ABB Robotics

Operating manual
Introduction to RAPID

Power and productivity
for a better world™

Operating manual

Introduction to RAPID

RobotWare 5.0

Document ID: 3HAC029364-001

Revision: -

ABB AB
Robotics Products
SE-721 68 Västerås
Sweden

Overview

About This Manual

This manual is intended as a first introduction to RAPID. A lot of functionality in RAPID is left out, but the most essential parts are described so that it can be easily understood for everybody. This manual does not make you an expert RAPID programmer, but it can help you understand the concept of programming with RAPID. The details can always be found in the reference manuals.

Usage

This manual should be read before starting to program. It does not contain everything you need to know, but you need to be familiar with most things in this manual, before starting to write a RAPID program.

This manual does not replace the educational courses in RAPID, but can complement it.

Who Should Read This Manual?

This manual is intended for someone with no previous experience in programming, e.g. a robot operator who wants to learn how to program the robot.

Prerequisites

There are no prerequisites for this manual.

Organization of Chapters

The manual is organized in the following chapters:

Chapter	Contents
1. RAPID basics	The fundamentals of programming. This functionality is similar in most high level programming languages.
2. RAPID robot functionality	Describes the functionality that makes RAPID unique, i.e. move instructions, I/O signals and communication with a FlexPendant.
3. Structure	Describes how to create procedures. Also contains a brief introduction to how to apply a structured design of a program.
4. Data with multiple values	Describes arrays and complex data types.
5. RAPID instructions and functions	A short explanation of what the RAPID instructions and functions are.
6. What to read next	Where to find more information if you want to continue your studies of RAPID.

References

Reference	Document Id
Technical reference manual - RAPID overview	3HAC16580-1
Technical reference manual - RAPID Instructions, Functions and Data types	3HAC16581-1
Technical reference manual - RAPID kernel	3HAC16585-1
Operating manual - IRC5 with FlexPendant	3HAC16590-1

Continues on next page

Continued

Revisions

Revision	Description
-	First edition

Product documentation, M2004

General

The robot documentation is divided into a number of categories. This listing is based on the type of information contained within the documents, regardless of whether the products are standard or optional. This means that any given delivery of robot products *will not contain all* documents listed, only the ones pertaining to the equipment delivered.

However, all documents listed may be ordered from ABB. The documents listed are valid for M2004 robot systems.

Product manuals

All hardware, robots and controllers, will be delivered with a **Product manual** that contains:

- Safety information
- Installation and commissioning (descriptions of mechanical installation, electrical connections)
- Maintenance (descriptions of all required preventive maintenance procedures including intervals)
- Repair (descriptions of all recommended repair procedures including spare parts)
- Additional procedures, if any (calibration, decommissioning)
- Reference information (article numbers for documentation referred to in Product manual, procedures, lists of tools, safety standards)
- Part list
- Foldouts or exploded views
- Circuit diagrams

Technical reference manuals

The following manuals describe the robot software in general and contain relevant reference information:

- **RAPID Overview**: An overview of the RAPID programming language.
- **RAPID Instructions, Functions and Data types**: Description and syntax for all RAPID instructions, functions and data types.
- **System parameters**: Description of system parameters and configuration workflows.

Application manuals

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

An application manual generally contains information about:

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

Continues on next page

Continued

Operating manuals

This group of manuals is aimed at those having first hand operational contact with the robot, that is production cell operators, programmers and trouble shooters. The group of manuals includes:

- **Emergency safety information**
- **Getting started - IRC5 and RobotStudio**
- **IRC5 with FlexPendant**
- **RobotStudio**
- **Trouble shooting - IRC5** for the controller and robot

Safety

Safety of personnel

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

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

Safety regulations

Before beginning work with the robot, make sure you are familiar with the safety regulations described in *Operating manual - IRC5 with FlexPendant*.

Terminology

About the terms

This manual is generally written for beginners, regarding both programming and robots. However, some terms are used that may be familiar only to those with some knowledge about programming and/or industrial robots. These terms are described in this terminology.

Terms

Term	Description
FlexPendant	A hand held terminal for controlling a robot system.
Robot controller	The robot controller is basically a computer that controls the robot.
Syntax	Rules for how a language is allowed to be written. It can be seen as the grammar of the programming language. The syntax of a programming language is much more strict than in ordinary human language. Humans are intelligent and would understand if I say "I fast run" instead of "I run fast". Computers, on the other hand, are stupid and would not understand anything unless the syntax is absolutely correct.

1 RAPID basics

1.1. About RAPID

What is RAPID

If you want a computer to do something, a program is required. RAPID is a programming language for writing such a program.

The native language of computers consists of only zeros and ones. This is virtually impossible for humans to understand. Therefore computers are taught to understand a language that is relatively easy to understand - a high level programming language. RAPID is a high level programming language, it uses some English words (like IF and FOR) to make it understandable for humans.

Simple RAPID program example

Let us look at a simple example to see what a RAPID program can look like:

```
MODULE MainModule
  VAR num length;
  VAR num width;
  VAR num area;

  PROC main()
     length := 10;
     width := 5;
     area := length * width;
     TPWrite "The area of the rectangle is " \Num:=area;
  END PROC
ENDMODULE
```

This program will calculate the area of a rectangle and write on the FlexPendant:

```
The area of the rectangle is 50
```

1.2 RAPID data

1.2.1. Variables

Data types

There are many different data types in RAPID. For now, we will focus on the three general data types:

Data type	Description
num	Numerical data, can be both integer and decimal number. E.g. 10 or 3.14159.
string	A text string. E.g. "This is a string". Maximum of 80 characters.
bool	A boolean (logical) variable. Can only have the values `TRUE` or `FALSE`.

All other data types are based on these three. If you understand them, how to perform operations on them and how they can be combined to more complex data types, you can easily understand all data types.

Variable characteristics

A variable contains a data value. If the program is stopped and started the variable keeps its value, but if the program pointer is moved to main the variable data value is lost.

Declaring a variable

Declaring a variable is the way of defining a variable name and which data type it should have. A variable is declared using the keyword `VAR`, according to the syntax:

```
VAR datatype identifier;
```

Example

```
VAR num length;
VAR string name;
VAR bool finished;
```

Assigning values

A value is assigned to a variable using the instruction `:=`

```
length := 10;
name := "John"
finished := TRUE;
```

Note that `:=` is not an equal sign. It means that the expression to the right is passed to the variable on the left. There can only be a variable to the left of `:=`

For example, the following is a correct RAPID code resulting in `reg1` having the value 3:

```
reg1 := 2;
reg1 := reg1 + 1;
```

The assignment can be made at the same time as the variable declaration:

```
VAR num length := 10;
VAR string name := "John";
VAR bool finished := TRUE;
```

1.2.2. Persistent variables

What is a persistent variable

A persistent variable is basically the same as an ordinary variable, but with one important difference. A persistent variable remembers the last value it was assigned, even if the program is stopped and started from the beginning again.

Declaring a persistent variable

A persistent variable is declared using the keyword `PERS`. At declaration an initial value mustbe assigned.

```
PERS num nbr := 1;
PERS string string1 := "Hello";
```

Example

Consider the following code example:

```
PERS num nbr := 1;
PROC main()
  nbr := 2;
ENDPROC
```

If this program is executed, the initial value is changed to 2. The next time the program is executed the program code will look like this:

```
PERS num nbr := 2;
PROC main()
  nbr := 2;
ENDPROC
```

1.2.3. Constants

What is a constant?

A constant contains a value, just like a variable, but the value is always assigned at declaration and after that the value can never be changed. The constant can be used in the program in the same way as a variable, except that it is not allowed to assign a new value to it.

Constant declaration

The constant is declared using the keyword `CONST` followed by data type, identifier and assignment of a value.

```
CONST num gravity := 9.81;
CONST string greating := "Hello"
```

Why use constants?

By using a constant instead of a variable, you can be sure that the value is not changed somewhere in the program.

Using a constant instead of writing the value directly in the program is better if you need to update the program with another value on the constant. Then you only have to change in one place and can be sure you have not forgotten any occurrence of the value.

1.2.4. Operators

Numerical operators

These operators operate on the data type `num` and return the data type `num`. I.e. in the examples below, variables `reg1`, `reg2` and `reg3` are of data type `num`.

Operator	Description	Example
+	Addition	`reg1 := reg2 + reg3;`
-	Subtraction Unary minus	`reg1 := reg2 - reg3;` `reg1 := -reg2;`
*	Multiplication	`reg1 := reg2 * reg3;`
/	Division	`reg1 := reg2 / reg3;`

Relational operators

These operators return the data type `bool`.

In the examples, `reg1` and `reg2` are data type `num` while `flag1` is `bool`.

Operator	Description	Example
=	equal to	`flag1 := reg1 = reg2;` `flag1` is `TRUE` if `reg1` equals `reg2`
<	less than	`flag1 := reg1 < reg2;` `flag1` is `TRUE` if `reg1` is less than `reg2`
>	greater than	`flag1 := reg1 > reg2;` `flag1` is `TRUE` if `reg1` is greater than `reg2`
<=	less than or equal to	`flag1 := reg1 <= reg2;` `flag1` is `TRUE` if `reg1` is less than or equal to `reg2`
>=	greater than or equal to	`flag1 := reg1 >= reg2;` `flag1` is `TRUE` if `reg1` is greater than or equal to `reg2`
<>	not equal to	`flag1 := reg1 <> reg2;` `flag1` is `TRUE` if `reg1` is not equal to `reg2`

Logical operators are often used together with the `IF` instruction. For code examples, see *Examples with logical conditions and IF statements on page 18*.

String operator

Operator	Description	Example

+	String concatenation	`VAR string firstname := "John";`
		`VAR string lastname := "Smith";`
		`VAR string fullname;`
		`fullname := firstname + " " + lastname;`
		The variable `fullname` will contain the string "John Smith".

1.3 Controlling the program flow

1.3.1. IF THEN

About the program flow

The program examples we have seen so far are executed sequentially, from top to bottom. For more complex programs, we may want to control which code is executed, in which order, and how many times. First we will have a look at how to set up conditions for if a program sequence should be executed or not.

IF

The `IF` instruction can be used when a set of statements only should be executed if a specified condition is met.

If the logical condition in the `IF` statement is true, then the program code between the keywords `THEN` and `ENDIF` is executed. If the condition is false, that code is not executed and the execution continues after `ENDIF`.

Example

In this example the string `string1` is written on the FlexPendant if it is not an empty string. If `string1` is an empty string, i.e. contains no characters, then no action is taken.

```
VAR string string1 := "Hello";

IF string1 <> "" THEN
  TPWrite string1;
ENDIF
```

ELSE

An `IF` statement can also contain program code to be executed if the condition is false.

If the logical condition in the `IF` statement is true, then the program code between the keywords `THEN` and `ELSE` is executed. If the condition is false, then the code between the keywords `ELSE` and `ENDIF` is executed.

Example

In this example the string `string1` is written on the FlexPendant if it is not an empty string. If `string1` is an empty string, then the text "The string is empty" is written.

```
VAR string string1 := "Hello";

IF string1 <> "" THEN
  TPWrite string1;
ELSE
  TPWrite "The string is empty";
ENDIF
```


Continues on next page

Continued

ELSEIF

Sometimes you have more than two alternative program sequences. You can then use `ELSEIF` to set up several alternatives.

Example

In this example different texts are written depending on the value on the variable `time`.

```
VAR num time := 38.7;

IF time < 40 THEN
  TPWrite "Part produced at fast rate";
ELSEIF time < 60 THEN
  TPWrite "Part produced at average rate";
ELSE
  TPWrite "Part produced at slow rate";
ENDIF
```

Note that since the first condition is true the first text will be written. The two other texts will not be written (even though it is true that `time` is less than 60).

1.3.2. Examples with logical conditions and IF statements

Example

Use the `IF` statement to determine which text to write on the FlexPendant. Write on the FlexPendant which part is fastest to produce.

```
VAR string part1 := "Shaft";
VAR num time1;
VAR string part2 := "Pipe";
VAR num time2;

PROC main()
  time1 := 41.8;
  time2 := 38.7;
  IF time1 < time2 THEN
     TPWrite part1 + " is fastest to produce";
  ELSEIF time1 > time2 THEN
     TPWrite part2 + " is fastest to produce";
  ELSE
     TPWrite part1 + " and " + part2 + " are equally fast to
          produce";
  ENDIF
ENDPROC
```

Example

If it takes more than 60 seconds to produce a part, write a message on the FlexPendant. If the boolean variable `full_speed` is `FALSE` the message will tell the operator to increase the robot speed. If `full_speed` is `TRUE`, the message will ask the operator to examine the reason for the slow production.

```
VAR num time := 62.3;
VAR bool full_speed := TRUE;

PROC main()
  IF time > 60 THEN
     IF full_speed THEN
        TPWrite "Examine why the production is slow";
     ELSE
        TPWrite "Increase robot speed for faster production";
     ENDIF
  ENDIF
ENDPROC
```

1.3.3. FOR loop

Repeating a code sequence

Another way of controlling the program flow is to repeat a program code sequence a number of times.

How does the FOR loop work

The following code will repeat writing "Hello" 5 times:

```
FOR i FROM 1 TO 5 DO
  TPWrite "Hello";
ENDFOR
```

The syntax of the FOR statement is:

```
FOR counter FROM startvalue TO endvalue DO
  program code to be repeated
ENDFOR
```

The *counter* does not have to be declared, but acts as a numeric variable inside the FOR loop. The first time the code is executed, the *counter* has the value specified by the *startvalue*. The value of the *counter* is then increased by 1 for each time the code is executed. The last time the code executes is when the *counter* is equal to *endvalue*. After that, the execution continues with the programming code after ENDFOR.

Using the counter value

The value of the *counter* can be used in the FOR loop.

For example, calculating the sum of all numbers from 1 to 50 (1+2+3+...+49+50) can be programmed like this:

```
VAR num sum := 0;

FOR i FROM 1 TO 50 DO
  sum := sum + i;
ENDFOR
```

It is **not** allowed to assign a value to the *counter* in the FOR loop.

1.3.4. WHILE loop

Repeating with condition

The repeating of a code sequence can be combined with the conditional execution of the code sequence. With the `WHILE` loop the program will continue repeating the code sequence as long as the condition is true.

WHILE syntax

The syntax for the `WHILE` loop is:

```
WHILE condition DO
  program code to be repeated
ENDWHILE
```

If the *condition* is false to begin with, the code sequence will never be executed. If the *condition* is true, the code sequence will be executed repeatedly until the *condition* is no longer true.

Example

The following program code will add numbers (1+2+3+...) until the sum reaches 100.

```
VAR num sum := 0;
VAR num i := 0;

WHILE sum <= 100 DO
  i := i + 1;
  sum := sum + i;
ENDWHILE
```

Do not create eternal or heavy loops without wait instruction

If the condition never becomes false the loop will continue constantly and consume vast amount of computer power. It is allowed to write an eternal loop, but then it must contain some waiting instruction that allows the computer to perform other tasks in the meantime.

Heavy loops (with lots of calculations and writing on the FlexPendant, without move instructions) can require some waiting instruction even if the number of loops are limited.

```
WHILE TRUE DO
  ! Some code
  ...
  ! Wait instruction that waits for 1 second
  WaitTime 1;
ENDWHILE
```

Note that move instructions work as waiting instructions, since the execution does not continue until the robot has reached its target.

1.4 Rules and recommendations for RAPID syntax

1.4.1. General RAPID syntax rules

Semicolon

The general rule is that each statement ends with a semicolon.

Examples

Variable declaration:

```
VAR num length;
```

Assigning values:

```
area := length * width;
```

Most instruction calls:

```
MoveL p10,v1000,fine,tool0;
```

Exceptions

Some special instructions do not end with a semicolon. Instead there are special keywords to indicate where they end.

Example of instructions that do not end with semicolon:

Instruction keyword	Terminating keyword
IF	ENDIF
FOR	ENDFOR
WHILE	ENDWHILE
PROC	ENDPROC

These keywords are very important to create a good structure of a RAPID program. They are thoroughly described later in this manual.

Comments

A line starting with ! will not be interpreted by the robot controller. Use this to write comments about the code.

Example

```
! Calculate the area of the rectangle
area := length * width;
```

1.4.2. Recommendations for RAPID code

Capitalized keywords

RAPID is not case sensitive, but it is recommended that all reserved words (e.g. `VAR`, `PROC`) are written in capital letters. For a complete list of reserved words, see *Technical reference manual - RAPID overview*.

Indentations

To make the programming code easy to grasp, use indentation. Everything inside a `PROC` (between `PROC` and `ENDPROC`) should be indented. Everything inside an `IF`-, `FOR`- or `WHILE`-statement should be further indented.

When programming with the FlexPendant, the indentation is done automatically.

Example

```
VAR bool repeat;
VAR num times;

PROC main()
  repeat := TRUE;
  times := 3;
  IF repeat THEN
     FOR i FROM 1 TO times DO
        TPWrite "Hello!";
     ENDFOR
  ENDIF
END PROC
```

Note that it is easy to see where the `IF` statement starts and ends. If you would have several `IF` statements and no indentations, it would be virtually impossible to find which `ENDIF` corresponds to which `IF`.

2 RAPID robot functionality

2.1 Move instructions

2.1.1. MoveL instruction

Overview

The advantage with RAPID is that, except for having most functionality found in other high level programming languages, it is specially designed to control robots. Most importantly, there are instructions for making the robot move.

MoveL

A simple move instruction can look like this:

```
MoveL p10, v1000, fine, tool0;
```

where:

- `MoveL` is an instruction that moves the robot linearly (in a straight line) from its current position to the specified position.
- `p10` specifies the position that the robot shall move to.
- `v1000` specifies that the speed of the robot shall be 1000 mm/s.
- `fine` specifies that the robot shall go exactly to the specified position and not cut any corners on its way to the next position.
- `tool0` specifies that it is the mounting flange at the tip of the robot that should move to the specified position.

MoveL syntax

```
MoveL ToPoint Speed Zone Tool;
```

ToPoint

The destination point defined by a constant of data type `robtarget`. When programming with the FlexPendant you can assign a `robtarget` value by pointing out a position with the robot. When programming offline, it can be complicated to calculate the coordinates for a position.

`robtarget` will be explained further later, in section *Composite data types on page 44*. For now, let us just accept that the position x=600, y=-100, z=800 can be declared and assigned like this:

```
CONST robtarget p10 := [ [600, -100, 800], [1, 0, 0, 0], [0, 0, 0,
     0], [ 9E9, 9E9, 9E9, 9E9, 9E9, 9E9] ];
```

Speed

The speed of the movement defined by a constant of data type `speeddata`. There are plenty of predefined values, such as:

Predefined speeddata	Value
v5	5 mm/s
v100	100 mm/s
v1000	1000 mm/s

Continues on next page

2.1.1. MoveL instruction

Continued

Predefined speeddata	Value
vmax	Maximum speed for the robot

A complete list of predefined `speeddata` values is found in *Technical reference manual - RAPID Instructions, Functions and Data types*, section *Data types* and *speeddata*.

When using a predefined value, it should not be declared or assigned.

Zone

Specifies a corner zone defined by a constant of data type `zonedata`. There are many predefined values, such as:

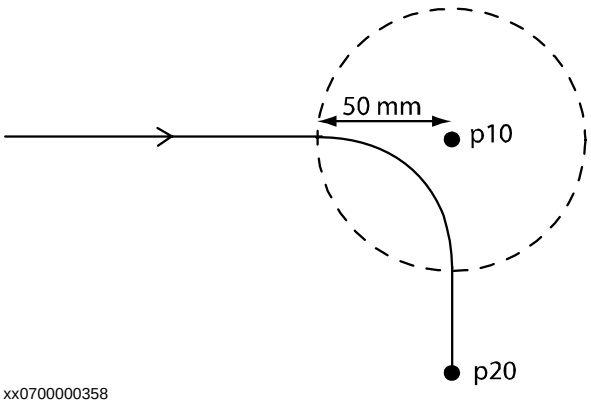
Predefined zonedata	Value
fine	The robot will go to exactly the specified position
z10	The robot path can cut corners when it is less than 10 mm from *ToPoint*.
z50	The robot path can cut corners when it is less than 50 mm from *ToPoint*.

A complete list of predefined `zonedata` values is found in *Technical reference manual - RAPID Instructions, Functions and Data types*, section *Data types* and *zonedata*.

When using a predefined value, it should not be declared or assigned.

The following RAPID instructions will result in the robot path shown below:

```
MoveL p10, v1000, z50, tool0;
MoveL p20, v1000, fine, tool0;
```



xx0700000358

Tool

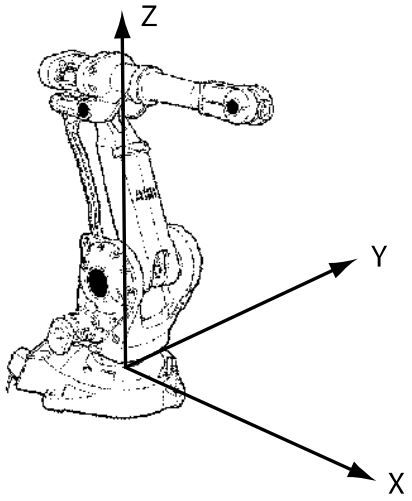
Specifies the tool that the robot is using, defined by a persistent variable of data type `tooldata`. If a welding gun, glue gun or a pen is attached to the robot, we want to program the *ToPoint* for the tip of this tool. This is done automatically if a `tooldata` is declared, assigned and used in the `MoveL` instruction.

`tool0` is a predefined tool, representing the robot without any tool mounted on it, and should not be declared or assigned. Any other tool should be declared and assigned before being used.

2.1.2. Coordinate systems

Base coordinate system

The position that a move instruction moves to is specified as coordinates in a coordinate system. If no coordinate system is specified, the position is given relative to the robot base coordinate system (also called base frame). The base coordinate system has its origin in the robot base.



xx0700000397

Customized coordinate systems

Another coordinate system can be defined and used by move instructions. Which coordinate system the move instruction shall use is specified with the optional argument `\WObj`.

```
MoveL p10, v1000, z50, tool0 \WObj:=wobj1;
```

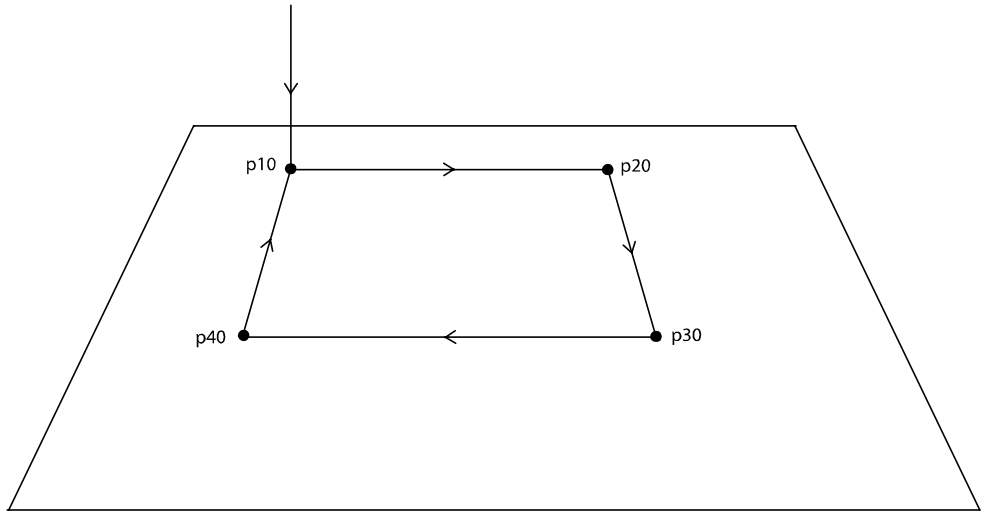
For information about how to define a coordinate system, see *Technical reference manual - RAPID Instructions, Functions and Data types*, section *Data types* and *wobjdata*.

For more information about coordinate systems, see *Technical reference manual - RAPID overview*, section *Coordinate systems*.

2.1.3. Examples with MoveL

Draw a square

A robot is holding a pen above a piece of paper on a table. We want the robot to move the tip of the pen down to the paper and then draw a square.



xx0700000362

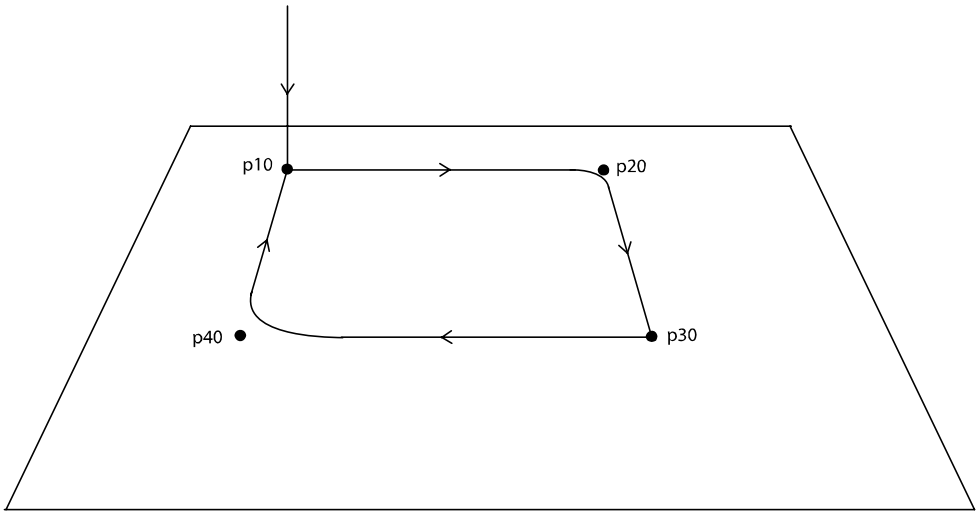
```
PERS tooldata tPen := [ TRUE, [[200, 0, 30], [1, 0, 0 ,0]], [0.8,
     [62, 0, 17], [1, 0, 0, 0], 0, 0, 0]];
CONST robtarget p10 := [ [600, -100, 800], [0.707170, 0, 0.707170,
     0], [0, 0, 0, 0], [ 9E9, 9E9, 9E9, 9E9, 9E9, 9E9] ];
CONST robtarget p20 := [ [600, 100, 800], [0.707170, 0, 0.707170,
     0], [0, 0, 0, 0], [ 9E9, 9E9, 9E9, 9E9, 9E9, 9E9] ];
CONST robtarget p30 := [ [800, 100, 800], [0.707170, 0, 0.707170,
     0], [0, 0, 0, 0], [ 9E9, 9E9, 9E9, 9E9, 9E9, 9E9] ];
CONST robtarget p40 := [ [800, -100, 800], [0.707170, 0, 0.707170,
     0], [0, 0, 0, 0], [ 9E9, 9E9, 9E9, 9E9, 9E9, 9E9] ];

PROC main()
  MoveL p10, v200, fine, tPen;
  MoveL p20, v200, fine, tPen;
  MoveL p30, v200, fine, tPen;
  MoveL p40, v200, fine, tPen;
  MoveL p10, v200, fine, tPen;
ENDPROC
```


Continues on next page

Draw with corner zones

Draw the same figure as in the previous example, but with a corner zone of 20 mm at `p20` and a corner zones of 50 mm at `p40`.



xx0700000363

```
VAR tooldata tPen := ...
...
VAR robtarget p40 := ...

PROC main()
  MoveL p10, v200, fine, tPen;
  MoveL p20, v200, z20, tPen;
  MoveL p30, v200, fine, tPen;
  MoveL p40, v200, z50, tPen;
  MoveL p10, v200, fine, tPen;
ENDPROC
```

2.1.4. Other move instructions

Several move instructions

There are a number of move instructions in RAPID. The most common are `MoveL`, `MoveJ` and `MoveC`.

MoveJ

`MoveJ` is used to move the robot quickly from one point to another when that movement does not have to be in a straight line.

Use `MoveJ` to move the robot to a point in the air close to where the robot will work. A `MoveL` instruction does not work if, for example, the robot base is between the current position and the programmed position, or if the tool reorientation is too large. `MoveJ` can always be used in these cases.

The syntax of `MoveJ` is analog with `MoveL`.

Example

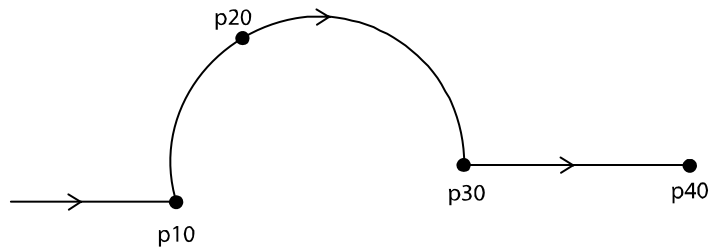
```
MoveJ p10, v1000, fine, tPen;
```

MoveC

`MoveC` is used to move the robot circularly in an arc.

Example

```
MoveL p10, v500, fine, tPen;
MoveC p20, p30, v500, fine, tPen;
MoveL p40, v500, fine, tPen;
```



xx0700000364

2.1.5. Execution behavior in corner zones

Why the special execution in corner zones?

The execution of a program is usually carried out in the order the statements are written.

In the following example the robot first moves to `p10`, then calculates the value of `reg1` and then moves to `p20`:

```
MoveL p10, v1000, fine, tool0;
reg1 := reg2 + reg3;
MoveL p20, v1000, fine, tool0;
```

But now look at this example:

```
MoveL p10, v1000, z50, tool0;
reg1 := reg2 + reg3;
MoveL p20, v1000, fine, tool0;
```

If the calculation of `reg1` would not start until the robot was at `p10`, then the robot would have to stop there and wait for the next move instruction. What actually happens is that the code is executed ahead of the robot movement. `reg1` is calculated and the robot path in the corner zone is calculated before the robot reaches `p10`.

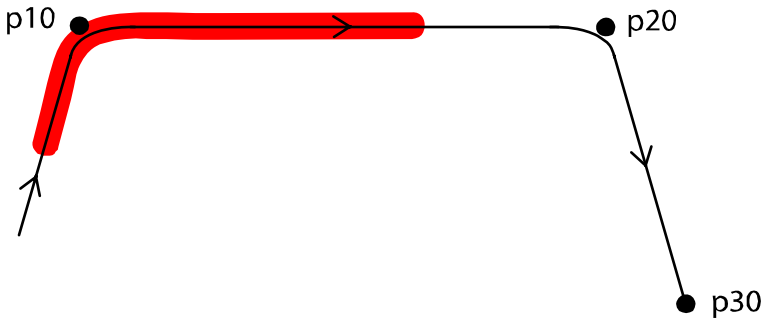
How does this affect my program

In many cases the exact time of execution does not affect the program. There are however examples of when it does affect the program.

If you want to draw a line with a spray gun between `p10` and `p20`, and write the program like this:

```
MoveL p10, v300, z10, tspray;
! Start spraying
SetDO do1, 1;
MoveL p20, v300, z10, tspray;
! Stop spraying
SetDO do1, 0;
MoveL p30, v300, fine, tspray;
```

The result may look something like this:



xx0700000387

Solution

If you want to set signals in corner zones, and not use `fine`, then there are special instructions for solving this, e.g. `MoveLDO`, `TriggL` and `DispL`. For more information about these instructions, see *Technical reference manual - RAPID Instructions, Functions and Data types*.

Continues on next page

Continued

Avoid wait instructions or heavy calculations after corner zone

The robot controller can calculate the robot movement in corner paths even if there are instructions in between the move instructions. However, if there is a wait instruction after a move instruction with a corner zone, the robot will not be able to handle this. Use `fine` in the move instruction before a wait instruction.

There is also a limitation as to how many (and complex) calculations the robot controller can calculate in between move instructions with corner zones. This is mainly a problem when calling procedures after a move instruction with a corner zone.

2.2 I/O signals

2.2.1. I/O signals

About signals

Signals are used for communication with external equipment that the robot cooperates with. Input signals are set by the external equipment and can be used in the RAPID program to initiate when to perform something with the robot. Output signals are set by the RAPID program and signals to the external equipment that they should do something.

Setting up signals

Signals are configured in the system parameters for the robot system. It is possible to set customized names for the signals. They should not be declared in the RAPID program.

Digital input

A digital input signal can have the values 0 or 1. The RAPID program can read its value but cannot set its value.

Example

If the digital input signal `di1` is 1 then the robot will move.

```
IF di1 = 1 THEN
  MoveL p10, v1000, fine, tPen;
ENDIF
```

Digital output

A digital output signal can have the values 0 or 1. The RAPID program can set the value for a digital output signal, and thus affect external equipment. The value of a digital output signal is set with the instruction `SetDO`.

Example

The robot has a grip tool that can be closed with the digital output signal `do_grip`. The robot moves to the position where the pen is and closes the gripper. The robot then moves to where it shall draw, now using the tool `tPen`.

```
MoveJ p0, vmax, fine, tGripper;
SetDO do_grip, 1;
MoveL p10, v1000, fine, tPen;
```

Other signal types

Digital signals are most common and easy to use. If a signal needs to have another value than 0 or 1, there are analog signals and groups of digital signals that can have other values. These types of signals are not covered in this manual.

2.3 User interaction

2.3.1. Communicate with the FlexPendant

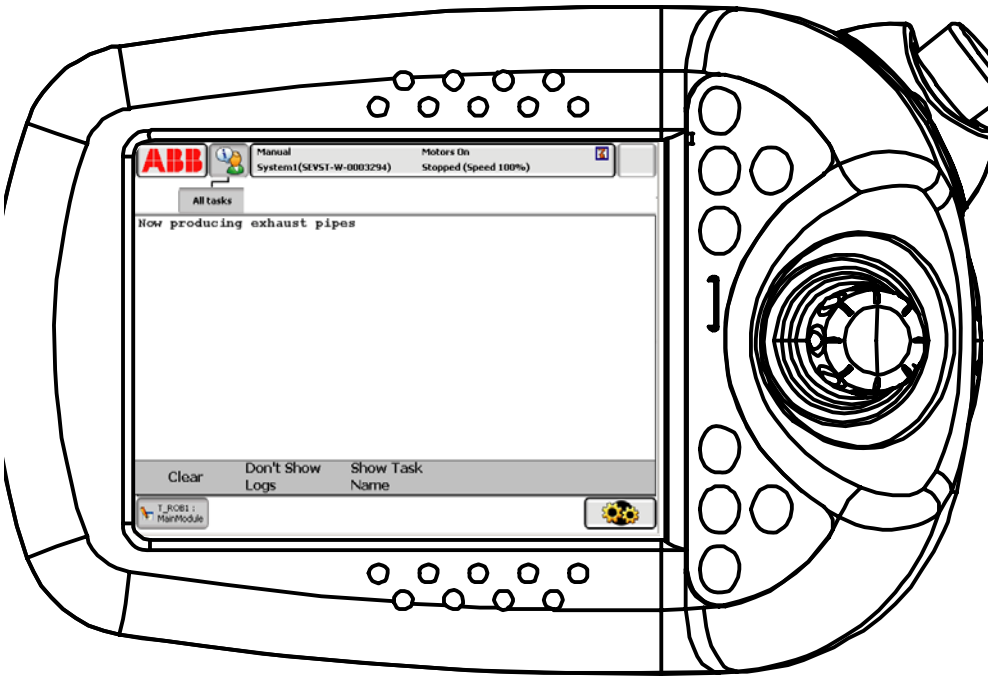
About read and write instructions

In RAPID, there are several instructions for writing information to the robot operator, as well as receiving input from the operator. We have already seen `TPWrite` in previous examples. The only instructions we will look at here are `TPWrite`, `TPReadFK` and `TPReadNum`.

TPWrite

Writing a message to the operator can be made with the instruction `TPWrite`.

```
TPWrite "Now producing exhaust pipes";
```



xx0700000374

Write a string variable

The text string written on the FlexPendant can come from a string variable, or the written text can be a concatenation of a string variable and another string.

```
VAR string product := "exhaust pipe";
! Write only the product on the FlexPendant
TPWrite product;
! Write "Producing" and the product on the FlexPendant
TPWrite "Producing " + product;
```

Write a numerical variable

A numerical variable can be added after the string using the optional argument `\Num`.

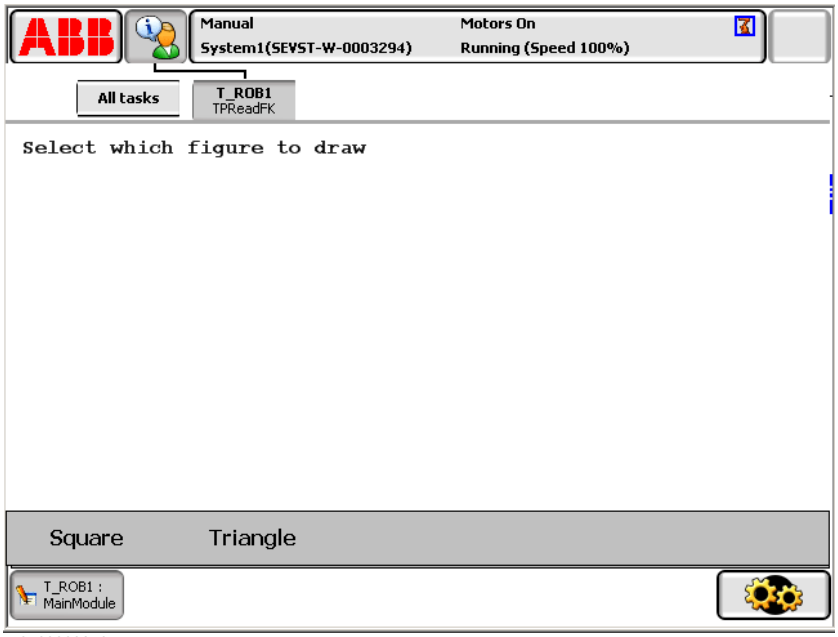
```
VAR num count := 13;
TPWrite "The number of produced units is: " \Num:=count;
```

Continues on next page

TPReadFK

When writing a RAPID program that requires the operator to make a choice, `TPReadFK` is a useful instruction. It allows up to five function keys to be displayed, and the operator can choose which one to tap. The buttons will correspond to the values 1 to 5.

```
VAR num answer;
TPReadFK answer, "Select which figure to draw", "Square",
     "Triangle", stEmpty, stEmpty, stEmpty;
IF answer = 1 THEN
  ! code to draw square
ELSEIF answer = 2 THEN
  ! code to draw triangle
ELSE
  ! do nothing
ENDIF
```



xx0700000376

If the user selects "Square", the numeric variable `answer` gets the value 1. If the user selects "Triangle", the numeric variable `answer` gets the value 2.

Five functions keys can be specified. If a key is not being used, write `stEmpty` instead of the text on the button.

TPReadNum

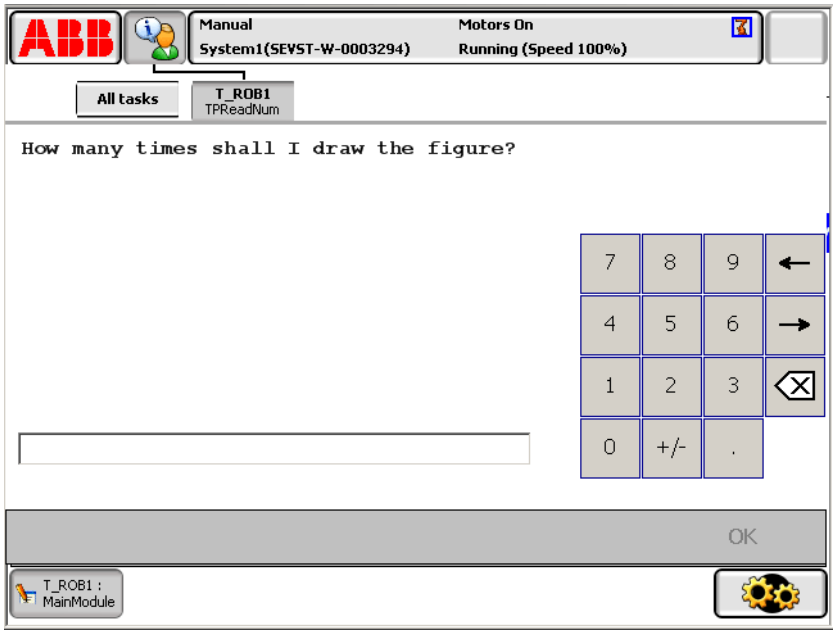
`TPReadNum` allows the operator to write a number on the FlexPendant, rather than just making a choice.

```
VAR num answer;
TPReadNum answer, "How many times shall I draw the figure?";
FOR i FROM 1 TO answer DO
  ! code to draw figure
ENDFOR
```

Continues on next page

2.3.1. Communicate with the FlexPendant

Continued



xx0700000378

The numeric variable `answer` gets the value that the operator types.

3 Structure

3.1. RAPID procedure

What is a procedure

So far, all the RAPID code examples we have looked at have only executed code in the procedure `main`. The execution automatically starts in the procedure named `main`, but there can be several procedures.

A procedure must be declared with the keyword `PROC` followed by the procedure name, the procedure arguments and the program code that the procedure should execute. A procedure is called from another procedure (except `main`, which is automatically called when the program starts).

Example

If we want to draw four squares of different sizes, we could write almost the same program code four times. This would result in a lot of code and make the program difficult to understand. A much more efficient way to write this program is to make a procedure that draws the square, and let the main procedure call this procedure four times.

```
PERS tooldata tPen:= ...
CONST robtarget p10:= ...

PROC main()
  ! Call the procedure draw_square
  draw_square 100;
  draw_square 200;
  draw_square 300;
  draw_square 400;
ENDPROC

PROC draw_square(num side_size)
  VAR robtarget p20;
  VAR robtarget p30;
  VAR robtarget p40;

  ! p20 is set to p10 with an offset on the y value
  p20 := Offs(p10, 0, side_size, 0);
  p30 := Offs(p10, side_size, side_size, 0);
  p40 := Offs(p10, side_size, 0, 0);

  MoveL p10, v200, fine, tPen;
  MoveL p20, v200, fine, tPen;
  MoveL p30, v200, fine, tPen;
  MoveL p40, v200, fine, tPen;
  MoveL p10, v200, fine, tPen;
ENDPROC
```


Continues on next page

Continued

Procedure arguments

When declaring a procedure, all arguments are declared inside parenthesis after the procedure name. This declaration contains data type and argument name for each argument. The argument gets its value from the procedure call and the argument acts as a variable inside the procedure (the argument cannot be used outside its procedure).

```
PROC main()
  my_procedure 14, "Hello", TRUE;
ENDPROC

PROC my_procedure(num nbr_times, string text, bool flag)
  ...
ENDPROC
```

Inside the procedure `my_procedure` above, `nbr_times` has the value 14, `text` has the value "Hello" and `flag` has the value `TRUE`.

When calling the procedure, the order of the arguments is important to give the right value to the right argument. No parenthesis are used in the procedure call.

Variables declared inside the procedure

Variables declared inside a procedure only exist inside that procedure. I.e. in the example above, `p10` can be used in all procedures in this module, but `p20`, `p30` and `p40` can only be used in the procedure `draw_square`.

3.2. Modules

About modules

A RAPID program can consist of one or several modules. Each module can contain one or several procedures.

The small and simple programs that are shown in this manual only use one module. In a more complex programming environment, some standard procedures, used by many different programs, can be placed in a separate module.

Example

The module `MainModule` contains code that is specific for this program and specifies what the robot should do in this particular program. The module `figures_module` contains standard code that can be used by any program that wants to draw a square, triangle or circle.

```
MODULE MainModule
  ...
  draw_square;
  ...
ENDMODULE

MODULE figures_module
  PROC draw_square()
     ...
  ENDPROC
  PROC draw_triangle()
     ...
  ENDPROC
  PROC draw_circle()
     ...
  ENDPROC
ENDMODULE
```

Program modules

A program module is saved with the file ending .mod, e.g. *figures_module.mod*.

It makes no difference for the robot controller if the program is written in several modules. The reason to use several program modules is only to make the program easier to grasp and easier to reuse for the programmers.

There can only be one program active on the robot controller, i.e. only one of the modules can contain a procedure named `main`.

System modules

A system module is saved with the file ending .sys, e.g. *system_data_module.sys*.

Data and procedures that should be kept in the system even if the program is changed should be placed in a system module. For example, if a persistent variable of type `tooldata` is declared in a system module, a recalibration of the tool is preserved even if a new program is loaded.

3.3. Structured design

About structure

When first confronting a problem that you want to solve with a RAPID program, sit down and analyze the problem and its components. If you start programming without first thinking through the design, your program will be irrational. A well designed program is less likely to contain errors and is easier for others to understand. The time spent on design is paid back many times in testing and maintenance of the program.

Break down the problem

Follow these steps to break down the problem into manageable parts:

	Action
1.	Identify larger functionality. Try to split the problem into smaller pieces that will be easier to handle.
2.	Create a design structure. Draw a map of the functionality and how they relate to each other.
3.	Look at each block in the design structure. Can a block be further split into smaller pieces? What is required to implement the block?

Example

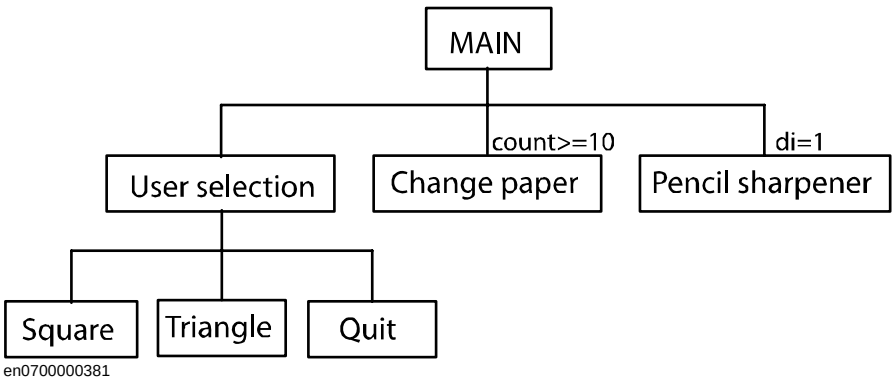
Problem description

Create a RAPID program that can draw squares or triangles on a piece of paper. Let the operator decide if it is a square or triangle that should be drawn next. When the robot is finished drawing the figure the user shall be able to make the same selection again until the operator taps on a Quit button.

When the robot has drawn 10 figures on the same paper, write a message that the paper should be replaced and wait for the operator to tap an OK button.

Between drawings, check if `di1` is 1. If it is, move to a pencil sharpener and set `do1` to 1 to start the sharpener and slowly move the pencil into the sharpener. Normally we would need to redefine the tool since it gets shorter when it is sharpened, but we will skip that step in this example.

Design structure



en0700000381

Continues on next page

Continued

Program code

```
MODULE MainModule
  PERS tooldata tPen := [ TRUE, [[200, 0, 30], [1, 0, 0 ,0]], [0.8,
       [62, 0, 17], [1, 0, 0, 0], 0, 0, 0]];
  CONST robtarget p10 := [ [600, -100, 800], [0.707170, 0,
       0.707170, 0], [0, 0, 0, 0], [ 9E9, 9E9, 9E9, 9E9, 9E9, 9E9,
       9E9] ];
  CONST robtarget pSharp1 := [ [200, 500, 850], [1, 0, 0, 0], [0,
       0, 0, 0], [ 9E9, 9E9, 9E9, 9E9, 9E9, 9E9, 9E9] ];
  PERS num count := 0;

  PROC main()
    user_selection;
    IF count >= 10 THEN
      change_paper;
      ! Reset count
      count := 0;
    ENDIF
    IF di=1 THEN
      sharpen_pencil;
    ENDIF
  ENDPROC

  PROC user_selection()
    VAR num answer;
    TPReadFK answer, "Select which figure to draw", "Square",
         "Triangle", "Quit", stEmpty, stEmpty;
    IF answer = 1 THEN
      draw_square;
      count := count + 1;
    ELSEIF answer = 2 THEN
      draw_triangle;
      count := count + 1;
    ELSE
      quit;
    ENDIF
  ENDPROC
```


Continues on next page

Continued

```
PROC draw_square()
   VAR robtarget p20;
   VAR robtarget p30;
   VAR robtarget p40;

   ! Define points that give a square with the side 200 mm
   p20 := Offs(p10, 0, 200, 0);
   p30 := Offs(p10, 200, 200, 0);
   p40 := Offs(p10, 200, 0, 0);

   MoveL p10, v200, fine, tPen;
   MoveL p20, v200, fine, tPen;
   MoveL p30, v200, fine, tPen;
   MoveL p40, v200, fine, tPen;
   MoveL p10, v200, fine, tPen;
ENDPROC

PROC draw_triangle()
   VAR robtarget p20;
   VAR robtarget p30;

   ! Define points for the triangle
   p20 := Offs(p10, 0, 200, 0);
   p30 := Offs(p10, 200, 100, 0);

   MoveL p10, v200, fine, tPen;
   MoveL p20, v200, fine, tPen;
   MoveL p30, v200, fine, tPen;
   MoveL p10, v200, fine, tPen;
ENDPROC

PROC quit()
   TPWrite "Good bye!"
   ! Terminate the program
   EXIT;
ENDPROC

PROC change_paper()
   VAR num answer;
   TPReadFK answer, "Change the paper", "OK", stEmpty, stEmpty,
        stEmpty, stEmpty;
ENDPROC
```


Continues on next page

Continued

```
  PROC sharpen_pencil()
    VAR robtarget pSharp2;
    VAR robtarget pSharp3;

    pSharp2 := Offs(pSharp1, 100, 0, 0);
    pSharp3 := Offs(pSharp1, 120, 0, 0);
    ! Move quickly to position in front of sharpener
    MoveJ pSharp1, vmax, z10, tPen;
    ! Place pencil in sharpener
    MoveL pSharp2, v500, fine, tPen;
    ! Start the sharpener
    SetDO do1, 1;
    ! Slowly move into the sharpener
    MoveL pSharp3, v5, fine, tPen;
    ! Turn off sharpener
    SetDO do1, 0;
    ! Move out of sharpener
    MoveL pSharp1, v500, fine, tPen;
  ENDPROC
ENDMODULE
```

Note that in production a program is normally run in continuous mode, so that when the execution reaches the end of the `main` procedure it starts from the beginning again. If this is not used, a `WHILE` loop can be used to repeat everything inside the `main` procedure.

3.3. Structured design

4 Data with multiple values

4.1. Arrays

What is an array

An array is a variable that contains more than one value. An index is used to indicate one of the values.

Declaring an array

The declaration of an array looks like any other variable, except that the length of the array is specified inside { }.

```
VAR num my_array{3};
```

Assigning values

An array can be assigned all its values at once. When assigning the whole array the values are surrounded by [ ] and separated by commas.

```
my_array := [5, 10, 7];
```

It is also possible to assign a value to one of the elements in an array. Which element to assign a value to is specified inside { }.

```
my_array{3} := 8;
```

Example

This example use a `FOR` loop and arrays to ask the operator for the estimated production time for each part. It is a very efficient way to write code compared to having one variable for each part and not be able to use the `FOR` loop.

```
VAR num time{3};
VAR string part{3} := ["Shaft", "Pipe", "Cylinder"];
VAR num answer;

PROC main()
  FOR i FROM 1 TO 3 DO
     TPReadNum answer, "Estimated time for " + part{i} + "?";
     time{i} := answer;
  ENDFOR
ENDPROC
```

4.2. Composite data types

What is a composite data type

A composite data type is a data type that contains more than one value. It is declared as a normal variable but contains a predefined number of values.

pos

A simple example of a composite data type is the data type `pos`. It contains three numerical values (x, y and z).

The declaration looks like a simple variable:

```
VAR pos pos1;
```

Assigning all values is done like with an array:

```
pos1 := [600, 100, 800];
```

The different components have names instead of numbers. The components in `pos` are named `x`, `y` and `z`. The value in one component is identified with the variable name, a point and the component name:

```
pos1.z := 850;
```

orient

The data type `orient` specifies the orientation of the tool. The orientation is specified by four numerical values, named `q1`, `q2`, `q3` and `q4`.

```
VAR orient orient1 := [1, 0, 0, 0];
TPWrite "The value of q1 is " \Num:=orient1.q1;
```

pose

A data type can be composed of other composite data types. An example of this is the data type `pose`, which consists of one `pos` named `trans` and one `orient` named `rot`.

```
VAR pose pose1 := [[600, 100, 800], [1, 0, 0, 0]];
VAR pos pos1 := [650, 100, 850];
VAR orient orient1;
pose1.pos := pos1;
orient1 := pose1.rot;
pose1.pos.z := 875;
```

robtarget

`robtarget` is too complex a data type to explain in detail here, so we will settle for a brief explanation.

`robtarget` consists of four parts:

Data type	Name	Description
pos	trans	x, y and z coordinates
orient	rot	Orientation
confdata	robconf	Specifies robot axes angles
extjoint	extax	Specifies positions for up to 6 additional axes. The value is set to 9E9 where no additional axis is used.

Continues on next page

Continued

```
VAR robtarget p10 := [ [600, -100, 800], [0.707170, 0, 0.707170,
     0], [0, 0, 0, 0], [ 9E9, 9E9, 9E9, 9E9, 9E9, 9E9] ];

! Increase the x coordinate with 50
p10.trans.x := p10.trans.x + 50;
```

Detailed descriptions

Detailed descriptions of these data types and many more can be found in *Technical reference manual - RAPID Instructions, Functions and Data types*, section *Data types*.

4.2. Composite data types

5 RAPID instructions and functions

5.1. Instructions

What is an instruction

A RAPID instruction acts as a premade procedure. An instruction call looks like a procedure call with the instruction name followed by argument values.

Some RAPID instructions are simple and could easily have been written as a procedure in RAPID. For example the instruction `Add`.

```
Add reg1, 3;
! The same functionality could be written:
reg1 := reg1 + 3;
```

Other RAPID instructions perform complicated processes that could not have been programmed without these premade instructions. For example `MoveL`, which may seem like a simple instruction but in the background there are calculations of how much to move each robot axis and how much current each motor should have. Because the program code for these calculations is already made, all you have to do is write a simple instruction call.

```
MoveL p10, v1000, fine, tool0;
```

Detailed descriptions

Detailed descriptions of instructions can be found in *Technical reference manual - RAPID Instructions, Functions and Data types*, section *Instructions*.

5.2. Functions

What is a function

A RAPID function is similar to an instruction but returns a value.

```
! Calculate the cosine of reg2
reg1 : = Cos(reg2);
```

Since the function returns a value, the result of the function can be assigned to a variable.

The arguments in a function call are written inside parenthesis and are separated with commas.

Include a function call in a statement

Anywhere, where a value can be used, a function returning a value of the same data type can be used.

```
! Perform something if reg1 is smaller than -2 or greater than 2
IF Abs(reg1) > 2 THEN
...

! Convert the num time to string and concatenate with other strings
string1 := name + "'s time was " + NumToStr(time);
```

Simplify complicated calculations

A single function call can often replace several complex statements.

For example:

```
p20 := Offs(p10, 100, 200, 300);
```

can replace the following code:

```
p20 := p10;
p20.trans.x := p20.trans.x + 100;
p20.trans.y := p20.trans.y + 200;
p20.trans.z := p20.trans.z + 300;
```

Detailed descriptions

Detailed descriptions of functions can be found in *Technical reference manual - RAPID Instructions, Functions and Data types*, section *Functions*.

6 What to read next

6.1. Where to find more information

What to find in which manual

What do you want to know	Where to read about it
• How to write programs on the FlexPendant • How to load programs to the robot controller • How to test the program	*Operating manual - IRC5 with FlexPendant*, section *Programming and testing*
• More detailed information about the functionality mentioned in this manual • What instructions are there for a specific category (e.g. move instructions or clock functionality) • Descriptions of more advanced functionality (e.g. interrupts or error handling)	*Technical reference manual - RAPID overview*
• Information about a specific instruction, function or data type	*Technical reference manual - RAPID Instructions, Functions and Data types*
• Details about how the robot controller handles different parts of RAPID	*Technical reference manual - RAPID kernel*

6 What to read next

6.1. Where to find more information

Contact us

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

3HAC029364-001 Rev -, en