in loaded or installed modules are already unique to the task and not shared among other tasks.

Note that the task directive only may be used at module level (not inside a routine).

For example:

LOCAL VAR num counter;- variable declaration
CONST num maxtemp := 39.5;- constant declaration
PERS pos refpnt := [100.23, 778.55, 1183.98];- persistent declaration
TASK PERS num lasttemp := 19.2;- persistent declaration
<DDN> - declaration placeholder

2.19 Predefined data objects

A predefined data object is supplied by the system and is always available. Predefined data objects are automatically declared and can be referenced from any module. See appendix C for the description of built-in data objects.

2.20 Scope rules

The *scope* of a data object denotes the area in which the object is *visible* and is determined by the context and position of its declaration.

The scope of a predefined data object comprises any RAPID module.

A data object declared outside any routine is called a *module data object* (*module variable*, *module constant* or *persistent*). The following scope rules are valid for module data objects:

- The scope of a local module data object comprises the module in which it is contained.

- The scope of a global module data object in addition comprises any other module in the task buffer.
- Within its scope a module data object hides any predefined object with the same name.
- Within its scope a local module data object hides any global module object with the same name.
- Two module objects declared in the same module may not have the same name.
- Two global objects declared in two different modules in the task buffer may not have the same name.
- A global data object and a module may not share the same name.

A data object declared inside a routine is called a *routine data object* (*routine variable* or *routine constant*). Note that the concept of routine data objects in this context also comprises routine parameters (see *5.1 Parameter declarations on page 50*).

The following scope rules are valid for routine data objects:

- The scope of a routine data object comprises the routine in which it is contained.
- Within its scope a routine data object hides any predefined or user defined object with the same name.
- Two routine data objects declared in the same routine may not have the same name.
- A routine data object may not have the same name as a label declared in the same routine.
- See *5 Routine declarations on page 49* and *9 Task modules on page 77* for information on routines and task modules.

2.21 Storage class

The *storage class* of a data object determines when the system allocates and deallocates memory for the data object. The storage class of a data object is determined by the kind of data object and the context of its declaration and can be either *static* or *volatile*.

Constants, persistents, and module variables are static. The memory needed to store the value of a static data object is allocated when the module that declares the object is loaded (see *9 Task modules on page 77*). This means that any value assigned to a persistent or a module variable always remains unchanged until the next assignment.

Routine variables (and *in* parameters, see *5.1 Parameter declarations on page 50*) are volatile. The memory needed to store the value of a volatile object is allocated first upon the call of the routine in which the declaration of the variable is contained. The memory is later deallocated at the point of the return to the caller of the routine. This means that the value of a routine variable is always undefined before the call of the routine and is always lost (becomes undefined) at the end of the execution of the routine.

In a chain of recursive routine calls (a routine calling itself directly or indirectly) each instance of the routine receives its own memory location for the "same" routine variable - a number of *instances* of the same variable are created.

2.22 Variable declarations

A variable is introduced by a variable declaration.

<variable declaration> ::=
VAR <data type> <variable definition> ';'

<variable definition> ::=
<identifier> ['{' <dim> { ',' <dim> } '}']
[':=' <constant expression>]

<dim> ::= <constant expression>

For example: **VAR** num x;
VAR pos curpos := [b+1, cy, 0];

Variables of any type (including installed types) can be given an array (of degree 1, 2, or 3) format by adding dimension information to the declaration. The dimension expression must represent an integer value (see *Num type on page 14*) greater than 0.

For example:
```
! pos (14 x 18) matrix
VAR pos pallet{14, 18};
```

Variables with value types (see *2.16 Data type value classes on page 18*) may be initialized (given an initial value). The data type of the constant expression used to initialize a variable must be equal to the variable type.

For example:
```
VAR string author_name := "John Smith";
VAR pos start := [100, 100, 50];
VAR num maxno{10} := [1, 2, 3, 9, 8, 7, 6, 5, 4, 3];
```


An uninitialized variable (or variable component/element) receives the following initial value.

Data Type	Initial Value
num (or alias for num)	0
dnum (or alias for dnum)	0
bool (or alias for bool)	FALSE
string (or alias for string)	""
Installed atomic types	all bits 0'ed

As described in chapter *2.18 Data declarations on page 20*, variables can be declared as local, task, or system global.

2.23 Persistent declarations

A persistent is introduced by a persistent declaration. Note that persistents can only be declared at module level (not inside a routine). A persistent can be given any value data type.

<persistent declaration> ::=
PERS <data type> <persistent definition> ';'

<persistent definition> ::=
<identifier> ['{' <dim> { ',' <dim> } '}']
[':=' <literal expression>]
Note! The literal expression may only be omitted for system global persistents.

For example: **PERS** num pcounter := 0;

Persistents of any type (including installed types) can be given an array (of degree 1, 2 or 3) format by adding dimension information to the declaration. The dimension expression must represent an integer value (see *Num type on page 14*) greater than 0.

For example:
```
! 2 x 2 matrix
PERS num grid{2, 2} := [[0, 0], [0, 0]];
```

As described in chapter *2.18 Data declarations on page 20*, persistents can be declared as local, task global or system global. Local and task global persistents must be initialized (given an initial value). For system global persistents the initial value may be omitted. The data type of the literal expression used to initialize a persistent must be equal to the persistent type. Note that an update of the value of a persistent automatically leads to an update of the initialization expression of the persistent declaration (if not omitted).

For example:
```
MODULE ...
  PERS pos refpnt := [0, 0, 0];
  ...
  refpnt := [x, y, z];
  ...
ENDMODULE
```

If the value of the variables x, y and z at the time of execution is 100.23, 778.55, and 1183.98 respectively and the module is saved, the saved module will look like this:

```
MODULE ...
  PERS pos refpnt := [100.23, 778.55, 1183.98];
  ...
  refpnt := [x, y, z];
  ...
ENDMODULE
```

A persistent without initial value (or persistent component/element) receives the following initial value.

Data Type	Initial Value
num (or alias for num)	0
dnum (or alias for dnum)	0
bool (or alias for bool)	FALSE
string (or alias for string)	""
Installed atomic types	all bits 0’ed

2.24 Constant declarations

A constant represents a static value and is introduced by a constant declaration. The value of a constant cannot be modified. A constant can be given any value data type.

<constant declaration> ::=
CONST <data type> <constant definition> ’;’
<constant definition> ::=
<identifier> [’{’ <dim> { ’,’ <dim> } ’}’]
’:=’ <constant expression>
<dim> ::= <constant expression>

For example: **CONST** num pi := 3.141592654;
CONST num siteno := 9;

A constant of any type (including installed types) can be given an array (of degree 1, 2 or 3) format by adding dimensioning information to the declaration. The dimension expression must represent an integer value (see *Num type on page 14*) greater than 0. The data type of the constant value must be equal to the constant type.

For example: **CONST** pos seq{3} := [[614, 778, 1020],
[914, 998, 1021],
[814, 998, 1022]];

3 Expressions

An expression specifies the evaluation of a value. An expression can be represented by the placeholder **<EXP>**.

<expression> ::=
<expr>
| **<EXP>**
<expr> ::= [**NOT**] <logical term> { (**OR** | **XOR**) <logical term> }
<logical term> ::= <relation> { **AND** <relation> }
<relation> ::= <simple expr> [<relop> <simple expr>]
<simple expr> ::= [<addop>] <term> { <addop> <term> }
<term> ::= <primary> { <mulop> <primary> }
<primary> ::=
<literal>
| <variable>
| <persistent>
| <constant>
| <parameter>
| <function call>
| <aggregate>
| ’(’ <expr> ’)’
<relop> ::= ’<’ | ’<=’ | ’=’ | ’>’ | ’>=’ | ’<>’
<addop> ::= ’+’ | ’-’
<mulop> ::= ’*’ | ’/’ | **DIV** | **MOD**

The relative *priority* of the operators determines the order in which they are evaluated. Parentheses provide a means to override operator priority. The rules above imply the following operator priority:

* / **DIV MOD** - highest
\+ -
< > <> <= >= =
AND
XOR OR NOT - lowest

An operator with high priority is evaluated before an operator with low priority. Operators of the same priority are evaluated from left to right.

Examples of evaluation order:

For example
a + b + c--> (a + b) + c-- left to right rule
a + b * c --> a + (b * c)-- * higher than +
a **OR** b **OR** c--> (a **OR** b) **OR** c-- Left to right rule
a **AND** b **OR** c **AND** d--> (a **AND** b) **OR** (c **AND** d)
a < b **AND** c < d--> (a < b) **AND** (c < d**)**

The left operand of a binary operator [1)] is evaluated prior to the right operand. Note that the evaluation of expressions involving **AND** and **OR** operators is *optimized* so that the right operand of the expression will not be evaluated if the result of the operation can be determined after the evaluation of the left operand.

3.1 Constant expressions

Constant expressions are used to represent values in data declarations.

<constant expression> ::= <expression>

A constant expression is a specialization of an ordinary expression. It may not at any level contain:

- Variables, Persistents
- Function calls

For example
```
CONST num radius := 25;
CONST num pi := 3.141592654;
! constant expression
CONST num area := pi * radius * radius;
```

3.2 Literal expressions

Literal expressions are used to represent initialization values in persistent declarations.

<literal expression> ::= <expression>

A literal expression is a specialization of an ordinary expression. It may only contain either a single literal value (+ or - may precede a numerical literal) or a single aggregate with members that in turn are literal expressions.

For example
```
PERS pos refpnt := [100, 778, 1183];
PERS num diameter := 24.43;
```

3.3 Conditional expressions

Conditional expressions are used to represent logical values.

<conditional expression> ::= <expression>

A conditional expression is a specialization of an ordinary expression. The resulting type must be bool (**TRUE**/**FALSE**).

For example
```
counter > 5 OR level < 0        - conditional expression
```

1. An operator that takes two operands, that is +, -, * etc.

3.4 Literals

A literal is a lexical element (indivisible) that represents a constant value of a specific data type.

<literal> ::= <num literal>
| <string literal>
| <bool literal>

For example	0.5, 1E2	- numerical literals
	"limit"	- string literal
	TRUE	- bool literal

3.5 Variables

Depending on the type and dimension of a variable it may be referenced in up to three different ways. A variable reference may mean the entire variable, an element of a variable (array) or a component of a variable (record).

<variable> ::=
<entire variable>
| <variable element>
| <variable component>

A variable reference denotes, depending on the context, either the value or the location of the variable.

Entire variable

An entire variable is referenced by the variable identifier.

<entire variable> ::= <ident>

If the variable is an array the reference denotes all elements. If the variable is a record the reference denotes all components. Note that the placeholder **<ID>** (see *2.3 Identifiers on page 8*) cannot be used to represent an entire variable.

For example
```
VAR num row{3};
VAR num column{3};
    ..
! array assignment
row := column;
```

Variable element

An array variable element is referenced using the index number of the element.

<variable element> ::= <entire variable> ’{’ <index list> ’}’

<index list> ::= <expr> { ’,’ <expr> }

An index expression must represent an integer value (see *Num type on page 14*) greater than 0. Index value 1 selects the first element of an array. An index value may not violate the declared dimension. The number of elements in the index list must fit the declared degree (1, 2, or 3) of the array.

For example column{10}- reference of the tenth element of column

mat{i * 10, j}- reference of matrix element

Variable component

A record variable component is referenced using the component name (names).

<variable component> ::= <variable> ’.’ <component name>

<component name> ::= <ident>

Note that the placeholder **<ID>** (see *2.3 Identifiers on page 8*) cannot be used to represent a component name.

For example	home.y	- referencing the Y component of a pos variable
	parr{i+2}.x	- referencing the X component of an element of a pos array
	p1.trans.x	- referencing a component (x) of another component (trans)

3.6 Persistents

A persistent reference may mean the entire persistent, an element of a persistent (array) or a component of a persistent (record).

<persistent> ::=
<entire persistent>
| <persistent element>
| <persistent component>

The rules concerning persistent references comply with the rules concerning variable references *Entire variable on page 29*, *Variable element on page 29*, and *Variable component on page 30*.

3.7 Constants

A constant reference may mean the entire constant, an element of a constant (array) or a component of a constant (record).

<constant> ::=
<entire constant>
| <constant element>
| <constant component>

The rules concerning constant references comply with the rules concerning variable references *Entire variable on page 29*, *Variable element on page 29*, and *Variable component on page 30*.

3.8 Parameters

A parameter reference may mean the entire parameter, an element of a parameter (array) or a component of a parameter (record).

<parameter> ::=
<entire parameter>
| <parameter element>
| <parameter component>

The rules concerning parameter references comply with the rules concerning variable references *Entire variable on page 29*, *Variable element on page 29*, and *Variable component on page 30*.

3.9 Aggregates

An aggregate denotes a composite value, that is an array or record value. Each aggregate member is specified by an expression.

<aggregate> ::= ’[’ <expr> { ’,’ <expr> } ’]’

For example
```
[x, y, 2*x]                 -- pos aggregate
["john", "eric", "lisa"]    -- string array aggregate
[[ 100, 100, 0 ], [ 0, 0, z ]]-- pos array aggregate
[[1, 2, 3], [a, b, c]]      -- num matrix (2*3) aggregate
```

The data type of an aggregate is (must be able to be) determined by the context. The data type of each aggregate member must be equal to the type of the corresponding member of the determined type.

For example
```
VAR pos p1;
    p1 := [1, -100, 12];-- Aggregate type pos - determined by p1
    IF [1,-100,12] = [a,b,b] THEN-- Illegal since the data type of neither of the
                                 aggregates can be determined by the
                                 context
```

3.10 Function calls

A function call initiates the evaluation of a specific function and receives the value returned by the function. Functions can be either predefined or user defined.

<function call> ::= <function> ’(’ [<function argument list>] ’)’
<function> ::= <ident>

The arguments of a function call is used to transfer data to (and possibly from) the called function. Arguments are evaluated from left to right. The data type of an argument must be equal to the type of the corresponding *parameter* (see *5.1 Parameter declarations on page 50*) of the function. An argument may be either *required*, *optional* or *conditional*. Optional arguments may be omitted but the order of the (present) arguments must be the same as the order of the parameters. Two or more parameters may be declared to mutually exclude each other, in which case at most one of them may be present in the argument list. Conditional arguments are used to support smooth propagation of optional arguments through chains of routine calls.

<function argument list> ::=
<first function argument> { <function argument> }
<first function argument> ::=
<required function argument>
| <optional function argument>
| <conditional function argument>

<function argument> ::=
’,’ <required function argument>
| <optional function argument>
| ’,’ <optional function argument>
| <conditional function argument>
| ’,’ <conditional function argument>
<required function argument> ::=
[<ident> ’:=’] <expr>
<optional function argument> ::=
’\’ <ident> [’:=’ <expr>]
<conditional function argument> ::=
’\’ <ident> ’?’ <parameter>

A required argument is separated from a preceding (if any) argument by ",". The parameter name may be included, or left out.

For example polar(3.937, 0.785398) - two required arguments

polar(dist := 3.937, angle := 0.785398) - .. using names

An optional or conditional argument is preceded by ’\’ and the parameter name. The specification of the parameter name is mandatory. Switch (See *5.1 Parameter declarations on page 50*) type arguments are somewhat special; they are used only to signal presence (of an argument). Switch arguments do therefore not include any

argument expression. Switch arguments may be propagated using the conditional syntax.

For example	cosine(45)	- one required argument
	cosine(0.785398\rad)	- .. and one switch (optional)
	dist(pnt:=p2)	- one required argument
	dist(\base:=p1, pnt:=p2)	- .. and one optional

A conditional argument is considered to be "present" if the specified optional parameter (of the calling function) is present (see *5.1 Parameter declarations on page 50*), otherwise it is simply considered to be "omitted". Note that the specified parameter must be optional.

For example

```
distance := dist(\base ? b, p);

.. is interpreted as

distance := dist(\base := b, p);

.. if the optional parameter b is present otherwise as

distance := dist( p);
```

The concept of conditional arguments thus eliminates the need for multiple "versions" of routine calls when dealing with propagation of optional parameters.

For example

```
IF Present(b) THEN
  distance := dist(\base:=b, p);
ELSE
  distance := dist(p);
ENDIF
```

More than one conditional argument may be used to match more than one alternative of mutually excluding parameters (see *5.1 Parameter declarations on page 50*). In that case at most one of them may be "present" (may refer a present optional parameter).

For example

```
The function

FUNC bool check (\switch on | switch off, ...

.. thus may be called as

check(\on ? high \ off ? low, ...

.. if at most one of the optional parameters 'high' and 'low' are
 present.
```

The parameter list (see *5.1 Parameter declarations on page 50*) of a function assigns each parameter an *access mode*. The access mode of a parameter puts restrictions on a corresponding argument and specifies how RAPID transfers the argument. See *5 Routine declarations on page 49* for the full description on routine parameters, access modes, and argument restrictions.

3.11 Operators

The tables below view the available operators divided into four classes:

- Multiplying operators
- Adding operators
- Relational operators
- Logical operators

The tables specify the legal operand types and the result type of each operator. Note that the relational operators = and <> are the only operators valid for arrays. The use of operators in combination with operands of types not equal to (see *2.17 Equal types on page 20*) the types specified below will cause a *type error* (see *1.4 Error classification on page 6*).

Multiplication operators

Operator	Operation	Operand types	Result type
*	multiplication	num * num	num[1)]
*	multiplication	dnum * dnum	dnum[1)]
*	scalar vector multiplication	num * pos or pos * num	pos
*	vector product	pos * pos	pos
*	linking of rotations	orient * orient	orient
/	division	num / num	num
/	division	dnum / dnum	dnum
div	integer division	num[2)] **div** num[2)]	num
div	integer division	dnum[2)] **div** dnum[2)]	dnum
mod	integer modulo; remainder	num[2)] **mod** num[3)]	num
mod	integer modulo; remainder	dnum[2)] **mod d**num[3)]	dnum

1) Preserves integer (exact) representation as long as operands and result are kept within the integer subdomain of the numerical type.
2) Must represent an integer value.
3) Must represent a positive (≥ 0) integer value.

Addition operators

Operator	Operation	Operand types	Result type
+	addition	num + num	num[2)]
+	addition	dnum +d num	dnum[2)]
+	unary plus; keep sign	+num or +pos	same[1)2)]
+	vector addition	pos + pos	pos
+	string concatenation	string + string	string

Operator	Operation	Operand types	Result type
-	subtraction	num - num	num[2)]
-	subtraction	dnum - dnum	dnum[2)]
-	unary minus; change sign	-num or -dnum or -pos	same[1)2)]
-	vector subtraction	pos - pos	pos

1) The result receives the same data type as the operand. If the operand has an alias data type (see *2.15 Alias types on page 17*) the result receives the alias "base" type (num or pos).
2) Preserves integer (exact) representation as long as operands and result are kept within the integer subdomain of the numerical type.

Relational operators

Operator	Operation	Operand types	Result type
<	less than	num < num	bool
<	less than	dnum < dnum	bool
<=	less than or equal to	num <= num	bool
<=	less than or equal to	dnum <= dnum	bool
=	equal to	anytype[1)] = anytype[1)]	bool
>=	greater than or equal to	num >= num	bool
>=	greater than or equal to	dnum >= dnum	bool
>	greater than	num > num	bool
>	greater than	dnum > dnum	bool
<>	not equal to	anytype[1)] <> anytype[1)]	bool

1) Only *value* and *semivalue* data types (see *2.16 Data type value classes on page 18*). Operands must have equal types.

Logical operators

Operator	Operation	Operand types	Result type
and	and	bool **and** bool	bool
xor	exclusive or	bool **xor** bool	bool
or	or	bool **or** bool	bool
not	unary not; negation	**not** bool	bool

Expressions

4 Statements

The concept of using installed routines (and types) to support the specific needs of the IRB application programmer has made it possible to limit the number of RAPID statements to a minimum. The RAPID statements support general programming needs and there are really no IRB specific RAPID statements. Statements may only occur inside a routine definition.

<statement> ::=
<simple statement>
| <compound statement>
| <label>
| <comment>
| **<SMT>**

A statement is either *simple* or *compound*. A compound statement may in turn contain other statements. A *label* is a "no operation" statement that can be used to define named (goto-) positions in a program. The placeholder **<SMT>** can be used to represent a statement.

<simple statement> ::=
<assignment statement>
| <procedure call>
| <goto statement>
| <return statement>
| <raise statement>
| <exit statement>
| <retry statement>
| <trynext statement>
| <connect statement>
<compound statement> ::=
<if statement>
| <compact if statement>
| <for statement>
| <while statement>
| <test statement>

4.1 Statement termination

Compound statements (except for the compact if statement) are terminated by statement specific keywords. Simple statements are terminated by a semicolon (;). Labels are terminated by a colon (:). Comments are terminated by a newline character (see *2.10 Comments on page 11*). Statement terminators are considered to be a part of the statement.

for example**WHILE** index < 100 **DO**

.
! Loop start - newline terminates a comment

```
  next:          - ":" terminates a label
  index := index + 1;- ";" terminates assignment statement
ENDWHILE - "endwhile" terminates the while statement
```

4.2 Statement lists

A sequence of zero or more statements is called a *statement list*. The statements of a statement list are executed in succession unless a goto, return, raise, exit, retry or trynext statement, or the occurrence of an interrupt or error causes the execution to continue at another point.

<statement list> ::= { <statement> }

Both routines and compound statements contain statement lists. There are no specific statement list separators. The beginning and end of a statement list is determined by the context.

for example

```
IF a > b THEN
  pos1 := a * pos2;      - start of statement list
  ...
  ! this is a comment
  pos2 := home;          - end of statement list
ENDIF
```

4.3 Label statement

Labels are "no operation" statements used to define named program positions. The goto statement (see *4.6 Goto statement on page 41*) causes the execution to continue at the position of a label.

<label> ::= <identifier> ':'

for example

```
next:
..
GOTO next;
```

The following scope rules are valid for labels:

- The scope of a label comprises the routine in which it is contained.
- Within its scope a label hides any predefined or user defined object with the same name.
- Two labels declared in the same routine may not have the same name.
- A label may not have the same name as a routine data object declared in the same routine.

4.4 Assignment statement

The assignment statement is used to replace the current value of a variable, persistent or parameter (assignment target) with the value defined by an expression. The assignment target and the expression must have equal types. Note that the assignment target must have value or semivalue data type (see *2.16 Data type value classes on page 18*). The assignment target can be represented by the placeholder **<VAR>**.

<assignment statement> ::= <assignment target> ':=' <expression> ';'
<assignment target> ::=
<variable>
| <persistent>
| <parameter>
| **<VAR>**

for example
count := count +1; - entire variable assignment
home.x := x * sin(30); - component assignment
matrix{i, j} := temp; - array element assignment
posarr{i}.y := x; - array element/component assignment
<VAR> := temp + 5; - placeholder use

4.5 Procedure call

A procedure call initiates the evaluation of a procedure. After the termination of the procedure the evaluation continues with the subsequent statement. Procedures can be either predefined or user defined. The placeholder **<ARG>** may be used to represent an undefined argument.

<procedure call> ::= <procedure> [<procedure argument list>] ';'
<procedure> ::=
<identifier>
| '%' <expression> '%'
<procedure argument list> ::=
<first procedure argument> { <procedure argument> }
<first procedure argument> ::=
<required procedure argument>
| <optional procedure argument>
| <conditional procedure argument>
| **<ARG>**
<procedure argument> ::=
',' <required procedure argument>
| <optional procedure argument>
| ',' <optional procedure argument>
| <conditional procedure argument>
| ',' <conditional procedure argument>
| ',' **<ARG>**
<required procedure argument> ::=
[<identifier> ':='] <expression>

<optional procedure argument> ::=
’\’ <identifier> [’:=’ <expression>]
<conditional procedure argument> ::=
’\’ <identifier> ’?’ (<parameter> | **<VAR>**)

The procedure (name) may either be statically specified by using an identifier (*early binding*) or evaluated during runtime from a (string type) expression (*late binding*). Even though early binding should be considered to be the "normal" procedure call form, late binding sometimes provides very efficient and compact code.

for example

```
! early binding
test product_id
case 1:
  proc1 x, y, z;
case 2:
  proc2 x, y, z;
case 3:
  ...

! same example using late binding
% "proc" + NumToStr(product_id, 0) % x, y, z;
  ...

! same example again using another variant of late binding
VAR string procname {3} := ["proc1", "proc2", "proc3"];
 ...
% procname{product_id} % x, y, z;
  ...
```

Note that late binding is available for procedure calls only, not for function calls.

The general rules concerning the argument list of the procedure call are exactly the same as those of the function call. Please refer to *3.10 Function calls on page 32* and *5 Routine declarations on page 49* for more details.

for example

```
move t1, pos2, mv;                    .. procedure call

move tool := t1, dest := pos2, movedata := mv;.. with names

move \reltool, t1, dest, mv;          .. with switch ’reltool’

move \reltool, t1, dest, mv \speed := 100;.. with optional
                                              speed’

move \reltool, t1, dest, mv \time := 20;.. with optional ’time’
```

Normally the procedure reference is solved (bind) according to the normal scope rules, but late binding provide a way to do a deviation from that rule.

The string expression in the statement %<expression>% is in the normal case a string with the name of a procedure found according to the scope rules, but the string could also have an enclosing description prefix that specify the location of the routine.

"name1:name2" specify the procedure "name2" inside the module "name1" (note that the procedure "name2" could be declared local in that module). ":name2" specify the global procedure "name2" in one of the task modules, this is very useful when a downwards call must be done from the system level (installed built in object).

4.6 Goto statement

The goto statement causes the execution to continue at the position of a label.

<goto statement> ::= **GOTO** <identifier> ';'

A goto statement may not transfer control into a statement list.

for example

```
next:
i := i + 1;
 ..
GOTO next;
```

4.7 Return statement

The return statement terminates the execution of a routine and, if applicable, specifies a return value. A routine may contain an arbitrary number of return statements. A return statement can occur anywhere in the statement list or the error or backward handler of the routine and at any level of a compound statement. The execution of a return statement in the entry (see *9 Task modules on page 77*) routine of a task terminates the evaluation of the task. The execution of a return statement in a trap (see *8.3 Trap routines on page 74*) routine resumes execution at the point of the interrupt.

<return statement> ::= **RETURN** [<expression>] ';'

The expression type must be equal to the type of the function. Return statements in procedures and traps must not include the return expression.

for example

```
FUNC num abs_value (num value)
  IF value < 0 THEN
    RETURN -value;
   ELSE
    RETURN value;
  ENDIF
ENDFUNC

PROC message ( string mess )
  write printer, mess;
  RETURN;              - might have been left out
ENDPROC
```

4.8 Raise statement

The raise statement is used to explicitly raise or propagate an error.

<raise statement> ::= **RAISE** [<error number>] ';'
<error number> ::= <expression>

A raise statement that includes an explicit error number raises an error with that number. The error number (see *7 Error recovery on page 59*) expression must represent an integer value (see *Num type on page 14*) in the range from 1 to 90. A raise statement including an error number must not appear in the error handler of a routine.

A raise statement with no error number may only occur in the error handler of a routine and raises again (reraises) the same (current) error at the point of the call of the routine, that is propagates the error. Since a trap routine can only be called by the system (as a response to an interrupt), any propagation of an error from a trap routine is made to the system error handler (see *7 Error recovery on page 59*).

for example

```
CONST errnum escape := 10;
  ...
  RAISE escape;        - recover from this position
  ...
ERROR
  IF ERRNO = escape THEN
    RETURN val2;
  ENDIF
ENDFUNC
```

4.9 Exit statement

The exit statement is used to immediately terminate the execution of a task.

<exit statement> ::= **EXIT** ';'

Task termination using the exit statement, unlike returning from the entry routine of the task, in addition prohibits any attempt from the system to automatically restart the task.

for example

```
TEST state
CASE ready:
  ...
DEFAULT :
  ! illegal/unknown state - abort
  write console, "Fatal error: illegal state";
  EXIT;
ENDTEST
```

4.10 Retry statement

The retry statement is used to resume execution after an error, starting with (reexecuting) the statement that caused the error.

<retry statement> ::= **RETRY** ';'

The retry statement can only appear in the error handler of a routine.

for example

```
...
! open logfile
open \append, logfile, "temp.log";
...
ERROR
  IF ERRNO = ERR_FILEACC THEN
    ! create missing file
    create "temp.log";
    ! resume execution
    RETRY;
  ENDIF
  ! propagate "unexpected" error
  RAISE;
ENDFUNC
```

4.11 Trynext statement

The trynext statement is used to resume execution after an error, starting with the statement following the statement that caused the error.

<trynext statement> ::= **TRYNEXT** ';'

The trynext statement can only appear in the error handler of a routine.

for example

```
...
! Remove the logfile
delete logfile;
...
ERROR
  IF ERRNO = ERR_FILEACC THEN
    ! Logfile already removed - Ignore
    TRYNEXT;
  ENDIF
  ! propagate "unexpected" error
  RAISE;
ENDFUNC
```

4.12 Connect statement

The connect statement allocates an interrupt number, assigns it to a variable or parameter (connect target) and connects it to a trap routine. When (if) an interrupt with that particular interrupt number later occurs the system responds to it by calling the connected trap routine. The connect target can be represented by the placeholder **<VAR>**.

<connect statement> ::= **CONNECT** <connect target> **WITH** <trap> ';'
<connect target> ::=
<variable>
| <parameter>
| **<VAR>**
<trap> ::= <identifier>

The connect target must have num (or alias for num) type and must be (or represent) a module variable (not a routine variable). If a parameter is used as connect target it must be a VAR or INOUT/VAR parameter - see *5.1 Parameter declarations on page 50*). An allocated interrupt number cannot be "disconnected" or connected with another trap routine. The same connect target may not be associated with the same trap routine more than once. This means that the same connect statement may not be executed more than once and that only one of two identical connect statements (same connect target and same trap routine) may be executed during a session. Note though, that more than one interrupt number may be connected with the same trap routine.

for example

```
PROC main()
  VAR intnum hp;
  ...
  CONNECT hp WITH high_pressure;
  ...
ENDPROC

TRAP high_pressure
  close_valve\fast;
  RETURN;
ENDTRAP
```

4.13 If statement

The if statement evaluates one or none of a number of statement lists, depending on the value of one or more conditional expressions.

<if statement> ::=
IF <conditional expression> **THEN** <statement list>
{**ELSEIF** <conditional expression>
THEN <statement list> | **<EIT>** }
[**ELSE** <statement list>]
ENDIF

The conditional expressions are evaluated in succession until one of them evaluates to true. The corresponding statement list is then executed. If none of them evaluates to true the (optional) else clause is executed. The placeholder **<EIT>** can be used to represent an undefined elseif clause.

for example
```
IF counter > 100 THEN
  counter := 100;
ELSEIF counter < 0 THEN
  counter := 0;
ELSE
  counter := counter + 1;
ENDIF
```

4.14 Compact IF statement

In addition to the general, structured if-statement presented above (*4.13 If statement on page 44*), RAPID provides an alternative, compact if statement. The compact if statement evaluates a single, simple statement if a conditional expression evaluates to true.

<compact if statement> ::=
IF <conditional expression> (<simple statement> | **<SMT>**)

The placeholder **<SMT>** can be used to represent an undefined simple statement.

for example **IF** ERRNO = escape1 **GOTO** next;

4.15 For statement

The for statement repeats the evaluation of a statement list while a loop variable is incremented (or decremented) within a specified range. An optional step clause makes it possible to select the increment (or decrement) value.

The loop variable:

- is declared (with type num) by its appearance.
- has the scope of the statement list (do .. endfor).
- hides any other object with the same name.
- is readonly, that is cannot be updated by the statements of the for loop.

<for statement> ::=
FOR <loop variable> **FROM** <expression>
TO <expression> [**STEP** <expression>]
DO <statement list> **ENDFOR**

<loop variable> ::= <identifier>

Initially the from, to and step expressions are evaluated and their values are kept. They are evaluated only once. The loop variable is initiated with the from value. If no step value is specified it defaults to 1 (or -1 if the range is descending).

Before each new (not the first) loop, the loop variable is updated and the new value is checked against the range. As soon as the value of the loop variable violates (is outside) the range the execution continues with the subsequent statement.

The from, to and step expressions must all have num (numeric) type.

for example
```
FOR i FROM 10 TO 1 STEP -1 DO
  a{i} := b{i};
ENDFOR
```

4.16 While statement

The while statement repeats the evaluation of a statement list as long as the specified conditional expression evaluates to true.

<while statement> ::=
WHILE <conditional expression> **DO**
<statement list> **ENDWHILE**

The conditional expression is evaluated and checked before each new loop. As soon as it evaluates to false the execution continues with the subsequent statement.

for example
```
WHILE a < b DO
  ...
  a := a + 1;
ENDWHILE
```

4.17 Test statement

The test statement evaluates one or none of a number of statement lists, depending on the value of an expression.

<test statement> ::=
TEST <expression>
{ **CASE** <test value> { ’,’ <test value> } ’:’
<statement list>) | **<CSE>** }
[**DEFAULT** ’:’<statement list>]
ENDTEST
<test value> ::= <expression>

Each statement list is preceded by a list of test values, specifying the values for which that particular alternative is to be selected. The test expression can have any value or semivalue data type (see *2.16 Data type value classes on page 18*). The type of a test value must be equal to the type of the test expression. The execution of a test statement will choose one or no alternative. In case more than one test value fits the test expression only the first is recognized. The placeholder **<CSE>** can be used to represent an undefined case clause.

The optional default clause is evaluated if no case clause fits the expression.

for example

```
TEST choice
CASE 1, 2, 3 :
  pick number := choice;
CASE 4 :
  stand_by;
DEFAULT:
  write console, "Illegal choice";
ENDTEST
```

5 Routine declarations

A routine is a named carrier of executable code. A *user* routine is defined by a RAPID routine declaration. A *predefined* routine is supplied by the system and is always available. There are three types of routines - *procedures*, *functions* and *traps*. A function returns a value of a specific type and is used in expression context (see *3.10 Function calls on page 32*). A procedure does not return any value and is used in statement context (see *4.5 Procedure call on page 39*). Trap routines provide a means to respond to interrupts (see *8 Interrupts on page 73*). A trap routine can be associated with a specific interrupt (using the connect statement - see *4.12 Connect statement on page 44*) and is then later automatically executed if that particular interrupt occurs. A trap routine can never be explicitly called from RAPID code. A routine declaration can be represented by the placeholder **<RDN>**.

<routine declaration> ::=
[**LOCAL**] (<procedure declaration>
| <function declaration>
| <trap declaration>)
| <comment>
| **<RDN>**

The declaration of a routine specifies its:

- Name
- Data Type (only valid for functions)
- Parameters (not for traps)
- Data Declarations and Statements (body)
- Backward Handler (only valid for procedures)
- Error Handler
- Undo Handler

Routine declarations may only occur in the last section of a module (see *9 Task modules on page 77*). Routine declarations cannot be nested, that is it is not possible to declare a routine inside a routine declaration.

The optional local directive of a routine declaration classifies a routine to be local, otherwise it is global (see *5.2 Scope rules on page 52*).

5.1 Parameter declarations

The parameter list of a routine declaration specifies the arguments (actual parameters) that must/can be supplied when the routine is called. Parameters are either *required* or *optional*. An optional parameter may be omitted from the argument list of a routine call (see *2.20 Scope rules on page 21*). Two or more optional parameters may be declared to mutually exclude each other, in which case at most one of them may be present in a routine call. An optional parameter is said to be *present* if the routine call supplies a corresponding argument, *not present* otherwise. The value of a not present, optional parameter may not be set or used. The predefined function Present can be used to test the presence of an optional parameter. The placeholders **<PAR>**, **<ALT>**, **<DIM>** can be used to represent undefined parts of a parameter list.

```
<parameter list> ::=
        <first parameter declaration> { <next parameter declaration> }
<first parameter declaration> ::=
        <parameter declaration>
        | <optional parameter declaration>
        | <PAR>
<next parameter declaration> ::=
        ',' <parameter declaration>
         | <optional parameter declaration>
         | ','<optional parameter declaration>
         | ',' <PAR>
<optional parameter declaration> ::=
        '\' ( <parameter declaration> | <ALT> )
            { '|' ( <parameter declaration> | <ALT> ) }
<parameter declaration> ::=
        [ VAR | PERS | INOUT ] <data type>
            <identifier> [ '{' ( '*' { ',' '*' } ) | <DIM> '}' ]
        | 'switch' <identifier>
```

The data type of an argument must be equal to the data type of the corresponding parameter.

Each parameter has an *access mode*. Available access modes are *in* (default), *var*, *pers*, *inout* and *ref* [1]. The access mode specifies how RAPID transfers a corresponding argument to a parameter:

- An *in* parameter is initialized with the value of the argument (expression). The parameter may be used (for example assigned a new value) as an ordinary routine variable.
- A *var*, *pers*, *inout* or *ref* parameter is used as an *alias* for the argument (data object). This means that any update of the parameter is also an update of the argument.

1. RAPID routines cannot have *ref* parameters, only predefined routines can.

The specified access mode of a parameter restricts a corresponding argument as:

X Legal argument
--- Illegal argument

Access mode

Argument	in	var	pers	inout	ref
constant	X				X
readonly variable [a]	X				X
variable	X	X		X	X
persistent	X		X	X	X
parameter:					
in	X	X	---	X	X
var	X	X	---	X	X
pers	X	---	X	X	X
inout:					
var	X	X	--- [b]	X	X
pers	X	--- [b]	X	X	X
any other expression	X	---	---	---	---

a. for example FOR loop variables (see *4.15 For statement on page 45*), errno, intno (see C Appendix:).
b. Execution error (see *1.4 Error classification on page 6*).

for example ..., **INOUT** num temp, ... - inout parameter

The built-in routines ’IsPers’ and ’IsVar’ can be used to test if an *inout* parameter is an alias for a variable or persistent argument.

The special type ’switch’ may (only) be assigned to optional parameters and provides a means to use "switch arguments", that is arguments given only by their names (no values). The domain of the switch type is empty and no value can be transferred to a switch parameter. The only way to use a switch parameter is to check its presence using the predefined function Present, or to pass it as an argument in a routine call.

for example
```
PROC glue ( \switch on | switch off, ...   - switch parameters
    ...
  IF Present(off ) THEN                    - check presence off
    ...                                      optional parameter ’off’
ENDPROC

glue\off, pos2;                            - argument use
```

Arrays may be passed as arguments. The degree of an array argument must comply with the degree of the corresponding parameter. The dimension of an array parameter is "conformant" (marked by ’*’). The actual dimension is later bound by the dimension of the corresponding argument of a routine call. A routine can determine the actual dimension of a parameter using the predefined function ’Dim’.

for example ... , **VAR** num pallet{*,*}, ... - num-matrix parameter

5.2 Scope rules

The *scope* of an object denotes the area in which the name is *visible*.

The scope of a predefined routine comprises any RAPID module.

The following scope rules are valid for user routines:

- The scope of a local user routine comprises the module in which it is contained.
- The scope of a global user routine in addition comprises any other module in the task buffer.
- Within its scope a user routine hides any predefined object with the same name.
- Within its scope a local user routine hides any global module object with the same name.
- Two module objects declared in the same module may not have the same name.
- Two global objects declared in two different modules in the task buffer may not have the same name.
- A global user routine and a module may not share the same name.

The scope rules concerning parameters comply with the scope rules concerning routine variables. Refer to *2.20 Scope rules on page 21* for information on routine variable scope.

Refer to *9 Task modules on page 77* for information on task modules.

5.3 Procedure declarations

A procedure declaration binds an identifier to a procedure definition.

<procedure declaration> ::=
PROC <procedure name>
’(’ [<parameter list>] ’)’
<data declaration list>
<statement list>
[**BACKWARD** <statement list>]
[**ERROR** [<error number list>] <statement list>]
[**UNDO** <statement list>]
ENDPROC

<procedure name> ::= <identifier>
<data declaration list> ::= { <data declaration> }

Note that a data declaration list can include comments (see *2.10 Comments on page 11*).

The evaluation of a procedure is either explicitly terminated by a return statement (see *4.7 Return statement on page 41*) or implicitly terminated by reaching the end (**ENDPROC**, **BACKWARD**, **ERROR** or **UNDO**) of the procedure.

for example Multiply all elements of a num array by a factor:

```
PROC arrmul( VAR num array{*}, num factor)
  FOR index FROM 1 TO Dim( array, 1 ) DO
    array{index} := array{index} * factor;
  ENDFOR
ENDPROC<-- implicit return
```

(the predefined ’Dim’ function returns the dimension of an array)

Procedures which are going to be used in late binding calls are treated as a special case. That is the parameters for the procedures, which are called from the same late binding statement, should be matching as regards optional/required parameters and mode, and should also be of the same basic type. For example if the second parameter of one procedure is required and declared as **VAR** num then the second parameter of other procedures, which are called by the same late binding statement, should have a second parameter which is a required **VAR** with basic type num. The procedures should also have the same number of parameters. If there are mutually exclusive optional parameters, they also have to be matching in the same sense.

5.4 Function declarations

A function declaration binds an identifier to a function definition.

<function declaration> ::=
FUNC <data type>
<function name>
’(’ [<parameter list>] ’)’
<data declaration list>
<statement list>
[**ERROR** [<error number list>] <statement list>]
[**UNDO** <statement list>]
ENDFUNC
<function name> ::= <identifier>

Functions can have (return) any value data type (including any available installed type). A function cannot be dimensioned, that is a function cannot return an array value.

The evaluation of a function must be terminated by a return statement (see *4.7 Return statement on page 41*).

for example — Return the length of a vector:

```
FUNC num veclen(pos vector)
  RETURN sqrt(quad(vector.x) + quad(vector.y) +
quad(vector.z));
ERROR
  IF ERRNO = ERR_OVERFLOW THEN
    RETURN maxnum;
  ENDIF
  ! propagate "unexpected" error
  RAISE;
ENDFUNC
```

5.5 Trap declarations

A trap declaration binds an identifier to a trap definition. A trap routine can be associated with an interrupt (number) by using the connect statement (see *4.12 Connect statement on page 44*). Note that one trap routine may be associated with many (or no) interrupts.

```
<trap declaration> ::=
        TRAP <trap name>
        <data declaration list>
        <statement list>
        [ ERROR [ <error number list> ] <statement list> ]
        [ UNDO <statement list> ]
        ENDTRAP
<trap name> ::= <identifier>
```

The evaluation of the trap routine is explicitly terminated using the return statement (see *4.7 Return statement on page 41*) or implicitly terminated by reaching the end (endtrap, error or undo) of the trap routine. The execution continues at the point of the interrupt.

for example — Respond to low pressure interrupt:

```
TRAP low_pressure
  open_valve\slow;
  ! return to point of interrupt
  RETURN;
ENDTRAP
```

for example — Respond to high pressure interrupt:

```
TRAP high_pressure
  close_valve\fast;
  ! return to point of interrupt
  RETURN;
ENDTRAP
```

6 Backward execution

RAPID supports stepwise, backward execution of statements. Backward execution is very useful for debugging, test and adjustment purposes during RAPID program development. RAPID procedures may contain a backward handler (statement list) that defines the backward execution "behavior" of the procedure (call).

The following general restrictions are valid for Backward execution:

- Only simple (not compound) statements can be executed backwards.
- It is not possible to step backwards out of a routine at the top of it's statement list (and reach the routine call).
- Simple statements have the following backward behavior:
- Procedure calls (predefined or user defined) can have any backward behavior - take some action, do nothing or *reject* [1)] the backward call. The behavior is defined by the procedure definition.
- The arguments of a procedure call being executed backwards are always (even in case of reject) executed and transferred to the parameters of the procedure exactly in the same way as is the case with forward execution. Argument expressions (possibly including function calls) are always executed "forwards".
- Comments, labels, assignment statements and connect statements are executed as "no operation" while all other simple statements rejects[1)] backward execution.
- 1) No support for backward step execution - no step is taken.

6.1 Backward handlers

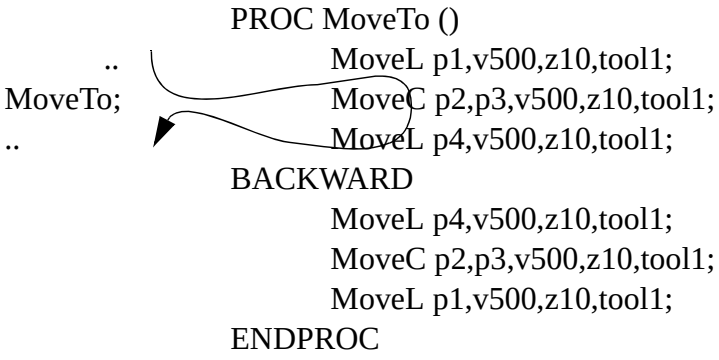
Procedures may contain a backward handler that defines the backward execution of a procedure call.

The backward handler is really a part of the procedure and the scope of any routine data also comprises the backward handler of the procedure.

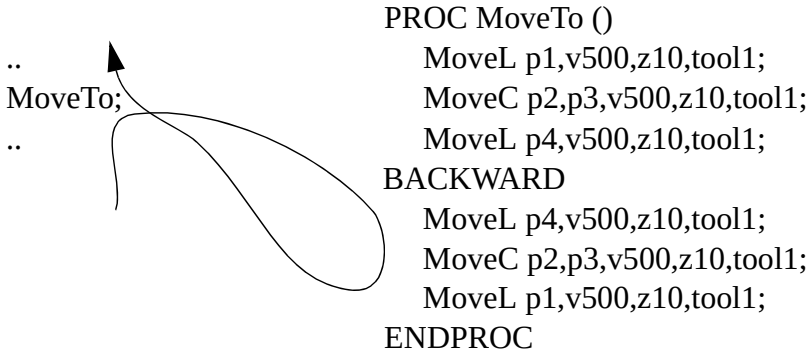
Example:
```
PROC MoveTo ()
    MoveL p1,v500,z10,tool1;
    MoveC p2,p3,v500,z10,tool1;
    MoveL p4,v500,z10,tool1;
BACKWARD
    MoveL p4,v500,z10,tool1;
    MoveC p2,p3,v500,z10,tool1;
    MoveL p1,v500,z10,tool1;
ENDPROC
```

Backward execution

When the procedure is called during forward execution, the following occurs:



When the procedure is called during backwards execution, the following occurs:



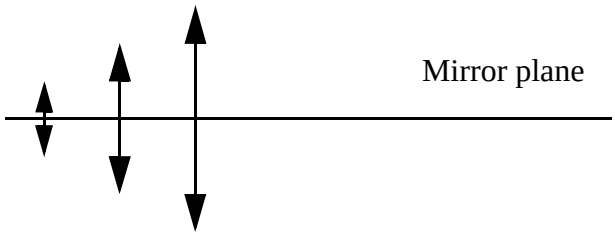
Instructions in the backward or error handler of a routine may not be executed backwards. Backward execution cannot be nested, that is two instructions in a call chain may not simultaneously be executed backwards.

A procedure with no backward handler cannot be executed backwards. A procedure with an empty backward handler is executed as “no operation”.

6.2 Limitation of move instructions in the backward handler

The move instruction type and sequence in the backward handler must be a mirror of the move instruction type and sequence for forward execution in the same routine:

```
PROC MoveTo ()
  MoveL p1,v500,z10,tool1;
  MoveC p2,p3,v500,z10,tool1;
  MoveL p4,v500,z10,tool1;
BACKWARD
  MoveL p4,v500,z10,tool1;
  MoveC p2,p3,v500,z10,tool1;
  MoveL p1,v500,z10,tool1;
ENDPROC
```



Note that the order of CirPoint *p2* and ToPoint *p3* in the MoveC should be the same.

By move instructions is meant all instructions that result in some movement of the robot or external axes such as MoveL, SearchC, TriggJ, ArcC, PaintL ...

Any departures from this programming limitation in the backward handler can result in faulty backward movement. Linear movement can result in circular movement and vice versa, for some part of the backward path.

Backward execution

7 Error recovery

An execution error (see *1.4 Error classification on page 6*) is an abnormal situation, related to the execution of a specific piece of RAPID program code. An error makes further execution impossible (or at least hazardous). "Overflow" and "division by zero" are examples of errors. Errors are identified by their unique *error number* and are always recognized by the system. The occurrence of an error causes suspension of the normal program execution and the control is passed to an *error handler.* The concept of error handlers makes it possible to respond to, and possibly recover from errors that arise during program execution. If further execution is not possible, at least the error handler can assure that the task is given a graceful abortion.

7.1 Error handlers

Any routine may include an error handler. The error handler is really a part of the routine and the scope of any routine data object (variable, constant, parameter) also comprises the error handler of the routine. If an error occurs during the evaluation of the routine the control is transferred to the error handler.

for example

```
FUNC num safediv(num x, num y)
  RETURN x / y;
ERROR
  IF ERRNO = ERR_DIVZERO THEN
    ! return max numeric value
    RETURN max_num;
  ENDIF
ENDFUNC
```

The predefined (readonly) variable *ERRNO* contains the error number of the (most recent) error and can be used by the error handler to identify the error. After necessary actions have been taken the error handler can:

- Resume execution starting with the statement in which the error occurred. This is made using the *RETRY* statement (see *4.10 Retry statement on page 43*).
- Resume execution starting with the statement after the statement in which the error occurred. This is made using the *TRYNEXT* statement (see *4.11 Trynext statement on page 43*).
- Return control to the caller of the routine by using the *RETURN* statement (see *4.7 Return statement on page 41*). If the routine is a function the *RETURN* statement must specify an appropriate return value.
- Propagate the error to the caller of the routine by using the *RAISE* statement (see *4.8 Raise statement on page 42*) - "Since I‘m not familiar with this error it’s up to my caller to deal with it".

If an error occurs in a routine that does not contain an error handler or reaching the end of the error handler (**ENDFUNC, ENDPROC** or **ENDTRAP**), the *system error handler* is called. The system error handler just reports the error and stops the execution.

In a chain of routine calls, each routine may have its own error handler. If an error occurs in a routine with an error handler, and the error is explicitly propagated using the *RAISE* statement, the same error is raised again at the point of the call of the routine - the error is *propagated*. If the top of the call chain (the entry routine of the task) is reached without any error handler found or if reaching the end of any error handler within the call chain, the system error handler is called. The system error handler just reports the error and stops the execution. Since a trap routine can only be called by the system (as a response to an interrupt), any propagation of an error from a trap routine is made to the system error handler.

In addition to errors detected and raised by the system, a program can explicitly raise errors using the *RAISE* statement (see *4.8 Raise statement on page 42*). The facility can be used to recover from complex situations. For example it can be used to escape from deeply nested code positions. Error numbers in the range from 1 to 90 may be used.

for example

```
CONST errnum escape1 := 10;
  ...
  RAISE escape1;
  ...
ERROR
  IF ERRNO = escape1 THEN
    RETURN val2;
  ENDIF
ENDFUNC
```

Note that it is not possible to recover from or respond to errors that occur *within* an error handler or backward handler. Such errors are always propagated to the system error handler.

7.2 Error recovery with long jump

Error recovery with long jump may be used to bypass the normal routine call and return mechanism to handle abnormal or exceptional conditions. To accomplish this, a specific error recovery point must be specified. By using the *RAISE* instruction the long jump will be performed and the execution control is passed to that error recovery point.

Error recovery with long jump is typically used to pass execution control from a deeply nested code position, regardless of execution level, as quickly and simple as possible to a higher level.

Execution levels

An execution level is a specific context that the RAPID program is running in. There are three execution levels in the system, Normal, Trap, and User:

- Normal level: All program are started at this level. This is the lowest level.
- Trap level: Trap routines are executed at this level. This level overrides the Normal level but can be overridden by the User level.
- User level: Event routines and Service routines are executed at this level. This level overrides Normal and Trap level. This level is the highest one and cannot be overridden by any other level.

Error recovery point

The essential thing for error recovery with long jump is the characteristic error recovery point.

The error recovery point is a normal *ERROR* clause but with an extended syntax, a list of error numbers enclosed by a pair of parentheses, see example below:

Example:

```
MODULE example1
    PROC main()
        ! Do something important
        myRoutine;
    ERROR (56, ERR_DIVZERO)
        RETRY;
    ENDPROC
ENDMODULE
```

Syntax

An error recovery point has the following syntax: (EBNF)

[**ERROR** [<error number list>] <statement list>]

<error number list> ::= '(' <error number> { ',' <error number>} ')'
<error number> ::=
<num literal>
| <entire constant>
| <entire variable>
| <entire persistent>

Using error recovery with long jump

```
MODULE example2
    PROC main()
        routine1;
        ! Error recovery point
    ERROR (56)
        RETRY;
    ENDPROC

    PROC routine1()
        routine2;
    ENDPROC

    PROC routine2()
        RAISE 56;
    ERROR
        ! This will propagate the error 56 to main
        RAISE;
    ENDPROC
ENDMODULE
```

The system handles a long jump in following order:

- The raised error number is search, starting at calling routine's error handler and to the top of the current call chain. If there is an error recovery point with the raise error number, at any routine in the chain, the program execution continues in that routine's error handler.
- If no error recovery point is found in the current execution level the searching is continued in the previous execution level until the NORMAL level is reached.
- If no error recovery point is found in any execution level the error will be raised and handled in the calling routine's error handler, if any.

Error recovery through execution level boundaries

It is possible to pass the execution control through the execution level boundaries by using long jump, that is the program execution can jump from a TRAP, USER routine to the Main routine regardless how deep the call chains are in the TRAP, USER, and NORMAL level.
This is useful way to handle abnormal situation that requires the program to continue or start over from good and safely defined position in the program.

When a long jump is done from one execution level to another level there can be an active instructions at that level. Since the long jump is done from one error handler to another, the active instruction will be undone by the system (for example an active MoveX instruction will clear its part of the path).

Remarks

By using the predefined constant LONG_JMP_ALL_ERR it is possible to catch all kinds of errors at the error recovery point.
Observe the following restrictions when using error recovery with long jump:

- Do not assume that the execution mode (cont, cycle, or forward step) is the same at the error recovery point as it was where the error occurred. The execution mode is not inherited at long jump.
- Be careful when using *StorePath*. Always call *RestoPath* before doing a long jump, otherwise the results are unpredictable.
- The numbers of retries are not set to zero at the error recovery point after a long jump.
- Be careful when using TRYNEXT at the error recovery point, the result can be unpredictable if the error occurs in a function call as in the example below:

Example:

```
MODULE Example3
   PROC main
      WHILE myFunction() = TRUE DO
         myRoutine;
      ENDWHILE
      EXIT;
   ERROR (LONG_JMP_ALL_ERR)
      TRYNEXT;
   ENDPROC
ENDMODULE
```

If the error occurs in the function myFunction and the error is caught in the main routine, the instruction TRYNEXT will pass the execution control to the next instruction, in this case EXIT. This is because the WHILE instruction considers to be the one that fails

UNDO handler

When using long jump, one or several procedures may be dropped without executing the end of the routine or the error handler. If no undo handler is used these routine may leave loose ends. In the example below, *routine1* would leave the file log open if the long jump was used and there was no undo handler in *routine1*.

To make sure that each routine cleans up after itself, use an undo handler in any routine that may not finish the execution due to a long jump.

Example:

```
MODULE example4
   PROC main()
      routine1;
      ! Error recovery point
   ERROR (56)
      RETRY;
   ENDPROC
```

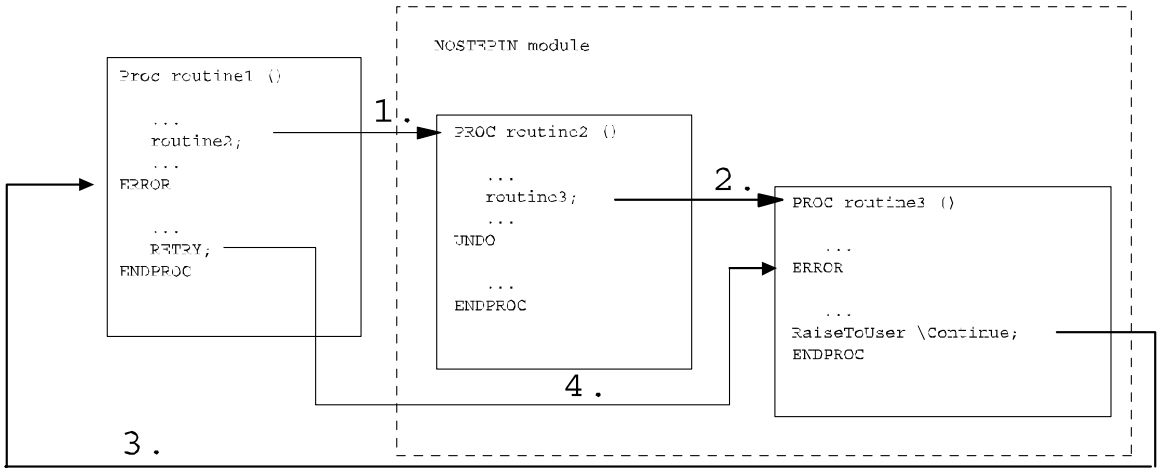
```
    PROC routine1()
      VAR iodev log;
      Open "HOME:" \File:= "FILE1.DOC", log;
      routine2;
      Write log, "routine1 ends with normal execution";
      Close log;
    ERROR
      ! Another error handler
    UNDO
      Close log;
    ENDPROC

    PROC routine2()
      RAISE 56;
    ERROR
      ! This will propagate the error 56 to main
      RAISE;
    ENDPROC
  ENDMODULE
```

7.3 Nostepin routines

The nostepin routines in a nostepin module can call each other in call chains. Using the *RAISE* instruction in the error handler of one of the routines in the call chain will propagate the error one step up in the call chain. In order to raise the error to the user level (outside the nostepin module) with the *RAISE* instruction, every routine in the call chain must have an error handler that raise the error.

By using the *RaiseToUser* instruction, the error can be propagated several steps up in the call chain. The error will then be handled by the error handler in the last routine in the call chain that is not a nostepin routine.

If *RaiseToUser* is called with the argument *\Continue*, the instruction (in the nostepin routine) that caused the error will be remembered. If the error handler that handles the error ends with *RETRY* or *TRYNEXT*, the execution will continue from where the error occurred.



1.	routine2 is called
2.	routine3 is called
3.	the error is raised to user level
4.	execution returns to the instruction in routine3 that caused the error

Note: One or several routines may be dropped without executing the end of the routine or the error handler. In the example this would have been the case for *routine2* if *RaiseToUser* had used the argument *\BreakOff* instead of *\Continue*. To make sure such a routine does not leave any loose ends (such as open files) make sure there is an undo handler that cleans up (for example close files).

Note: If the routine that calls the nostepin routine (*routine1* in the example) is made to a nostepin routine, the error will no longer be handled by its error handler. Changing a routine to a nostepin routine can require the error handler to be moved to the user layer.

7.4 Asynchronously raised errors

About asynchronously raised errors

If a move instruction ends in a corner zone, the next move instruction must be executed before the first move instruction has finished its path. Otherwise the robot would not know how to move in the corner zone. If each move instruction only move a short distance with large corner zones, several move instructions may have to be executed ahead.

An error may occur if something goes wrong during the robot movement. However, if the program execution has continued, it is not obvious which move instruction the robot is carrying out when the error occur. The handling of asynchronously raised errors solve this problem.

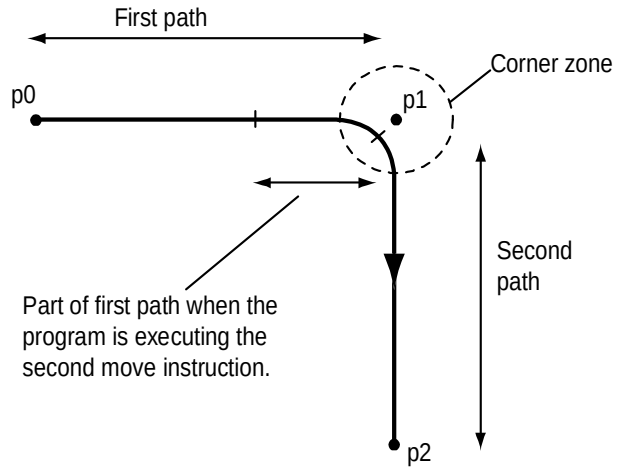
The basic idea is that an asynchronously raised error is connected to a move instruction and is handled by the error handler in the routine that called that instruction.

Two types of asynchronously raised errors

There are two ways of creating asynchronously raised errors, resulting in slightly different behavior:

- *ProcerrRecovery \SyncOrgMoveInst* creates an asynchronous error that is connected to the move instruction which created the current robot path.
- *ProcerrRecovery \SyncLastMoveInst* creates an asynchronous error that is connected to the move instruction that is currently being executed. If no move instruction is being executed this error is connected to the next move instruction that will be executed.

If an error occurs during the first path but when the program is calculating the second path (see illustration below), the behavior depends on the argument of *ProcerrRecovery*. If the error is created with *\SyncOrgMoveInst*, it is connected to the first move instruction (the instruction that created the first path). If the error is created with *\SyncLastMoveInst*, it is connected to the second move instruction (the instruction that created the second path).

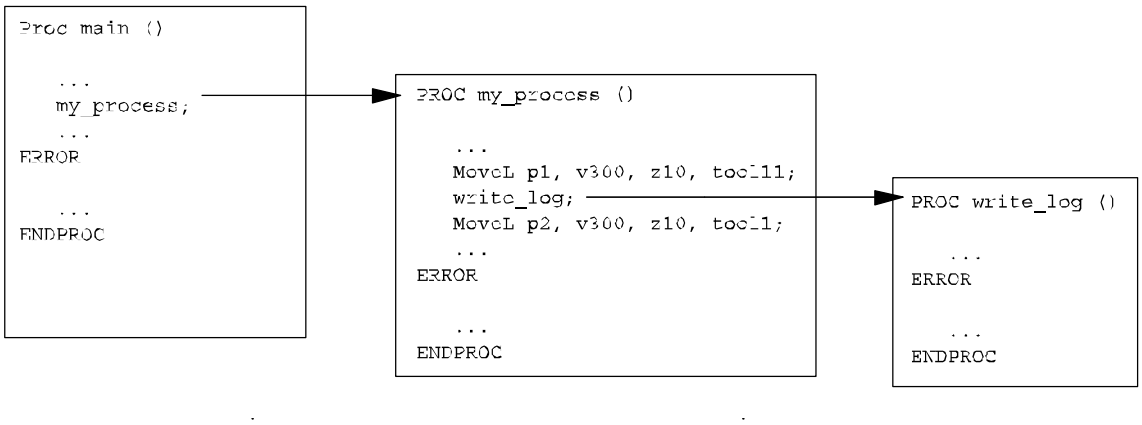


Attempt to handle errors in the routine that called the move instruction

If you create a routine with error handling to take care of process errors that may occur during robot movement, you want these errors to be handled in this routine. If the error is raised when the program pointer is in a subroutine, you do not want the error handler of that subroutine to handle the error.

Asynchronously raised errors are connected to the path that the robot is currently performing. An asynchronously raised error can be handled by the error handler in the routine whose move instruction created the path the robot is carrying out when the error occurs.

In the example shown below, a process error occurs before the robot has reached p1, but the program pointer has already continued to the subroutine *write_log*.



If there was no handling of asynchronously raised errors, an error that was raised when the program pointer was in *write_log* would be handled by the error handler in *write_log*. The handling of asynchronously raised errors will make sure that the error is handled by the error handler in *my_process*.

An asynchronous error created with *ProcerrRecovery \SyncOrgMoveInst* would instantly be handled by the error handler in *my_process*. An asynchronous error created with *ProcerrRecovery \SyncLastMoveInst* would wait for the program pointer to reach the second move instruction in *my_process* before being handled by the error handler in *my_process*.

Note: If a subroutine (*write_log* in the example) were to have move instructions and *\SyncLastMoveInst* is used, the error might be handled by the error handler in the subroutine.

If the error handler in *my_process* ends with ***EXIT***, all program execution is stopped.

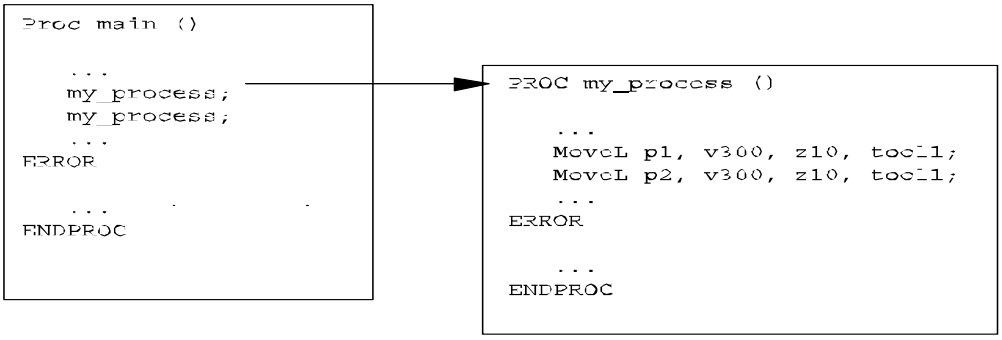
If the error handler in *my_process* ends with ***RAISE***, the error is handled by the error handler in *main*. The routine calls to *my_process* and *write_log* are dropped. If the error handler in *main* ends with *RETRY*, the execution of *my_process* starts over.

If the error handler in *my_process* ends with ***RETRY*** or ***TRYNEXT***, the program execution continues from where the program pointer is (in *write_log*). The error handler should have solved the error situation and called *StartMove* to resume the movement for the instruction that caused the error. Even if the error handler ends with *RETRY*, the first *MoveL* instruction is **not** executed again.

Note: In this case TRYNEXT works the same way as RETRY because the system can be restarted from where the error occurred.

What happens when a routine call is dropped?

When the execution reach the end of a routine, that routine call is dropped. The error handler of that routine call cannot be called if the routine call has been dropped. In the example below, the robot movement will continue after the first *my_process* routine call has been dropped (since the last move instruction has a corner zone).



If the program pointer is in *main* when an error originating from the first *my_process* occurs, it cannot be handled by the error handler in the *my_process* routine call. Where this error is handled will then depend on how the asynchronous error is created:

- If the error is raised with *ProcerrRecovery* *SyncOrgMoveInst*, the error will be handled one step up in the call chain. The error is handled by the error handler in the routine that called the dropped routine call. In the example above, the error handler in *main* would handle the error if the *my_process* routine call has been dropped.
- If the error is raised with *ProcerrRecovery* *SyncLastMoveInst*, the error will be handled by the error handler where the next move instruction is, that is the second routine call to *my_process*. The raising of the error may be delayed until the program pointer reach the next move instruction.

Tip: To make sure asynchronously raised errors are handled in a routine, make sure the last move instruction ends with a stop point (not corner zone) and does not use *Conc*.

Code example

In this example, asynchronously raised errors can be created in the routine *my_process*. The error handler in *my_process* is made to handle these errors.

A process flow is started by setting the signal *do_myproc* to 1. The signal *di_proc_sup* supervise the process, and an asynchronous error is raised if *di_proc_sup* becomes 1. In this simple example, the error is resolved by setting *do_myproc* to 1 again before resuming the movement.

```
MODULE user_module
    VAR intnum proc_sup_int;
    VAR iodev logfile;

    PROC main()
        ...
```

```
    my_process;
    my_process;
    ...
ERROR
...
ENDPROC

PROC my_process()
    my_proc_on;
    MoveL p1, v300, z10, tool1;
    write_log;
    MoveL p2, v300, z10, tool1;
    my_proc_off;
ERROR
    IF ERRNO = ERR_PATH_STOP THEN
        my_proc_on;
        StartMove;
        RETRY;
    ENDIF
ENDPROC

PROC write_log()
    Open "HOME:" \File:= "log.txt", logfile \Append;
    Write logfile "my_process executing";
    Close logfile;
ERROR
    IF ERRNO = ERR_FILEOPEN THEN
        TRYNEXT;
    ENDIF
UNDO
    Close logfile;
ENDPROC

TRAP iprocfail
    my_proc_off;
    ProcerrRecovery \SyncLastMoveInst;
    RETURN;
ENDTRAP

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

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

Error when PP is in write_log

What will happen if a process error occurs when the robot is on its way to p1, but the program pointer is already in the subroutine *write_log*?

The error is raised in the routine that called the move instruction, that is *my_process*, and is handled by its error handler.

Since the *ProcerrRecovery* instruction, in the example, use the switch *\SyncLastMoveInst*, the error will not be raised until the next move instruction is active. Once the second *MoveL* instruction in *my_process* is active, the error is raised and handled in the error handler in *my_process*.

If *ProcerrRecovery* had used the switch *\SyncOrgMoveInst*, the error would have been raised in *my_process* instantly.

Error when execution of my_process has finished

What will happen if a process error occurs when the robot is on its way to p2, but the program pointer has already left the routine *my_process*?

The routine call that caused the error (the first *my_process*) has been dropped and its error handler cannot handle the error. Where this error is raised depends on which switch is used when calling *ProcerrRecovery.*

Since the *ProcerrRecovery* instruction, in the example, use the switch *\SyncLastMoveInst*, the error will not be raised until the next move instruction is active. Once a move instruction is active in the second *my_process* routine call, the error is raised and handled in the error handler in *my_process*.

If *ProcerrRecovery* had used the switch *\SyncOrgMoveInst*, the error would have been raised in *main*. The way *\SyncOrgMoveInst* works is that if the routine call that caused the error (*my_process*) has been dropped, the routine that called that routine (*main*) will raise the error.

Note: If there had been a move instruction between the *my_process* calls in *main*, and *\SyncLastMoveInst* was used, the error would be handled by the error handler in *main*. If another routine with move instructions had been called between the *my_process* calls, the error would have been handled in that routine. This shows that when using *\SyncLastMoveInst* you must have some control over which is the next move instruction.

Nostepin move instructions and asynchronously raised errors

When creating a customized nostepin move instruction with a process, it is recommended to use *ProcerrRecovery \SyncLastMoveInst*. This way, all asynchronously raised errors can be handled by the nostepin instruction.

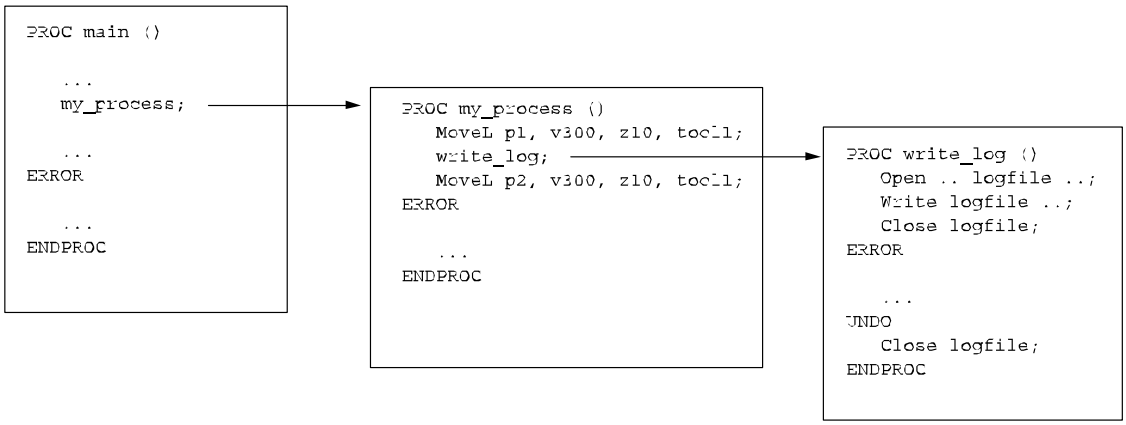
This requires that the user only use this type of move instruction during the entire movement sequence. The movement sequence must begin and end in stop points. Only if two instructions have identical error handlers can they be used in the same movement sequence. This means that one linear move instruction and one circular, using the same process and the same error handler, can be used in the same movement sequence.

If an error should be raised to the user, use RaiseToUser \Continue. After the error has been resolved, the execution can then continue from where the error occurred.

UNDO handler

The execution of a routine can be abruptly ended without running the error handler in that routine. This means that the routine will not clean up after itself.

In the example below, we assume that an asynchronously raised error occurs while the robot is on its way to p1 but the program pointer is at the *Write* instruction in *write_log*. If there was no undo handler, the file *logfile* would not be closed.



This problem can be solved by using undo handlers in all routines that can be interrupted by an asynchronously raised error. It is in the nature of asynchronously raised errors that it is difficult to know where the program pointer will be when they occur. Therefore, when using asynchronously raised errors, use undo handlers whenever clean up may be necessary.

7.5 SkipWarn

An error that is handled in an error handler still generates a warning in the event log. If, for some reason, you do not want any warning to be written to the event log, the instruction *SkipWarn* can be used.

Example:
In the following example code, a routine tries to write to a file that other robot systems also have access to. If the file is busy, the routine waits 0.1 seconds and tries again. If *SkipWarn* was not used, the log file would write a warning for every attempt, even though these warnings are totally unnecessary. By adding the *SkipWarn* instruction, the operator may not notice that the file was busy at the first attempt.

Note that the maximum number of retries is determined by the parameter *No Of Retry.* To make more than 4 retries, you must configure this parameter.

```
PROC routine1()
    VAR iodev report;
    Open "HOME:" \File:= "FILE1.DOC", report;
    Write report, "No parts from Rob1="\Num:=reg1;
    Close report;
ERROR
    IF ERRNO = ERR_FILEOPEN THEN
        WaitTime 0.1;
        SkipWarn;
        RETRY;
    ENDIF
ENDPROC
```

8 Interrupts

Interrupts are program defined events identified by *interrupt numbers*. An interrupt occurs as a consequence of an *interrupt condition* turning true. Unlike errors, the occurrence of an interrupt is not directly related to (synchronous with) a specific code position. The occurrence of an interrupt causes suspension of the normal program execution and the control is passed to a *trap routine*. Interrupt numbers are allocated and connected (associated) with a trap routine using the connect statement (see *4.12 Connect statement on page 44*). Interrupt conditions are defined and manipulated using predefined routines. A task may define an arbitrary number of interrupts.

8.1 Interrupt recognition and response

Even though the system recognizes the occurrence of an interrupt immediately, the response in the form of calling the corresponding trap routine can only take place at specific program positions, namely:

- at the entry of next (after interrupt recognition) statement (of any type).
- after the last statement of a statement list.
- any time during the execution of a waiting routine (for example WaitTime).

This means that, after the recognition of an interrupt, the normal program execution always continues until one of these positions are reached. This normally results in a delay of 2-30 ms between interrupt recognition and response, depending on what type of movement is being performed at the time of the interrupt.

8.2 Editing interrupts

Interrupt numbers are used to identify interrupts/interrupt conditions. Interrupt numbers are not just "any" numbers. They are "owned" by the system and must be allocated and connected with a trap routine using the connect statement (see *4.12 Connect statement on page 44*) before they may be used to identify interrupts.

for example
```
VAR intnum full;
...

CONNECT full WITH ftrap;
```

Interrupts are defined and edited using predefined routines. The *definition* of an interrupt specifies an interrupt condition and associates it with an interrupt number.

for example
```
! define feeder interrupts
ISignalDI sig3, high, full;
```

An interrupt condition must be *active* to be watched by the system. Normally the definition routine (for example ISignalDI) activates the interrupt but that is not always the case. An active interrupt may in turn be *deactivated* again (and vice versa).

for example
```
! deactivate empty
ISleep empty;

! activate empty again
IWatch empty;
```

The *deletion* of an interrupt deallocates the interrupt number and removes the interrupt condition. It is not necessary to explicitly delete interrupts. Interrupts are automatically deleted when the evaluation of a task is terminated.

for example
```
! delete empty
IDelete empty;
```

The raising of interrupts may be *disabled* and *enabled*. If interrupts are disabled any interrupt that occurs is queued and raised first when interrupts are enabled again. Note that the interrupt queue may contain more than one waiting interrupt. Queued interrupts are raised in *fifo* order. Interrupts are always disabled during the evaluation of a trap routine (see *8.3 Trap routines on page 74*).

for example
```
! enable interrupts
IEnable;

! disable interrupts
IDisable;
```

8.3 Trap routines

Trap routines provide a means to respond to interrupts. A trap routine is connected with a particular interrupt number using the connect statement (see *4.12 Connect statement on page 44*). If an interrupt occurs, the control is immediately (see *8.1 Interrupt recognition and response on page 73*) transferred to it's connected trap routine.

for example
```
LOCAL VAR intnum empty;
LOCAL VAR intnum full;

PROC main()
  ...
  ! Connect feeder interrupts
  CONNECT empty WITH ftrap;
  CONNECT full WITH ftrap;
  ! define feeder interrupts
  ISignalDI sig1, high, empty;
  ISignalDI sig3, high, full;
  ...
ENDPROC
```

```
TRAP ftrap
  TEST INTNO
  CASE empty:
    open_valve
  CASE full:
    close_valve;
  ENDTEST
  RETURN;
ENDTRAP
```

More than one interrupt may be connected with the same trap routine. The predefined (readonly) variable *INTNO* contains the interrupt number and can be used by a trap routine to identify the interrupt. After necessary actions have been taken a trap routine can be terminated using the return statement (see *4.7 Return statement on page 41*) or by reaching the end (endtrap or error) of the trap routine. The execution continues at the point of the interrupt. Note that interrupts are always disabled (see *8.2 Editing interrupts on page 73*) during the evaluation of a trap routine.

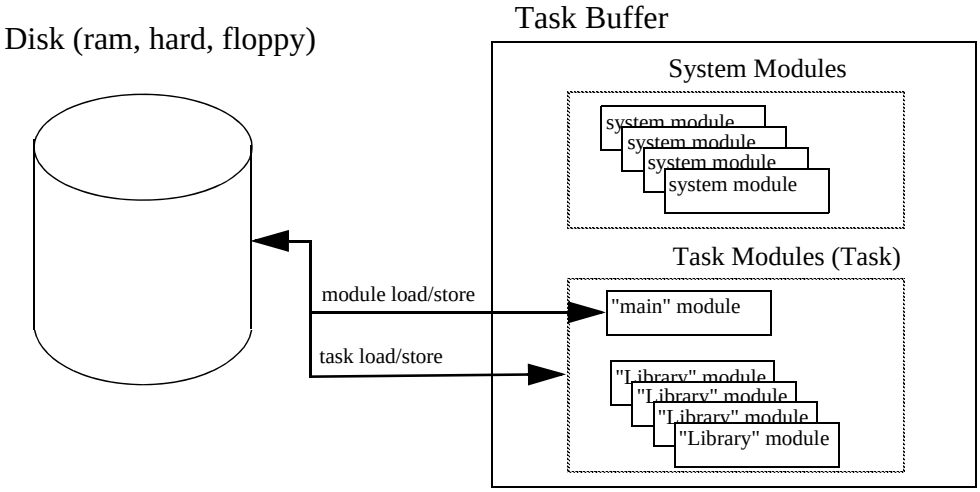
Since a trap routine can only be called by the system (as a response to an interrupt), any propagation of an error from a trap routine is made to the system error handler (see *7 Error recovery on page 59*).

9 Task modules

A RAPID application is called a *task*. A task is composed of a set of *modules*. A module contains a set of type definitions, data and routine declarations. The *task buffer* is used to host modules currently in use (execution, development) on a system. RAPID program code in the task buffer may be loaded/stored from/to file oriented external devices (normally disk files) either as separate modules or as a group of modules - a *Task*.

RAPID distinguishes between *task modules* and *system modules*. A task module is considered to be a part of the task/application while a system module is considered to be a part of the “system”. System modules are automatically loaded to the task buffer during system start and are aimed to (pre)define common, system specific data objects (tools, weld data, move data ..), interfaces (printer, logfile ..) etc. System modules are not included when a task is saved on a file. This means that any update made to a system module will have impact on all existing (old) tasks currently in, or later loaded to the task buffer. In any other sense there is no difference between task and system modules; they can have any content.

While small applications usually are contained in a single task module (besides the system module/s), larger applications may have a "main" task module that in turn references routines and/or data contained in one or more other, "library" task modules.



A "library" module may for example define the interface of a physical or logical object (gripper, feeder, counter etc.) or contain geometry data generated from a CAD-system or created on-line by digitizing (teach in).

One task module contains the *entry* procedure of the task. *Running* the task really means that the entry routine is executed. Entry routines cannot have parameters.

9.1 Module declarations

A module declaration specifies the name, *attributes* and body of a module. A module name hides any predefined object with the same name. Two different modules may not share the same name. A module and a global module object (type, data object or routine) may not share the same name. Module attributes provide a means to modify some aspects of the systems treatment of a module when it is loaded to the task buffer. The body of a module declaration contains a sequence of data declarations followed by a sequence of routine declarations.

<module declaration> ::=
MODULE <module name> [<module attribute list>]
<type definition list>
<data declaration list>
<routine declaration list>
ENDMODULE
<module name> ::= <identifier>
<module attribute list> ::= ’(’ <module attribute> { ’,’ <module attribute> } ’)’
<module attribute> ::=
SYSMODULE
| **NOVIEW**
| **NOSTEPIN**
| **VIEWONLY**
| **READONLY**
<routine declaration list> ::= { <routine declaration> }

<type definition list> ::= { <type definition> }

<data declaration list> ::= { <data declaration> }

The module attributes have the following meaning:

attribute	if specified, the module ..
SYSMODULE	.. is a system module, otherwise a task module
NOVIEW	.. (it’s source code) cannot be viewed (only executed)
NOSTEPIN	.. cannot be entered during stepwise execution
VIEWONLY	.. cannot be modified.
READONLY	.. cannot be modified, but the attribute can be removed.

An attribute may not be specified more than once. If present, attributes must be specified in table order (see above). The specification of **noview** excludes **nostepin**, **viewonly** and **readonly** (and vice versa)**.** The specification of **viewonly** excludes **readonly** (and vice versa).

For example task:

The following three modules could represent a (very simple)

```
MODULE prog1(SYSMODULE, VIEWONLY)
  PROC main()
    ! init weldlib
    initweld;
    FOR i FROM 1 TO Dim(posearr,1) DO
      slow posearr{i};
    ENDFOR
  ENDPROC
  PROC slow(pose p)
    arcweld p \speed := 25;
  ENDPROC
ENDMODULE
```

```
MODULE weldlib
  LOCAL VAR welddata w1 := sysw1;
  ! weldlib init procedure
  PROC initweld()
    ! override speed
    w1.speed := 100;
  ENDPROC
  PROC arcweld(pose position \ num speed | num time)
    ...
  ENDPROC
ENDMODULE
```

```
MODULE weldpath                    - (CAD) generated module
  CONST pose posearr{768} := [
                        [[234.7, 1136.7, 10.2], [1, 0, 0, 0]],
                        ...
                        [[77.2, 68.1, 554.7], [1, 0, 0, 0]]
                        ];
ENDMODULE
```

9.2 System modules

System modules are used to (pre)define system specific data objects (tools, weld data, move data ..), interfaces (printer, logfile ..) etc. Normally, system modules are automatically loaded to the task buffer during system start.

For example

```
MODULE sysun1(SYSMODULE)
  ! Provide predefined variables
  VAR num n1 := 0;
  VAR num n2 := 0;
  VAR num n3 := 0;
  VAR pos p1 := [0, 0, 0];
  VAR pos p2 := [0, 0, 0]; ...
  ! Define channels - open in init function
  VAR channel printer;
  VAR channel logfile; ...
  ! Define standard tools
  PERS pose bmtool := [...
  ! Define basic weld data records
  PERS wdrec wd1 := [ ...
  ! Define basic move data records
  PERS mvrec mv1 := [ ...
  ! Define home position - Sync. Pos. 3
  PERS robtarget home := [ ...
  ! Init procedure
  LOCAL PROC init()
    Open\write, printer, "/dev/lpr";
    Open\write, logfile, "/usr/pm2/log1"... ;
  ENDPROC
ENDMODULE
```

The selection of system module/s is a part of system configuration.

10 Syntax summary

Each rule or group of rules are prefixed by a reference to the section where the rule is introduced.

2.1:

<character> ::= -- *ISO 8859-1 (Latin-1)*--

<newline> ::= -- *newline control character* --

<tab> ::= -- *tab control character* --

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<hex digit> ::= <digit> | A | B | C | D | E | F | a | b | c | d | e | f

<octal digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

<binary digit> ::= 0 | 1

<letter> ::=
<upper case letter>
| <lower case letters>

<upper case letter> ::=
A | B | C | D | E | F | G | H | I | J
| K | L | M | N | O | P | Q | R | S | T
| U | V | W | X | Y | Z | À | Á | Â | Ã
| Ä | Å | Æ | Ç | È | É | Ê | Ë | Ì | Í
| Î | Ï | [1)] | Ñ | Ò | Ó | Ô | Õ | Ö | Ø
| Ù | Ú | Û | Ü | [2)] | [3)] | ß

<lower case letter> ::=
a | b | c | d | e | f | g | h | i | j
| k | l | m | n | o | p | q | r | s | t
| u | v | w | x | y | z | ß | à | á | â | ã
| ä | å | æ | ç | è | é | ê | ë | ì | í
| î | ï | [1)] | ñ | ò | ó | ô | õ | ö | ø
| ù | ú | û | ü | [2)] | [3)] | ÿ

1) Icelandic letter eth.
2) Letter Y with acute accent.
3) Icelandic letter thorn

2.3:

<identifier> ::=
<ident>
| **<ID>**

<ident> ::= <letter> {<letter> | <digit> | '_'}

2.5: <num literal> ::=
<integer> [<exponent>]
| <decimal integer>) [<exponent>]
| <hex integer>
| <octal integer>
| <binary integer>
| <integer> ’.’ [<integer>] [<exponent>]
| [<integer>] ’.’ <integer> [<exponent>]

<integer> ::= <digit> {<digit>
<decimal integer> ::= '0' ('D' | 'd') <integer>
<hex integer> ::= '0' ('X' | 'x') <hex digit> {<hex digit>}
<octal integer> ::= '0' ('O' | 'o') <octal digit> {<octal digit>}
<binary integer> ::= '0' ('B' | 'b') <binary digit> {<binary digit>}

<exponent> ::= (’E’ | ’e’) [’+’ | ’-’] <integer>

2.6: <bool literal> ::= **TRUE** | **FALSE**

2.7 <string literal> ::= ’"’ { <character> | <character code> } ’"’

<character code> ::= ’\’ <hex digit> <hex digit>

2.10: <comment> ::= ’!’ { <character> | <tab> } <newline>

2.11: <type definition> ::=
[**LOCAL**] (<record definition>
| <alias definition>)
| <comment>
| <DN>

<record definition> ::=
RECORD <identifier>
<record component list>
ENDRECORD

<record component list> ::=
<record component definition> |
<record component definition> <record component list>

<record component definition> ::=
<data type> <record component name> ’;’

<alias definition> ::=
ALIAS <data type> <identifier> ’;’

<data type> ::= <identifier>

2.18: <data declaration> ::=
[**LOCAL**] (<variable declaration>
| <persistent declaration>
| <constant declaration>)
| **TASK** (<variable declaration>
| <persistent declaration>
| <comment>
| **<DDN>**

2.22: <variable declaration> ::=
VAR <data type> <variable definition> ';'

<variable definition> ::=
<identifier> ['{' <dim> { ',' <dim> } '}']
[':=' <constant expression>]

<dim> ::= <constant expression>

2.23: <persistent declaration> ::=
PERS <data type> <persistent definition> ';'

<persistent definition> ::=
<identifier> ['{' <dim> { ',' <dim> } '}']
[':=' <literal expression>]

Note! The literal expression may only be omitted for system global persistents.

2.24: <constant declaration> ::=
CONST <data type> <constant definition> ';'

<constant definition> ::=
<identifier> ['{' <dim> { ',' <dim> } '}']
':=' <constant expression>

<dim> ::= <constant expression>

3: <expression> ::=
<expr>
| **<EXP>**

<expr> ::= [**NOT**] <logical term> { (**OR** | **XOR**) <logical term> }

<logical term> ::= <relation> { **AND** <relation> }

<relation> ::= <simple expr> [<relop> <simple expr>]

<simple expr> ::= [<addop>] <term> { <addop> <term> }

<term> ::= <primary> { <mulop> <primary> }

<primary> ::=
<literal>
| <variable>
| <persistent>
| <constant>
| <parameter>
| <function call>
| <aggregate>
| ’(’ <expr> ’)’

<relop> ::= ’<’ | ’<=’ | ’=’ | ’>’ | ’>=’ | ’<>’

<addop> ::= ’+’ | ’-’

<mulop> ::= ’*’ | ’/’ | **DIV** | **MOD**

3.1: <constant expression> ::= <expression>

3.1 <literal expression> ::= <expression>

3.1: <conditional expression> ::= <expression>

3.4: <literal> ::= <num literal>
| <string literal>
| <bool literal>

3.5: <variable> ::=
| <entire variable>
| <variable element>
| <variable component>

<entire variable> ::= <ident>

<variable element> ::= <entire variable> ’{’ <index list> ’}’

<index list> ::= <expr> { ’,’ <expr> }

<variable component> ::= <variable> ’.’ <component name>

<component name> ::= <ident>

3.6 <persistent> ::=
<entire persistent>
| <persistent element>
| <persistent component>

3.6 <constant> ::=
<entire constant>
| <constant element>
| <constant component>

3.8 <parameter> ::=
<entire parameter>
| <parameter element>
| <parameter component>

3.9: <aggregate> ::= ’[’ <expr> { ’,’ <expr> } ’]’

3.10: <function call> ::= <function> ’(’ [<function argument list>] ’)’

<function> ::= <identifier>

<function argument list> ::=
<first function argument> { <function argument>

<first function argument> ::=
<required function argument>
| <optional function argument>
| <conditional function argument>

<function argument> ::=
’,’ <required function argument>
| <optional function argument>
| ’,’ <optional function argument>
| <conditional function argument>
| ’,’ <conditional function argument>

<required function argument> ::= [<ident> ’:=’] <expr>

<optional function argument> ::= ’\’ <ident> [’:=’ <expr>]

<conditional function argument> ::=’\’ <ident> ’?’ <parameter>

4: <statement> ::=
<simple statement>
| <compound statement>
| <label>
| <comment>
| **<SMT>**

<simple statement> ::=
<assignment statement>
| <procedure call>
| <goto statement>
| <return statement>
| <raise statement>
| <exit statement>
| <retry statement>
| <trynext statement>
| <connect statement>

<compound statement> ::=
<if statement>
| <compact if statement>
| <for statement>
| <while statement>
| <test statement>

4.2: <statement list> ::= { <statement> }

4.3: <label> ::= <identifier> ':'

4.4: <assignment statement> ::= <assignment target> ':=' <expression> ';'

<assignment target> ::=
<variable>
| <persistent>
| <parameter>
| **<VAR>**

4.5: <procedure call> ::= <procedure> [<procedure argument list>] ';'

<procedure> ::=
<identifier>
| '%' <expression> '%'

<procedure argument list> ::=
<first procedure argument> { <procedure argument> }

<first procedure argument> ::=
<required procedure argument>
| <optional procedure argument>
| <conditional procedure argument>
| **<ARG>**

<procedure argument> ::=
’,’ <required procedure argument>
| <optional procedure argument>
| ’,’ <optional procedure argument>
| <conditional procedure argument>
| ’,’ <conditional procedure argument>
| ’,’ **<ARG>**

<required procedure argument> ::= [<identifier> ’:=’]
<expression>

<optional procedure argument> ::= ’\’ <identifier> [’:=’
<expression>]

<conditional procedure argument> ::=
’\’ <identifier> ’?’ (<parameter> | **<VAR>**)

4.6: <goto statement> ::= **GOTO** <identifier> ’;’

4.7: <return statement> ::= **RETURN** [<expression>] ’;’

4.8: <raise statement> ::= **RAISE** [<error number>] ’;’

<error number> ::= <expression>

4.9 <exit statement> ::= **EXIT** ’;’

4.10 <retry statement> ::= **RETRY** ’;’

4.11 <trynext statement> ::= **TRYNEXT** ’;’

4.12 <connect statement> ::= **CONNECT** <connect target> **WITH** <trap> ’;’

<connect target> ::=
<variable>
| <parameter>
| **<VAR>**

<trap> ::= <identifier>

4.13: <if statement> ::=
IF <conditional expression> **THEN** <statement list>
{ **ELSEIF** <conditional expression>
THEN <statement list> | **<EIT>** }
[**ELSE** <statement list>]
ENDIF

4.13: <compact if statement> ::=
IF <conditional expression> (<simple statement>
| **<SMT>**)

4.15: <for statement> ::=
FOR <loop variable> **FROM** <expression>
TO <expression> [**STEP** <expression>]
DO <statement list> **ENDFOR**

<loop variable> ::= <identifier>

4.16: <while statement> ::=
WHILE <conditional expression> **DO**
<statement list> **ENDWHILE**

4.17: <test statement> ::=
TEST <expression>
{ **CASE** <test value> { ’,’ <test value> } ’:’
<statement list>) | **<CSE>** }
[**DEFAULT** ’:’<statement list>]
ENDTEST

<test value> ::= <constant expression>

5: <routine declaration> ::=
[**LOCAL**] (<procedure declaration>
| <function declaration>
| <trap declaration>)
| <comment>
| **<RDN>**

5.1 <parameter list> ::=
<first parameter declaration> { <next parameter declaration> }

<first parameter declaration> ::=
<parameter declaration>
| <optional parameter declaration>
| **<PAR>**

<next parameter declaration> ::=
’,’ <parameter declaration>
| <optional parameter declaration>
| ’,’ <optional parameter declaration>
| ’,’ **<PAR>**

<optional parameter declaration> ::=
’\’ (<parameter declaration> | **<ALT>**)
{ ’|’ (<parameter declaration> |
<ALT>) }

<parameter declaration> ::=
[**VAR** | **PERS** | **INOUT**] <data type>
<identifier> [’{’ (’*’ { ’,’ ’*’ }) |
<DIM> ’}’]
| ’switch’ <identifier>

5.3 <procedure declaration> ::=
PROC <procedure name>
’(’ [<parameter list>] ’)’
<data declaration list>
<statement list>
[**BACKWARD** <statement list>]
[**ERROR** [<error number list>] <statement list>]
[**UNDO** <statement list>]
ENDPROC

<procedure name> ::= <identifier>

<data declaration list> ::= { <data declaration> }

5.4: <function declaration> ::=
FUNC <data type>
<function name>
’(’ [<parameter list>] ’)’
<data declaration list>
<statement list>
[**ERROR** [<error number list>] <statement list>]
[**UNDO** <statement list>]
ENDFUNC

<function name> ::= <identifier>

5.5 <trap declaration> ::=
TRAP <trap name>
<data declaration list>
<statement list>
[**ERROR** [<error number list>] <statement list>]
[**UNDO** <statement list>]
ENDTRAP

<trap name> ::= <identifier>

<error number list> ::= '(' <error number> { ',' <error number>} ')'

<error number> ::=
<num literal>
| <entire constant>
| <entire variable>
| <entire persistent>

9.1: <module declaration> ::=
MODULE <module name> [<module attriutelist>]
<type definition list>
<data declaration list>
<routine declaration list>
ENDMODULE

<module name> ::= <identifier>

<module attribute list> ::=
’(’ <module attribute> { ’,’ <module attribute> } ’)’

<module attribute> ::=
SYSMODULE
| **NOVIEW**
| **NOSTEPIN**
| **VIEWONLY**
| **READONLY**

<type definition list> ::= { <type definition> }

<routine declaration list> ::= { <routine declaration> }

11 Built-in routines

Dim

The dim function is used to get the size of an array (datobj). It returns the number of array elements of the specified dimension.

FUNC num Dim (***REF***[1] *anytype*[2] datobj, num dimno)

legal dimno values : 1 -> select first array dimension
2 -> select second array dimension
3 -> select third array dimension

Present

The present function is used to test if the argument (datobj) is present (see 5.1). It returns **FALSE** if datobj is a not present optional parameter, **TRUE** otherwise.

FUNC bool Present (***REF***[1.] *anytype*[2.] datobj)

Break

The break (breakpoint) procedure causes a temporary stop of program execution. Break is used for RAPID program code debugging purposes.

PROC Break ()

IWatch

The iwatch procedure activates the specified interrupt (ino). The interrupt can later be deactivated again using the isleep procedure.

PROC IWatch (**VAR** intnum ino)

ISleep

The isleep procedure deactivates the specified interrupt (ino). The interrupt can later be activated again using the iwatch procedure.

PROC ISleep (**VAR** intnum ino)

IsPers

The ispers function is used to test if a data object (datobj) is (or is an alias for) a persistent (see 5.1). It returns **TRUE** in that case, **FALSE** otherwise.

FUNC bool IsPers (**INOUT** *anytype*[2.] datobj)

IsVar

1. Refer to 3.10 for more information on the *ref* access mode. Note that RAPID routines cannot have REF parameters.
2. The argument can have any data type. Note that *anytype* is just a marker for this property and should not be confused with a “real” data type. Also note that RAPID routines cannot be given *anytype* parameters.

The isvar function is used to test if a data object (datobj) is (or is an alias for) a variable (see 5.1). It returns **TRUE** in that case, **FALSE** otherwise.

FUNC bool IsVar (**INOUT** *anytype*[1] datobj)

1. The argument can have any data type. Note that *anytype* is just a marker for this property and should not be confused with a “real” data type. Also note that RAPID routines cannot be given *anytype* parameters.

12 Built-in data objects

Table 1:

Object Name	Object Kind	Data Type	Description
ERRNO	variable*)	errnum	most recent error number
INTNO	variable*)	intnum	most recent interrupt
ERR_ALRDYCNT	constant	errnum	"variable and trap routine already connected"
ERR_ARGDUPCND	constant	errnum	"duplicated present conditional argument"
ERR_ARGNOTPER	constant	errnum	"argument is not a persistent reference"
ERR_ARGNOTVAR	constant	errnum	"argument is not a variable reference"
ERR_CALLPROC	constant	errnum	"procedure call error (syntax, not procedure) at run time (late binding)"
ERR_CNTNOTVAR	constant	errnum	"CONNECT target is not a variable reference"
ERR_DIVZERO	constant	errnum	"division by zero"
ERR_EXECPHR	constant	errnum	"cannot execute placeholder"
ERR_FNCNORET	constant	errnum	"missing return value"
ERR_ILLDIM	constant	errnum	"array dimension out of range"
ERR_ILLQUAT	constant	errnum	"illegal orientation value"
ERR_ILLRAISE	constant	errnum	"error number in RAISE out of range"
ERR_INOMAX	constant	errnum	"no more interrupt number available"
ERR_MAXINTVAL	constant	errnum	"integer value too large"
ERR_NEGARG	constant	errnum	"negative argument not allowed"
ERR_NOTARR	constant	errnum	"data object is not an array"
ERR_NOTEQDIM	constant	errnum	"mixed array dimensions"
ERR_NOTINTVAL	constant	errnum	"not integer value"
ERR_NOTPRES	constant	errnum	"parameter not present"
ERR_OUTOFBND	constant	errnum	"array index out of bounds"
ERR_REFUNKDAT	constant	errnum	"reference to unknown entire data object"
ERR_REFUNKFUN	constant	errnum	"reference to unknown function"
ERR_REFUNKPRC	constant	errnum	"reference to unknown procedure at linking time or at run time (late binding)"
ERR_REFUNKTRP	constant	errnum	"reference to unknown trap"
ERR_STRTOOLNG	constant	errnum	"string too long"
ERR_UNKINO	constant	errnum	"unknown interrupt number"

*) Read only - can only be updated by the system - not by a RAPID program.

13 Built in objects

There are three groups of “Built in objects”: *Language kernel reserved objects, Installed objects* and *User installed objects*.

- *Language kernel reserved objects* are part of the system and cannot be removed (or leaved out in the configuration). Objects in this group are the instruction *Present*, the variables *intno*, *errno*, and much more. The set of objects in this group is the same for all tasks (multitasking) and installations.
- Most of the *Installed objects* are installed at the first system start (or each P-start) by the internal system configuration and cant be removed (for example the instructions *MoveL, MoveJ* ...). Data objects corresponding to *iosignals* and *mecunits* are installed according to the user configuration at each system start.
- The last group *user installed objects* are objects that are defined in RAPID modules and installed according to the user configuration at the first system start or each P-start.

The objects could be any RAPID object, that is procedure, function, record, record component, alias, const, var, or pers. Object values of pers and var could be changed, but not the code it self, because of that a modpos of a built in constant declared robtarget is not allowed.

The built in RAPID code can never be viewed.

13.1 Object scope

The scope of object denotes the area in which the object is visible. A *built in object* is visible at all other levels in the task, if not the same object name is used for another object at a level between the use of the reference and the built in level.

The table below show the order of scope levels lookup, for a object referred from different places:

Table 2: Object search according to scope rules.

Scope level / The object is used in a	Own routine	Own module (local declared)	Global in the program (global declared in one module)	Built in objects
routine declared in a user or system module	1	2	3	4
data or routine declaration in a user or system module		1	2	3
routine declared in a user installed module	1	2		3
data or routine declaration in a user installed module		1		2
installed object (only for system developers)				1

There are ways to bind a reference in runtime to objects (not functions) outside its scope. For data object see *Technical reference manual - RAPID Instruction, functions and data types - SetDataSearch* and for procedures use late binding with lookup, described in *4.5 Procedure call on page 39*.

13.2 The value of a *built in* data object durability

The init value of a *built in* PERS or VAR object is set when the object is installed. It could though be changed from the normal program. The object will always keep its latest value even if the normal program is reset, erased, or replaced. The only way to re-initialize the object is to do a P-start or to change the configuration (then an automatic P-start will be performed).

Note that the value of *built in* VAR object with a separate value per task, will be reset at PP to Main. ERRNO is an example of a *built in* VAR object with a separate value for each task.

Note that a *built in* PERS object is not replacing its init value with its latest as a normal PERS object do.

13.3 The way to define *user installed objects*

The only way to install a *user installed object* is to define the object in a RAPID module, create an new instance in the system parameter *Task modules* with the file path to that module. The attribute *Storage* must then be set to *Built in*. (see system parameter, type *Controller*). There are also an attribute for *Task modules* named *TextResource* that is only valid for *Built in* objects, this makes it possible to use national language or site

depended names in the RAPID code for identifiers, without changing the code itself. In the normal case that attribute should not be changed, but for the advanced users see *15 Text files on page 103*.

Note that all used references in a *Built in* module must be known to the system at the time for that module installation.

14 Intertask objects

There are two groups of *Intertask objects*: *installed shared object* and *System global persistent data object.*

- An installed shared object is configured as shared for all tasks. This make it possible to save memory by reusing RAPID code for more than one task. Its also the only way to share non-value and semi-value data object. (see *Built in objects* in this manual). The object could be any RAPID object, that is procedure, function, const, var, or pers.
- The current value of a system global persistent data object is shared by all tasks where it is declared with the same name and type.

14.1 Symbol levels

A symbol in RAPID could be found at different levels, in a routine, in a module (local), in the program of one task (in one module and defined as global) or at the system level. Installed shared objects are on the system level.

The system level is departed into two part, a shared part and a task part. Objects in the task part are local to that task, but objects in the shared part is global to all task.

The installed shared part is physically existing in task 0 (the shared task), but existing logical in each task.

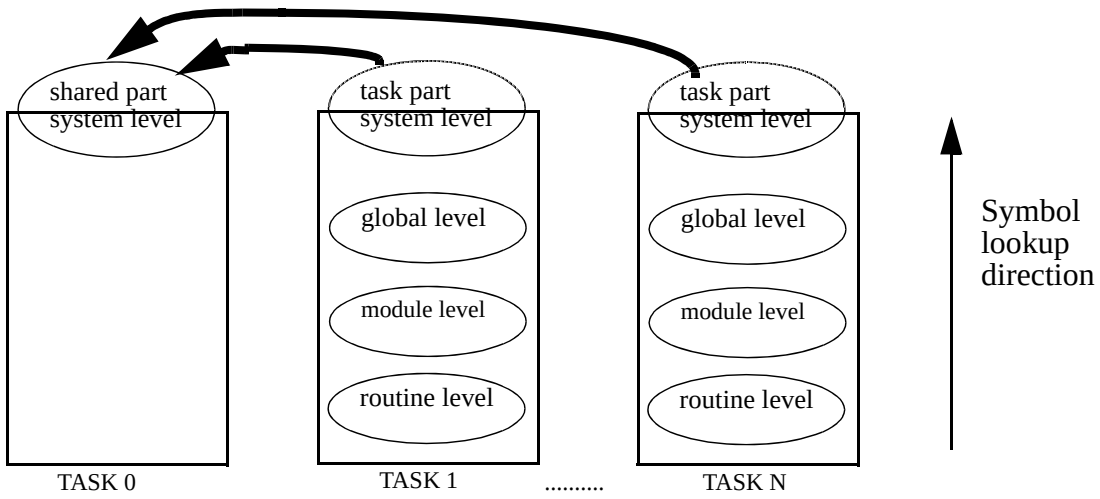


Figure 1 Symbol levels with a departed system level.

The symbol search will start from that position (level) where the object is referred and then, if not found, in nearest level above and so on. See the “Symbol lookup direction” - arrow in Figure 1.

14.2 Data object handling

Even if the definition is shared for a data object the value of it could be local in the task. That is the fact for the installed system variables *errno, intno,* and all stack allocated objects (object defined in a routine). All other data objects share the value with other tasks. This fact will demand a careful manipulation and reading of those values.

If the object has an atomic type (num, bool ...) there is no problem. But if not, make sure that the total object is read/manipulated without any interfering from another task. For example if the object is of a record type and each component are assign one by one, a reading (between the setting of two record components) from another task will get an inconsistent record.

Also remember that a routine could be called from more than one task at the same time and therefore should be reentrant, that is use local stack allocated object (parameters and data object declared in the routine).

14.3 The way to define *installed shared object*

The only way to install an *installed shared object* is to define the object in a RAPID module, create an new instance of *Task/Automatic loading of Modules* in the system parameter with the file path to the module. The attribute *shared* must be set to YES (see also system parameter type *Controller*).

14.4 System global persistent data object

The current value of a system global persistent data object (for example, not declared as task or local) is shared by all tasks where it is declared with the same name and type. The object will still exist even if one module where it is declared is removed as long as that module do not contain the last declaration for that object. A persistent object could only be of value type.

A declaration can specify an initial value to a persistent object, but it will only set the initial value of the persistent when the module is installed or loaded for the first time.

Example of usage (several initial values):

task 1: **PERS** tooldata tool1 := [...];

task 2: **PERS** tooldata tool1 := [...];

Note that the current value of *tool1* **won't** be updated with the initial value of *tool1* in the second loaded module. This is a problem if the initial value differs in the two tasks. This is solved by specifying initial value in one declaration only.

Example of usage (one initial value):

task 1: **PERS** tooldata tool1 := [...];

task 2: **PERS** tooldata tool1;

After load of the two tasks the current value of *tool1* is guaranteed to be equal to the initial value of the declaration in *task 1* regardless of the load order of the modules.

It is recommended to use this technique for types such as *tooldata*, *wobjdata* and *loaddata*. Specify initial value along with data declaration in the motiontask and omit initial value in other tasks.

It is also possible to specify no initial value at all.

Example of usage (no initial value):

task 1: **PERS** num state;

task 2: **PERS** num state;

The current value of *state* will be initialized like a variable without initial value, in this case *state* will be equal to zero. This case is useful for intertask communication where the state of the communication should not be saved when the program is saved or at backup.

15 Text files

This is a most effective tool that should be used when the demand for the application includes:

- Easily changeable texts, for example help and error texts (the customer should also be able to handle these files)
- Memory saving, text strings in a Text file use a smaller amount of memory than RAPID strings.

In a Text file you can use ASCII strings, with the help of an off-line editor, and fetch them from the RAPID code. The RAPID code should not be changed in any way even if the result from the execution may look totally different.

15.1 Syntax for a text file

The application programmer must build one (or several) ASCII file(s), the “Text file(s)”, that contains the strings for a text resource.

The “Text file” is organized as:

```
<text_resource>::
# <comment>
<index1>:
 “<text_string>”

# <comment>
<index2>:
 “<text_string>”
...
```

The parameters

<text_resource>

This name identifies the text resource. The name must end with “::”. If the name does not exist in the system, the system will create a new text resource, otherwise the indexes in the file will be added to a resource that already exists. The application developer is responsible for ensuring that one’s own resources do not crash with another application. A good rule is to use the application name as a prefix to the resource name, for example MYAPP_TXRES. The name of the resource must not exceed 80 characters. Do not use exclusively alphanumeric as the name to avoid a collision with system resources.

<index>

This number identifies the <text_string> in the text resource. This number is application defined and it must end with a “:” (colon).

<text_string>

The text string starts on a new line and ends at the end of the line or, if the text string must be formatted on several lines, the new line character string “\n” must be inserted.

<comment>

A comment always starts on a new line and goes to the end of the line. A comment is always preceded by “#”. The comment will not be loaded into the system.

15.2 Retrieving text during program execution

It is possible to retrieve a text string from the RAPID code. The functions *TextGet* and *TextTabGet* are used for this, see the chapter RAPID Support Functions.

The text file: text_file.eng

```
#This is the text file .....
ACTION_TXRES::
# Start the activity
1:
Go
# Stop the activity
2:
Stop
3:
Wait
# Get Help
10:
Call_service_man
#Restart the controller
11:
Restart
```

The module: write_from_file.mod

```
MODULE write_from_file

VAR num text_res_no;
VAR string text1;

PROC main()

IF TextTabFreeToUse("ACTION_TXRES") THEN
TextTabInstall "HOME:/text_file.eng";
  ENDIF

  text_res_no := TextTabGet("ACTION_TXRES");
  text1 := TextGet(text_res_no, 2);
  TPWrite text1; ! The word “Stop” will be printed.

ENDPROC

ENDMODULE
```

Figure 2 Example with text from separate text file.

15.3 Loading text files

Loading of the text file into the system can be done with the RAPID instruction *TextTabInstall* and the function *TextTabFreeToUse.*

16 Storage allocation for RAPID objects

All RAPID programs stored on PC or controller have ASCII format.
At loading of RAPID program from PC/controller memory into the program memory (internal format), the storage of the program needs about four times more memory space.

For memory optimization of RAPID programs, the storage allocation in program memory (internal format in bytes) for some common instructions, data etc. are specified below.

For other instructions or data the storage allocation can be read from the operating message 10040 after loading of a program or program module.

16.1 Module, routine, program flow and other basic instruction

Table 3 Storage allocation for module, routine, program flow and other basic instructions

Instruction or data	Storage in bytes
New empty module: MODULE module1 ... ENDMODULE	1732
New empty procedure without parameters: PROC proc1() ... ENDPROC	224
Procedure call without arguments: proc1;	36
Module numeric variable declaration: VAR num reg1;	**156**
Numeric assignment: reg1:=1;	44
Compact IF: IF reg1=1 proc1;	124
IF statement: IF reg1=1 THEN proc1; ELSE proc2; ENDIF	184
Waits a given amount of time: WaitTime 1;	88
Comments: ! 0 - 7 chars (for every additional 4 chars)	36 (+4)
Module string constant declaration with 0-80 chars init string value: CONST string string1 := “0-80 characters“;	**332**
Module string variable declaration with 0-80 chars init string value: VAR string string1 := “0-80 characters“;	**344**
Module string variable declaration: VAR string string1;	**236**
String assignment: string1:= “0-80 characters”;	52
Write text on FlexPendant: TPWrite “0-80 characters”;	**176**

16.2 Move instructions

Table 4 Storage allocation for move instructions.

Instruction or data	Storage in bytes
Module robtarget constant declaration: CONST robtarget p1 := [...];	292
Robot linearly move: MoveL p1,v1000,z50,tool1;	244
Robot linearly move: MoveL *,v1000,z50,tool1;	312
Robot circular move: MoveC *,*,v1000,z50,tool1;	432

16.3 I/O instructions

Table 5 Storage allocation for I/O instructions

Instruction or data	Storage in bytes
Set digital output: Set do1;	88
Set digital output: SetDO do1,1;	140
Wait until one digital input is high: WaitDI di1,1;	140
Wait until two digital inputs are high: WaitUntil di1=1 AND di2=1;	220

S

T

U

V

W

Contact us

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

3HAC16585-1 Rev G, en