

PROCESS-PLC

Programming Manual

JETTER GmbH
Gräterstraße 2
D-71642 Ludwigsburg
Tel +49 7141 2550 0
Fax +49 7141 2550 425
Hotline +49 7141 2550 444
E-Mail jetter@jetter.de
Internet www.jetter.de



Edition 1.2
February 1999

JETER GmbH reserves the right to make alterations to its products in the interest of technical progress. These alterations need not be documented in every single case.

This manual and the information contained herein has been compiled with the necessary care. JETER GmbH makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. JETER GmbH shall not be liable for errors contained herein or for incidental or consequential damage in connection with the furnishing, performance, or use of this material.

The brand names and product names used in this hardware description are trade marks or registered trade marks of the respective title owner.

Table of Contents

I. SYMPAS PROGRAMMING ENVIRONMENT	5
1. Survey	5
2. The SYMPAS System	7
2.1 Hardware (Requirements)	7
2.2 Software	7
2.3 Hardware Installation	8
2.4 Software Installation	9
2.4 SYMPAS for Several Networked Controllers (JETWay-H)	12
3. Operation of the SYMPAS Programming Environment	16
3.1. Starting of the SYMPAS Programming Environment	16
3.2 Description of the Screens	17
3.3 Program Input	19
3.3.1 Keys and Functions in the Program Editor	25
3.3.2 Program Transfer	28
3.4 The Setup Screen (Setup Mode)	29
3.4.1 Keys and Functions in the Setup Screen	30
3.4.2 Description of the Fields	31
3.5 Description of the Menus	37
3.5.1 Keys and Functions in the Pull-Down Menus	37
3.5.2 The "Project" Menu	38
3.5.3 The "File" Menu	42
3.5.4 The "Edit" Menu	46
3.5.5 The "Block" Menu	48
3.5.6 The "Transfer" Menu	51
3.5.7 The "Listing" Menu	57
3.5.8 The "Monitor" Menu	59
3.5.9 The "Scope" Menu	61
3.5.10 The "Special" Menu	68
3.6 Symbolic Programming - the Symbol Editor	77
3.6.1 Keys and Functions in the Symbol Editor	78
3.6.2 Creating a Symbol File (in the Symbol Editor)	81
3.7 INCLUDE Files	84
3.7.1 INCLUDE Files in the Program Editor	84
3.7.2 INCLUDE Files in the Symbol Editor	87
3.8 Error Messages	90
3.9 Files, Extensions, etc.	100
3.10 Miscellaneous	103
3.10.1 Indirect Addressing	103
3.10.2 Commentaries	103
3.10.3 Call-up by the /o Switch (Laptop, Notebook)	104
3.10.4 The NOSYMPAS.EXE Program	105
3.10.5 Switching to DOS	106
3.10.6 Password	106
3.10.7 SYMPAS Version 3.09 ff, and MIKRO up to 2.10	107
3.10.8 SYMPAS and PASE-J (up to version 4.04)	107
3.10.9 SYMPAS in the Network (PASE-E up to version 4.04)	108
3.10.10 Further Command Line Parameters (Call-Up Switches)	108
II. SYMPAS PROGRAMMING	109

1. Overview	109
2. Fundamentals of Programming	110
2.1 Principles of Program Setup	110
2.1.1 Rules for Program Structure - Task Structure	116
2.1.2 Special Registers / Flags for Task Control	121
2.2 Symbolic Programming	124
2.2.2 Examples of Symbolic Notation	126
2.3 Remarks on the Program Examples	128
3. The Programming Language	129
3.1 Overview over Instructions	129
3.2 Basic Instructions	133
3.2.1 Waiting Condition <code>WHEN ... THEN</code>	133
3.2.2 Waiting Condition <code>WHEN_MAX ... THEN</code>	135
3.2.3 Branch Condition <code>IF ... THEN ... (ELSE)</code>	137
3.2.4 The <code>DELAY</code> Instruction	140
3.3 Boolean Expressions	143
3.3.1 Phrasing Elementary Conditions	144
3.3.2 Examples of Connected Expressions	149
3.4 Arithmetic Expressions	152
3.4.1 Numbers	153
3.4.2 Arithmetic Expressions	154
3.4.3 Assignment to Integer Registers	155
3.4.4 Assignment to a Floating Point Register	157
3.5 Tasks, Labels, Jumps and Subroutines	160
3.5.1 Tasks, Flags and Jumps	160
3.5.2 Subroutines	163
3.5.3 Functions	168
3.6 Registers and Flags	173
3.6.1 Basic Information on Registers	174
3.6.2 Instructions for Register Loading	178
3.6.3 Calculating with Registers	185
3.6.4 Register Bit Instructions	189
3.6.5 Flags and Flag Instructions	192
3.7 Inputs and Outputs	195
3.7.1 Inputs	195
3.7.2 Outputs	197
3.8 Display Instructions and User Input	200
3.8.1 Display of Texts	200
3.8.2 Display of Register Contents	204
3.8.3 Reading of Register Values by the Program	208
3.8.4 Special Registers for User Input	210
3.9 Instructions for Axis Controlling	220
3.9.1 Positioning	220
3.9.2 Enquiries on the Present Condition	228
3.10 Task Instructions	229
3.10.1 Taskbreak	229
3.10.2 Taskcontinue	230
3.10.3 Taskrestart	230
3.10.4 Examples of the Task Instructions	231
3.11 Various Instructions	232
3.11.1 Time Instructions	232
3.11.2 <code>NOP</code>	237
3.11.3 The Commentary Character	237
3.11.4 Special Functions	238
3.11.5 The <code>LIMITS</code> Instruction	241

3.11.6 Word Processing	242
3.12 Network Instructions	246
3.12.1 Sending Register Values to Slave Controllers	247
3.12.2 Getting Register Values from a Slave Controller	248
3.12.3 Network Operation by 50000er Numbers	251
3.12.4 Special Registers / Flags for Network Operation	260
4. Description of the Memory	262
4.1 Basics on Registers and Flags	262
4.1.1 Registers	262
4.1.2 Flags	272
5. Realtime Clock	273
5.1 Overview, Function	273
5.2 Register Description	274
5.3 Realtime Clock: An Exemplary Program	275
REGISTER DESCRIPTION	275
EXEMPLARY PROGRAM: REALTIME CLOCK	276
6. Demonstrating Example: Handling-System	277
6.1 Problem Description	277
6.2 Flow Charts of the Three Tasks	279
6.2.1 TASK 0 - Control Task	279
6.2.2 TASK 1 - Automatic Task	280
6.2.3 TASK 2 - Display Task	281
6.3 Program Listing	282
6.4 Symbol Listing	291
INDEX	295

I. SYMPAS Programming Environment



1. Survey

The stages of program development are supported

SYMPAS is the programming environment for PROCESS-PLC programs. With the help of this programming software, problems originating from a process that should be controlled, can be directly expressed in a SYMPAS program for all PROCESS-PLC control systems. All important stages of program development - from editing via syntax check, up to transfer into the controller and setup in the integrated setup mode are supported by the SYMPAS programming environment.

Hardware requirements:
PC, IBM compatible

As a hardware for the use of SYMPAS an IBM compatible personal computer will be needed. The PC serves for data input as well as monitoring the program flow and the register conditions during the setup stage.

PROCESS-PLC programs and register sets can be stored on, and read from, hard or floppy disk drives.

The personal computer will be needed, until the program has been transferred to the controller (any register sets included) and tested successfully. After that, the PC can be used again for other tasks.

Menu and window structure

In the SYMPAS pull-down menu and window structure of SYMPAS, maximum clarity has been combined with user friendly operation. To grant the professional user the possibility of swift working, the most important functions can also be accessed by hotkeys.

**Call up help
by pressing
(F1)**

Context sensitive information has been provided by the help text that is always displayed in the status line, and by the help windows, that can be activated by the F1 function key.

2. The SYMPAS System

2.1 Hardware (Requirements)

The requirements for the use of the SYMPAS programming environment are:

- An IBM compatible personal computer with at least 512 kByte RAM and 2 disk drives (or 1 disk drive plus hard disk) and a DOS operating system.
- one serial interface (COM1 or COM2).
- one programming cable EM-PK connecting the PC with the controller.
- A PROCESS-PLC controller - PASE-E, DELTA, NANO, or MIKRO.

2.2 Software

**Up-to-date
information in
the
README file**

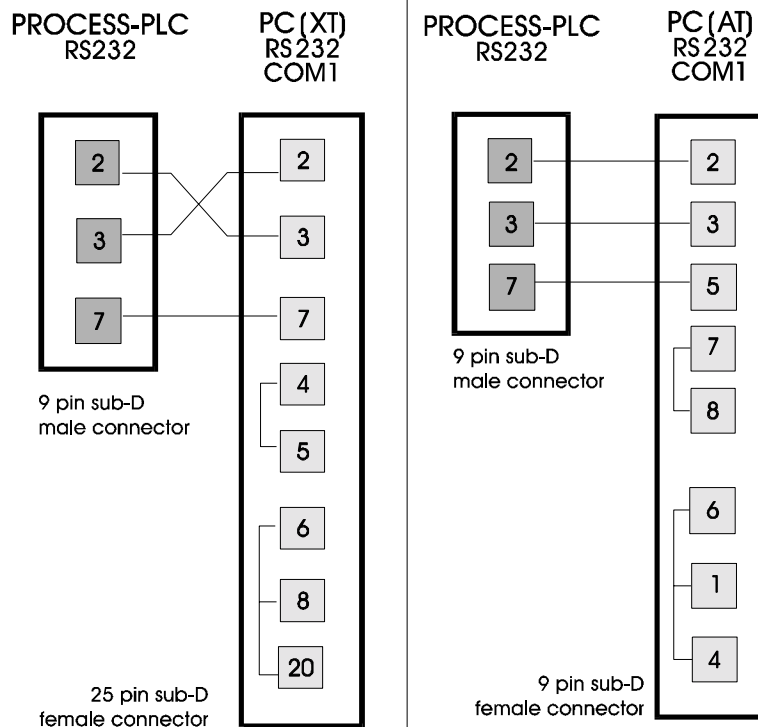
A survey of the available files can be taken from the README file, which should be read by the user in any case, as it contains the latest important information, which cannot be found in the manual. This file will be displayed on the screen by giving the instruction TYPE A:README; it will be printed by giving the DOS command PRINT A:README.

2.3 Hardware Installation

To use SYMPAS together with a PROCESS-PLC, connection to the serial interface (COM1 or COM2) of the PC has to be established. The interface can be configured in the context of the SYMPAS programming environment. For XT compatible systems, a 25-pin sub-D male connector for COM1, for AT compatible systems, a 9-pin sub-D male connector for COM1 has been provided. The connection cable EM-PK has to be used.

Programming cable EM-PK

The programming cable EM-PK can be produced according to the following figure:



2.4 Software Installation



The SYMPAS programming environment is being installed By INSTALL.EXE

The software installation is carried out by the INSTALL.EXE program. A subdirectory called SYMPAS is opened on floppy disk or hard disk (default setting) by this program. All important files are copied into this directory. Which files and subdirectories are to be copied is determined by the configuration, which can be defined in the configuration window shown above before the actual installation process. For installation, write the line

A:\INSTALL or B:\INSTALL.

Using the cursor keys ↓ and ↑, the menu line can be selected, and by pressing the RETURN key↵, this selected line can be changed. When the basic configuration has been chosen, the installation process can be started by pressing function key F9. The following definitions can be made under these selection lines:

Controller Type

Here, a choice can be made between the PASE-E, DELTA, NANO, and MIKRO controllers. Independent from the installation, the controller type can be selected anew any time in the SYMPAS programming environment.

Destination Directory

Here, the entire destination path can be defined. If the programming environment is to be installed in another subdirectory, this line has to be edited correspondingly, for example:

```
C:\DIRECTORY
```

By this instruction, SYMPAS is installed in the subdirectory "Directory" on the C: disk.

After selecting the line with the cursor key, a window will be opened by the ↵ ENTER key, where the destination path can be edited.

Copy Tools?

Here, the installation of the available tools can be determined. These tools have been documented in the README file and can be attributed to an individual directory.

Language

Here, the language to operate the programming environment with can be chosen. A selection can be made between German and English. Even after installation, the dialogue, as well as the programming language can be changed any time in SYMPAS itself.

Start installation by pressing (F9)

By pressing the function key F9, the installation process will be started and carried out according to the definitions that have been made.

2.4 SYMPAS for Several Networked Controllers (JETWay-H)

JETWay-H:
126 participants
115 kBaud



The following advantages are granted by using the JETWay-H interface as a programming interface instead of the RS232 interface:

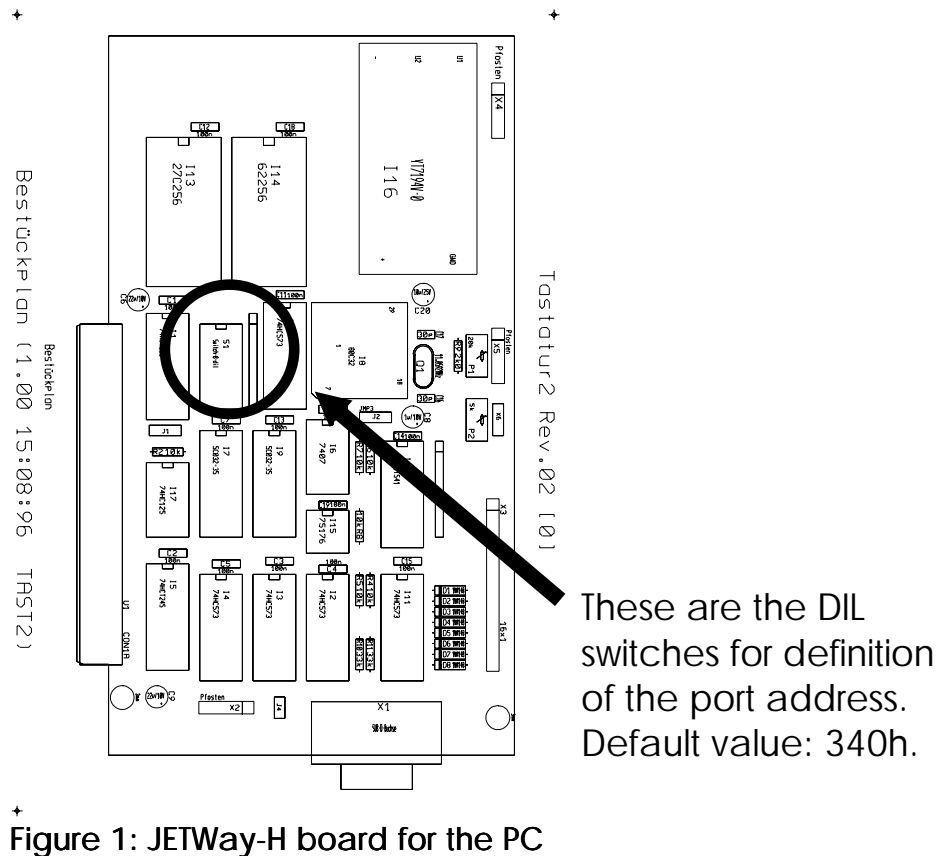
- Up to 126 PROCESS-PLC can be addressed from one SYMPAS desktop.
- Transfer rates of up to 115 kBaud can be realised.
- Greater distances are possible.

JETWay-H Cable		
Connection on the PROCESS-PLC	Shielding	Specification max. Length
9 pin sub-D male connector or 15 pin sub-D male connector	<p>Please shield extensively! Only use metallised housings!</p>	RS485 max. cable length: 400m
Pin	Signal	Pin
7	Gnd	7
8	Data +	8
9	Data -	9

The JETWay-H Board for the PC



The connection between SYMPAS and up to 126 PROCESS-PLC controllers via JETWay H can be established using the PC board shown below.



AUTOEXEC.BAT

Into the AUTOEXEC.BAT of your PC the following line is to be written (only, if default setting is used):

```
SET JETWAY_PORT=340h
```

DIL Switches

If you want to, or have to, use another port address, this is possible by the DIL switches shown above on the JETWay-H.

DIL Switches on the JETWay-H Board						
Port	Switch 2	Switch 3	Switch 4	Switch 5	Switch 6	Switch 7
300h	OFF	OFF	ON	ON	ON	ON
310h	OFF	OFF	ON	ON	ON	OFF
320h	OFF	OFF	ON	ON	OFF	ON
330h	OFF	OFF	ON	ON	OFF	OFF
340h ^{*)}	OFF	OFF	ON	OFF	ON	ON
350h	OFF	OFF	ON	OFF	ON	OFF
360h	OFF	OFF	ON	OFF	OFF	ON
^{*)} Default setting						

The opposite
line must be
written into the
AUTOEXEC.BAT

Correspondingly, the line in the AUTOEXEC.BAT has to be changed:

```
SET JETWAY_PORT=x
```


In the SYMPAS menu "Special / Settings" a choice can be made between the programming interface via RS232 and via JETWay-H.

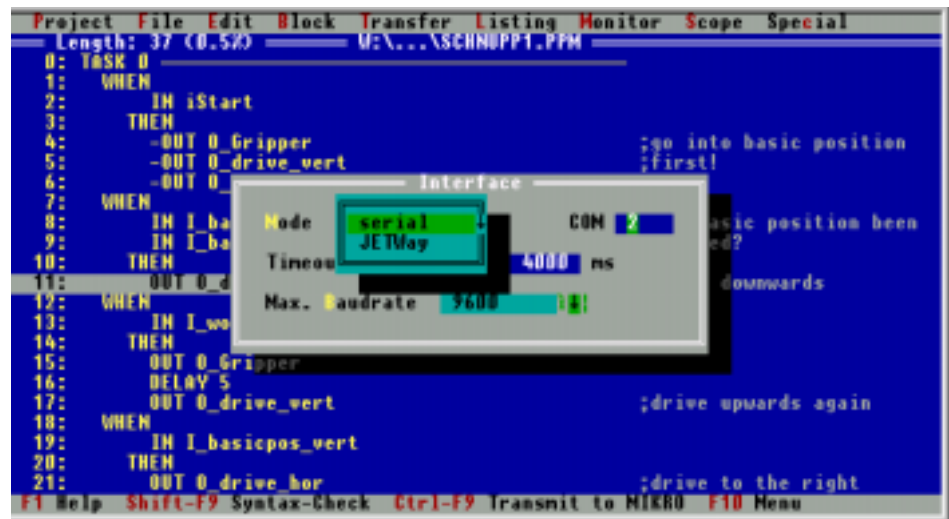


Figure 2: SYMPAS Menu: Special / Interface



Note:

For making this cable, the following minimum requirements have to be met:

Number of wires:	3
Diameter:	0,25 ²
Male connector	SUB-D, metallised
Shielding:	total, not in pairs

The shield needs extensive contact to the plug housings on both sides.

3. Operation of the SYMPAS Programming Environment

3.1. Starting of the SYMPAS Programming Environment

After the software has been installed, following the instructions in *Chapter 2.4 Software Installation*, the programming environment can be started by entering SYMPAS.

```
C:\SYMPAS>SYMPAS
```

By this instruction, the SYMPAS programming environment is started.

It might be helpful to create another subdirectory in the SYMPAS subdirectory, e.g. PROJECT1:

Create a meaningful structure of subdirectories

```
C:\SYMPAS\PROJECT1>
```

Further, the respective path definition for the SYMPAS call-up is to be written into AUTOEXEC.BAT.

Now, SYMPAS can be started, for example, the following way:

Start SYMPAS

```
C:\SYMPAS\PROJECT1>SYMPAS
```

All kinds of information, files, etc, which refer to "Project1", are now filed in this subdirectory. That way, the overview - even over a great number of projects - will be maintained.

These three screens represent the global structure of the SYMPAS graphic user interface.

The menu line has been placed in the upper line of the respective screen. It is identical for all three screens, yet, some functions only refer to one specific screen (e.g. setup screen) and have no meaning for the other screens (dim display of the selection lines).

Call up the menu or the function by pressing the (ALT)key plus the highlighted letter

The menu line is activated by the F10 function key. Using the cursor keys ← and →, you can move in the menu line. Using the ↓ cursor key, the respective pull-down menu can be opened, where you can move with the ↑ and ↓ cursor keys.

The pull-down menu or the menu line can be left with the ESC key, which can also be used for terminating any other activated function.

All functions that can be called up from the three screens are supported by help texts. First, a context related help text can always be found in the status line, which is the bottom line of the screen. Extensive help information will appear in a special window after pressing function key F1. This information is also context related. The help windows are left by pressing ESC.

With the help of the program editor, the programs for PROCESS-PLC control systems are written. The symbol editor serves for defining the symbolism of a program. The setup screen, finally, helps testing and optimising the program in connection with the control system and the process to be controlled.

3.3 Program Input

Instruction input by gramma- logues

The important instructions of the SYMPAS programming language are written into the program editor by the two first letters of the instruction. A selection window of all SYMPAS instructions starting with letter "T" will appear by pressing "T". Now, by either pressing the cursor keys or "A" (the second letter of the word TASK), the TASK instruction can be activated. After writing the parameter number into a window, the instruction will appear in the programming editor screen. Following this pattern, the program is written.

By pressing the "?" key a window is opened, where all available instructions are listed up, and from where those instructions can be taken.

Exemplary creation of a NANO program

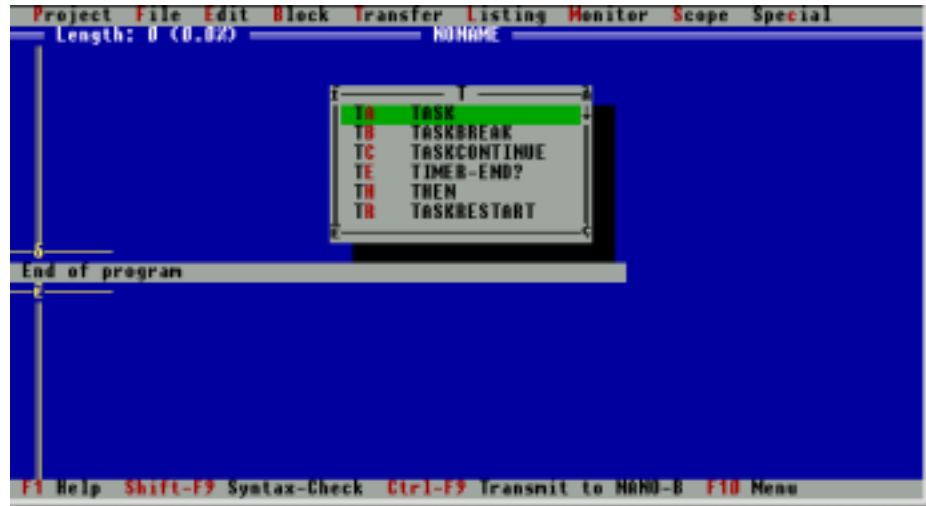
Following, an exemplary NANO program will be created.

After having started SYMPAS (*Chapter 3.1.*) the program editor screen will appear. First you open the "Project" pull-down menu by pressing ALT-P. By pressing cursor key ↓, the line "Edit Project Data" is selected and activated by ENTER<↓. Now a window is opened by SYMPAS, where the global project data are defined. Under the menu lines "program name", "customer/project", "place", "version" and "symbol file" the respective information can be written. After confirming the input of the last line "symbol file" with ENTER ↵, the window is closed. Now program input on the program editor screen can be started. Here the pattern of writing the grammalogue of an instruction to be integrated into the program text as briefly mentioned above will apply again.

First Instruction:

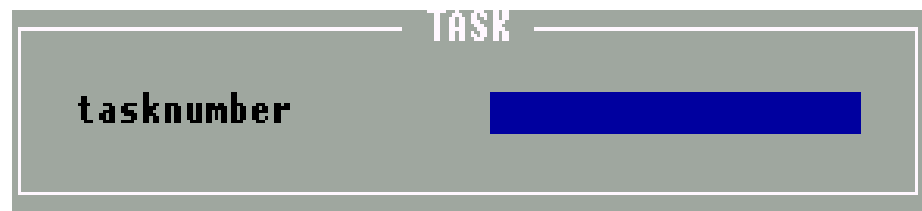
- Press the "T" key -> an input window of all instructions starting with letter "T" will appear.

(T) will open a window of all instructions starting with "T"



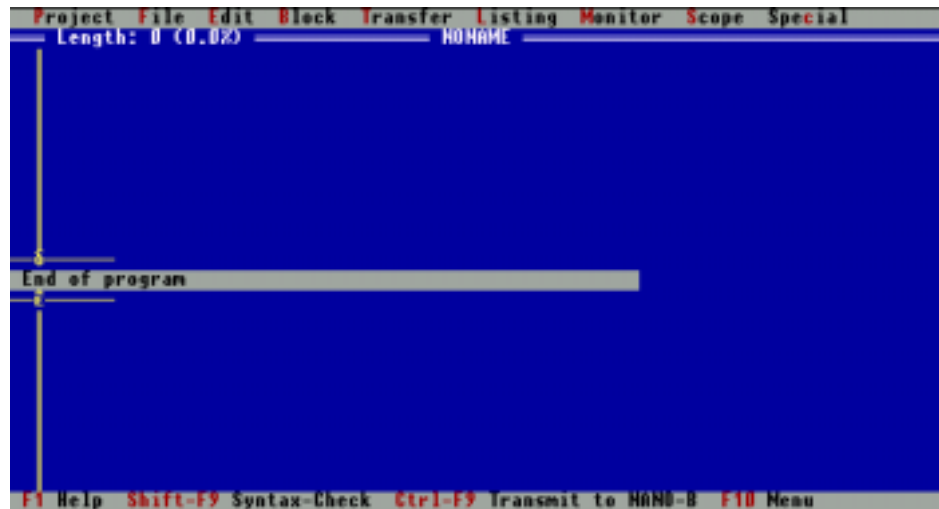
- Press the "A" key -> an input window to define the desired task number will appear

(A) will open a window where the task number is specified



- now press the "0" (zero) key and confirm by ENTER ↵ -> the TASK0 instruction will appear on the screen.

The `TASK 0` instruction will appear on the screen

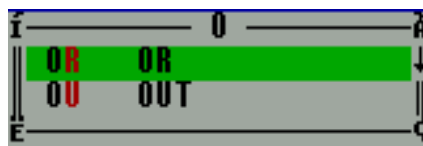


Note:
Each program **must** be started with the `TASK 0` instruction.

Second Instruction:

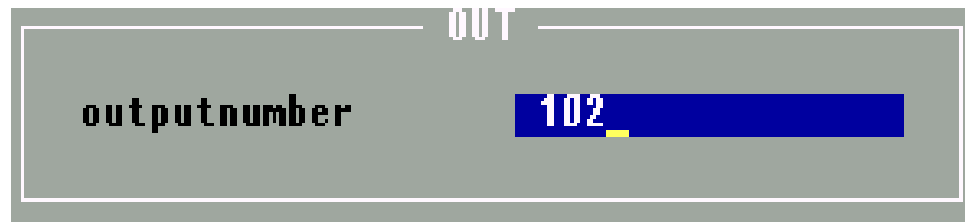
- Press the "O" key -> an input window of all instructions starting with letter "O" will appear.

(O) will open a window of all instructions starting with "O"



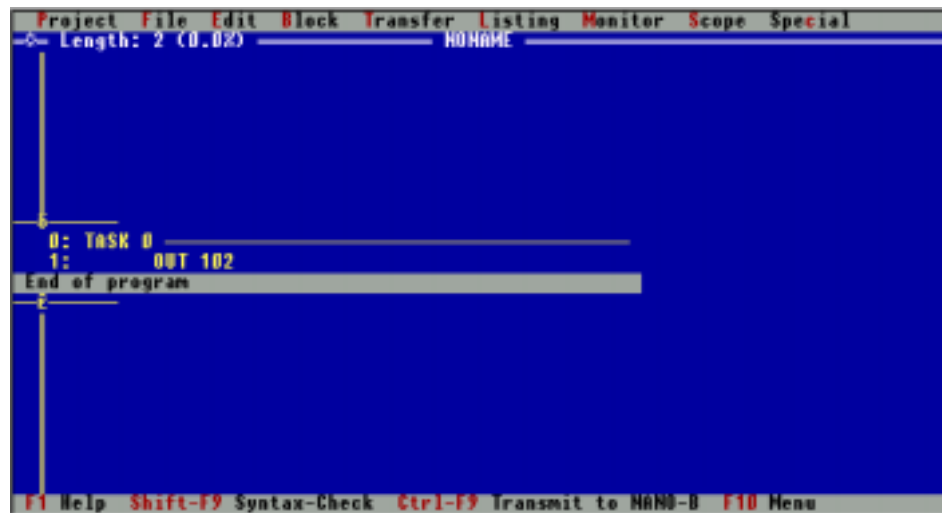
- Press the "U" key -> an input window for the output parameter will appear.

(U) will open a window where the output parameter is defined



- Now press keys "102" and confirm by ENTER ↵ -> the OUT 102 instruction (output 102) will appear on the screen. This instruction causes output 102 to be set or activated.

The OUT 102 instruction will appear on the screen



Third Instruction:

- Press the "D" and "E" keys -> an input window will appear, where the required delay can be input in multiples of 100 ms.
- Now input "10" and confirm by ENTER ↵ -> The DELAY instruction with parameter "10" will appear on the screen. This instruction causes the controller to delay for 1 second and then to continue with the further program execution.

Fourth Instruction:

- Press the "O" and "U" key -> an input window to define the desired output number will appear.
- This time you input "-102" and confirm by ENTER ↵ -> the -OUT 102 instruction (-OUTPUT 102) will appear on the screen. This instruction causes output 102 to be reset.

Fifth Instruction

- Press the "D" and "E" keys -> an input window will appear, where the delay time can be defined.
- Confirm the default value of the last instruction input "10" -> this instruction causes the controller to once more delay for 1 second.

Sixth Instruction:

- Press the "G" and "O" key -> an input window will appear where the goto destination (task or label) can be defined which the program is to branch out to.
- Input "0" and confirm by ENTER ↵ -> the GOTO 0 instruction will appear on the screen. The program run will return to TASK0, thus, the program will form an endless loop.

**Note:**

Each program task **must** be closed in itself by a GOTO instruction.

Now the program text input is terminated. Using the cursor keys ↑ and ↓ you can move in the program text. Program lines can be erased by pressing the DEL key. Automatic input can be made over the actual line, which is marked by the cursor. With the help of cursor keys ← and →, a selection between program text and commentary can be made.

3.3.1 Keys and Functions in the Program Editor

Cursor Movement:

Key:	Function:
cursor up	one line back
cursor down	one line forward
page up	page back
page down	page forward
Ctrl-page up	to top of program
Ctrl-page down	to end of program
cursor left	instruction range
cursor right	commentary range

Editor Instructions:

Key:	Function:
A..Z	An instruction is directly activated, if its first letter appears in the instruction list only once. Otherwise a selection window for the required instruction is offered. There is also the possibility to integrate an instruction into the program text by input of its first two letters.
?	A complete instruction list is offered as a selection window.
SPACE	The instruction that has been input last will be repeated.
ENTER	Edit the parameters of the actual instruction.
BS(←)	Delete the instruction preceding the actual instruction line.
DEL	Delete current instruction.

Block Operations:

Key:	Function:
Ctrl K B	mark top of block
Ctrl K K	mark end of block
Ctrl K V	move block
Ctrl K C	copy block
Ctrl K Y	delete block
Ctrl K R	load block from disk
Ctrl K W	save block to disk
Ctrl K P I	print block
Ctrl K H	switch off block
Ctrl K L	mark line
Ctrl Q B	find top of block
Ctrl Q K	find end of block

Storage of Cursor Position:

Key:	Function:
Ctrl-K 0..9	store cursor position 0 to 9 in the program text.
Ctrl-Q 0..9	go to stored cursor positions 0 to 9.

Miscellaneous:

Key:	Function:
Ctrl S	The symbol parameters of the current line are displayed, until the Ctrl key is released.
Ctrl M	The variable content of the current line is displayed, until the Ctrl key is released.

3.3.2 Program Transfer

Store the program by pressing (F2)

Before the program can be run in the controller, storing, e.g. on hard disk, should be made. By pressing ALT-D the pull-down menu "File" can be opened. By pressing the cursor key ↓ you will find the "save" selection line. Then the "save" procedure can be triggered by pressing ENTER ↵ (or quicker by function key F2). "Save" will be changed into "save as", when the file name has not been defined before. To load a program from hard disk into the program editor, the selection line "open..." in the same pull-down menu must be used.

Transfer the program to the controller by pressing (CTRL) (F9)

After the program has been input, it can be transferred to the controller by pressing CTRL-F9. This will trigger three functions: First, the program will be transferred to the controller and started there. Secondly, the screen will change to "setup" screen. Thirdly, an acoustic start signal can be heard.

Starting with *Chapter 3. Operation of the SYMPAS Programming Environment* you created a PROCESS-PLC program in the program editor, saved it on hard disk, transferred it to the controller and started it there automatically. The LED at input 102 will now function as a flashing signal with a constant signal time of one second.

By pressing function key F4, you will automatically get back to the program editor, e.g. to make changes in the program. Or else you will stay in the setup screen, in order to test a program for proper functioning. Such a program will surely be more complex than the flashing light that has just been programmed. Then the setup mode will support SYMPAS effectively in testing the program related to the controller and the process that is to be controlled.

3.4 The Setup Screen (Setup Mode)



In the setup screen a great number of functions are offered, which are to support program setup in connection with the controller and the process that is to be controlled.

By pressing
(F1) call up
help

Here you will also find direct support in the status line, while in the help window (to be activated using function key F1) extensive, context specific help will be offered.

In the setup mode, inputs, outputs, register contents (as numbers or texts), and axis parameters can be displayed and modified. Further, the contents of the user interfaces (LCD9, LCD16, etc.), as well as the number of the program line just being operated can be displayed.

3.4.1 Keys and Functions in the Setup Screen

The field identification number for activation of the individual fields can either be taken from the status line or from the brackets after the field name.

In an active field, the following instructions can be given:

Key:	Function:
INS	insert a new line
DEL	delete a current line
ENTER	<ul style="list-style-type: none">• input of a new value• input of a new number
Cursor up	one input field back
Cursor down	one input field forward
Cursor right Cursor left	switch between number and value input (only inputs, outputs, flags, registers, bin registers, text registers)
Ctrl-cursor	selection of bits 0 to 23 in the "binreg" field
+	incrementation of the cursor value
-	decrementation of the cursor value

3.4.2 Description of the Fields

**Changes
between the
individual
fields
can be made**

Individual fields can be selected with the help of the field identification numbers. Thus, the input field can be activated by key "1", the output field by key "2", the field for flags by key "3", the field for registers by key "4", the index field by key "6", the field for the display contents by key 7, the binary register, in which the content of any register can be displayed in binary mode, by key "8", and finally the text register by key "9".

The following function fields are available:

Inputs Key 1

Outputs Key 2

Flags Key 3

Axes Key 5

Registers Key 4

Status Window Key 7

Display Key 7

BinReg Key 8

TextReg Key 9

Index Key 6

Input Field

Key (1)

- Press key "1" -> now the input field is doubly framed; this means, it is active.
Press key "Insert" -> a field will appear, where the number of the input to be displayed can be defined. Confirm by pressing ENTER ↵.

The same applies to flags.

**Output
Field****Key (2)**

- Press key "2" -> now the output field is doubly framed; this means, it is active.
Press the "Insert" key -> a field will appear, where the number of the output to be displayed can be defined. Confirm by pressing ENTER ↵. This procedure can be continued, until the field is filled with displayed outputs. With the help of the ↑ and ↓ cursor keys various outputs can be selected, with the help of the ← and → cursor keys, one can switch between output number and output status. The output can be set or reset by the "+" or "-" keys. The same way, the output numbers can be incremented or decremented using the "+" or "-" key.
With the help of the "Delete" key, the display of the output made by the cursor is deleted.
By the "Insert" key, an output can be inserted at the present cursor position.

The same applies to flags.

**Register
Field****Key (4)**

- In contrast to other fields, a value can be attributed to the registers in field 4. With the help of cursor keys ← and →, one can switch between register content and register number. If the cursor is positioned on the register content and ENTER ↵ is pressed, a field will appear, where the register content can be changed. After input of the new register value, confirm by ENTER ↵

Axis Field**Key (5)**

- After you have opened the axis field by key "5", press the "Insert" key. A field will appear, where the desired axis number can be input (confirm by ENTER ↵). After this, all parameters are set according to the axis condition. By pressing the cursor keys, certain parameters can be selected; by pressing ENTER ↵, the parameters can be edited in a field (confirm by pressing ENTER ↵).

Index
Field

Key (6)

- The conditions of individual tasks are displayed in the index field. Press the "Insert" key and input the number of the task that is to be displayed. Repeat this procedure, until all relevant tasks appear are displayed in the field.

The tasks are displayed according to the following pattern:

- the task number, as it has been defined by the user
- :
- the line number that is being operated in the task at the moment
- if applicable, a status description of the task, which is expressed by the following four signs:

- o "‡" DELAY; delay time defined in the program.
- o "i" Input; program is waiting for user input.
- o "M" WHEN_MAX
- o "⊥" Taskbreak; the parallel branch is interrupted at the moment.
- o "----" Error; the called-up task does not exist in the program.
- o "Err" invalid program line

**Remark:**

The index field is only functioning, when SYMPAS has not been left since program transfer; otherwise "-1" will be displayed.

Display Field**Key (7)**

- In order to activate the display field, press key "7". In this field it is shown what is displayed by the connected user interface (e.g. LCD9/10) at that moment.

Binreg Field**Key (8)**

- Call up the binreg field by pressing key "8". In this field, a register content can be displayed in binary form. With the help of key combination CTRL and one of the two cursor keys ← and →, an individual bit can be selected and modified (+ and -) The display of the slave module SV1 status register 10100 can be selected by pressing the "INSERT" key, followed by input of the desired register number and confirmation by ENTER ↵.

Text Register Field**Key (9)**

- The text-register field is activated by pressing key "9". After pressing the "INSERT" key, a register number is queried. Input register 200, for example. With the help of cursor keys ← and → you can switch between a register number and its corresponding input text. If the cursor is positioned on the input text, press ENTER ↵, in order to edit the text.
- By this function, dialog texts for VIADU'KT can be written, for example (maximum length 40 bit). The text can be stored in one of the registers starting from register 200. In Bits 0 to 7, information on the length of the texts, in Bits 8 to 15 status information, and then each character, will be stored in ASCII format (three characters per register).

If you display output 102 after starting *Chapter 3.3 Program Input* on the setup screen, you will be able to monitor the change of status every second. This way, the status of a great number of functions - even for complex processes - can be displayed, monitored, and also modified. The axis, input, output, flag, register, display, and any other conditions, can be visualised simultaneously.



- underneath the "project" menu line of the setup-screen there is a status display of the general function of the screen.

→

The rotating arrow indicates that the setup screen is active. If the displayed data do not change, it can be verified by the still rotating arrow, that the present conditions of the individual inputs, outputs, etc. are static, and that malfunctioning of the setup mode can be excluded.

Number

By the number behind the rotating arrow, the duration of a refresh cycle is displayed in 1/100 seconds. This is the time, which passes, until the state of all inputs, outputs, flags, registers, etc. have been realised in the display.

3.5 Description of the Menus

Here, the individual pull-down menus are described, which can be activated from the menu bar. The description is given in the order of the individual functions in the pull-down menus of the three screens. Basically, the pull-down menus are identical for all three screens - program editor, symbol-editor, and setup-screen. Some functions are only possible in connection with a certain screen and are thus displayed in grey colour, which means, they are not to be activated, on the other two screens.

3.5.1 Keys and Functions in the Pull-Down Menu

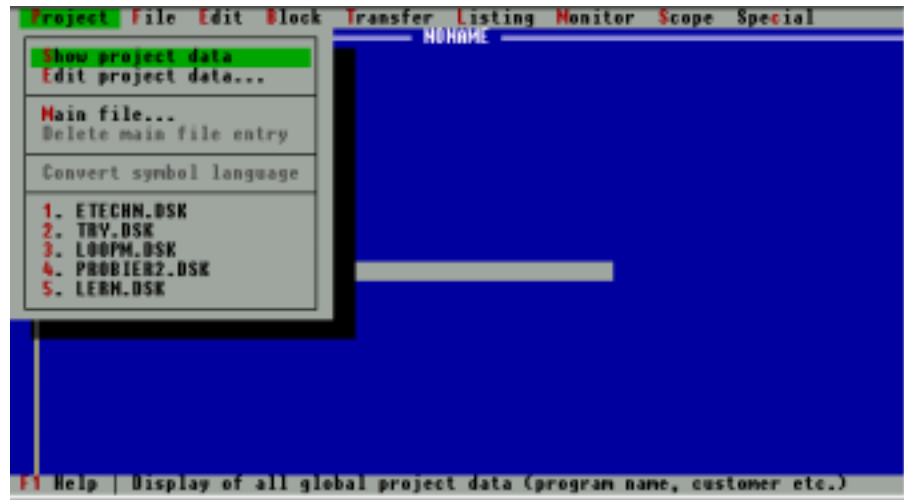
The following keys can be used to move in the menu bar and in the pull-down menus:

Key:	Function:
Cursor up	one menu line backwards
Cursor down	one menu line forward
Cursor left	one menu function backwards
Cursor right	one menu function forward
Home	first selection line
End	last selection line
ENTER <↵	activate function under cursor
ESC	terminate

By pressing a highlighted letter or a function key defined behind a menu line, the corresponding selection can be activated (hotkeys).

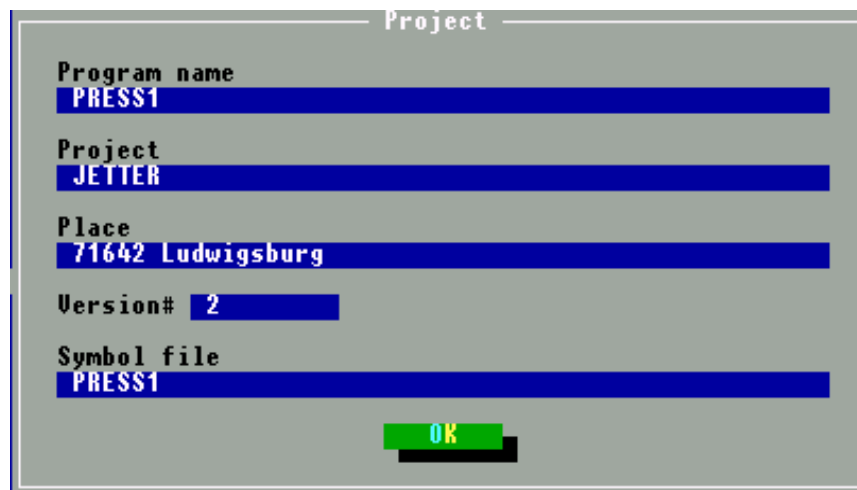
Selection lines are marked ([x]) by pressing the ALT key together with the highlighted key.

3.5.2 The "Project" Menu



Show Project Data

Under the headlines shown below, general information is given:



Edit Project Data

In this menu, the respective general information can be input (also see illustration above).

- Program name
- Customer/project
- Place
- Version
- Symbol file

The input is confirmed by pressing the ENTER ↵ key. Each time the program editor is activated, the version number is incremented. If this has not been desired, the version number can be reset by hand.

Main File

If INCLUDE files are to be integrated into the program text, a main file must be defined, where up to 32 INCLUDE files can be stored. The following line will appear in the program editor:

```
#INCLUDE NAME
```

The design of an INCLUDE file is the same as that of a common program file. Thus, already existing programs or program sequences can be assembled to one main file. Another advantage of working with INCLUDE files is the possibility to work with program sizes that could not be stored in a PC memory any more. By dividing up the program into various INCLUDE files which will again be logically combined in the main file, the restriction of memory space by the size of the PC memory can be bypassed. An extensive description can be found in *Chapter 3.7 INCLUDE Files*.



Note:

In the main file 32 INCLUDE files can be defined as a maximum.

Delete Main File Entry

The main file that has been defined in the "Main File..." selection is deleted.

Convert Symbol Language

The language used in the SYMPAS symbolism is converted into another language. For this purpose the alternative expression is given in square brackets in the symbol editor.

Convert Symbol Language

Before Converting

Define the alternative symbol in square brackets

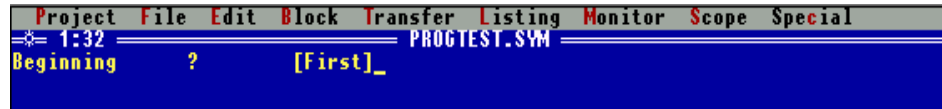


Figure 6: Symbol editor before converting

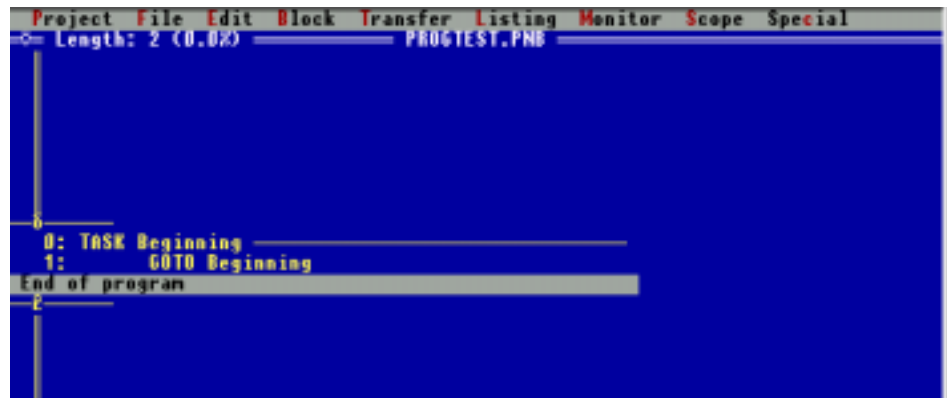


Figure 7: Program editor before converting

Start converting in the menu "Project / Symbol Language"

After Converting



Figure 7: symbol editor after converting

Select the programming language in the menu "Special / Settings"

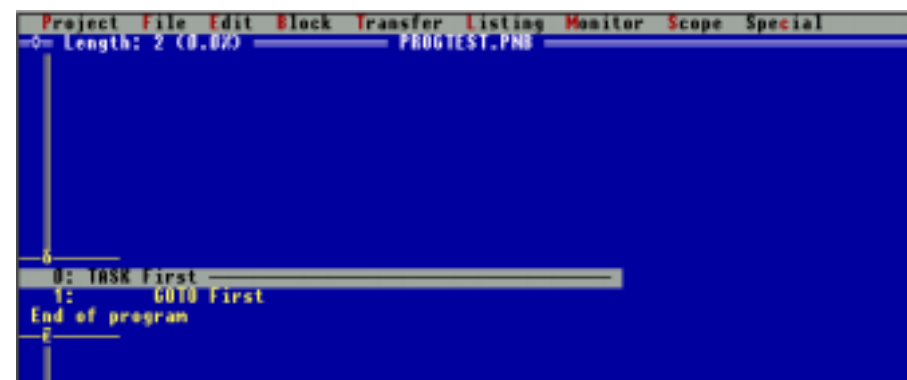
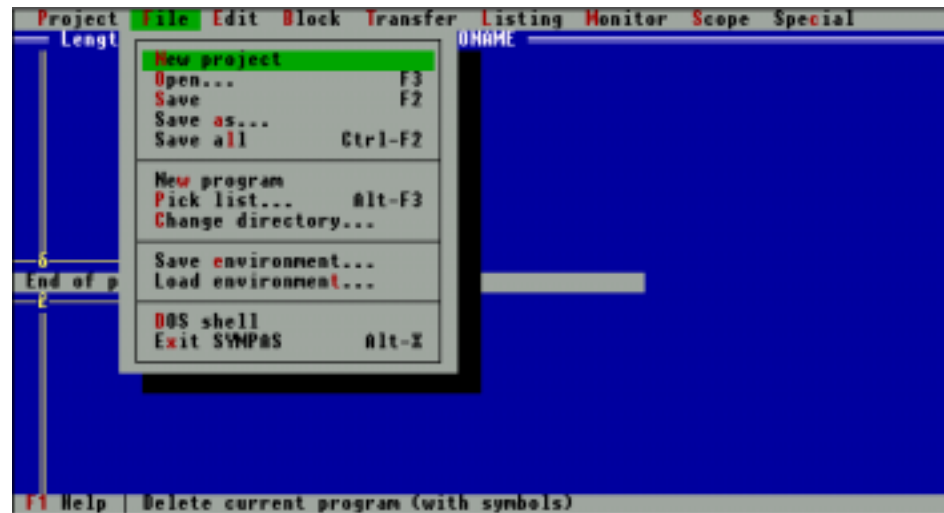


Figure 8: Program editor after converting

3.5.3 The "File" Menu



New Project

In order to start a new program, the program and symbol editor can be reset. The present content of the two editors will get lost by taking this step, thus it should be stored in advance. You will now find the same conditions as after a new SYMPAS start.

Open...

Program or symbol files are loaded from the disk or hard disk drive by this function. A window will appear, where the complete path and file name can be defined. If the window is ignored and RETURN ↵ is pressed, all available file names, among which a selection can be made with the help of the cursor keys (load by pressing ENTER ↵), will be displayed.

Save

By this function, the program or symbol editor is stored on the drive under the name that has been defined in the menu part "edit project data".

Save as...

By this function , the program editor, as well as the symbol editor, will be saved on the disk under any desired name. After "Save as ..." has been activated, a window will be opened, where the file name defined under "project data" will appear. This name (and path) can be confirmed by ENTER ↵, or else be modified; then the function can be ended by ENTER ↵. The defined name will be the actual name.

Save all

By this function, the program-, as well as the symbol editor will be stored on floppy or hard disk. If no symbol file has been defined in the "Edit project data" window, only the program editor will be stored by the settings from the "Edit project data ..." window.

New Program

The program in the program editor is deleted. The content of the symbol editor remains unchanged (Add INCLUDE-files to the project).

Pick-List...

32 file names can be loaded into a selection window as a maximum, or the corresponding files out of the window into the program editor.

A main file and up to 32 INCLUDE files can easily be managed by this pick list. SHIFT-F9, helps to switch between the two files which have last been worked with.

Change Directory...

This function helps to change the directory or drive.

Save Environment...

By this function, the following settings are saved under a definable file name. (Select the required file by pressing ENTER and activate by pressing ALT-W.)

Select
directory by
ENTER and
change
directory by
ALT-W

Program Editor:

- Program name
- Cursor position
- Block data
- Program label data
- Switch to "Display symbol parameter"
- Switch to "Monitor function"

Symbol Editor:

- Symbol file name
- Cursor position
- Block data
- Program labels

Setup Screen:

- here, the complete screen will be stored. The condition of all the windows up to the cursor position is kept in the memory.

The extension of the peripheral files is .DSK.

Load environment

A *.DSK file is loaded by this function. See "Store environment".

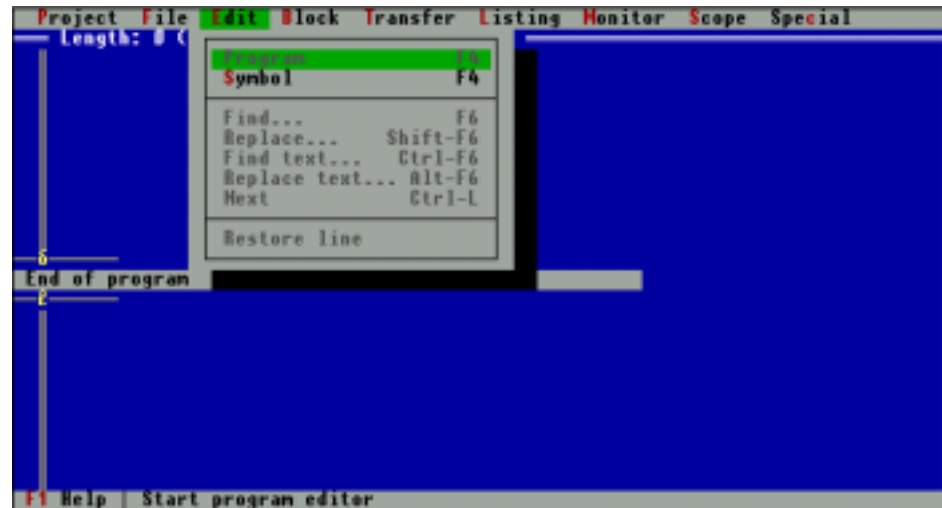
DOS shell

By this instruction SYMPAS is interrupted in order to operate on DOS surface, and from there one can return to SYMPAS by the DOS instruction EXIT.

Exit SYMPAS

By this function SYMPAS is terminated in order to return to the DOS level.

3.5.4 The "Edit" Menu



Program

By this instruction the program editor is activated.

Symbol

By this instruction the symbol editor is activated.

Find...

**Criteria for searching:
Instruction,
parameter
and line
number
included**

This function helps to find an instruction or a program line. After activating the function, a window will appear, where the expression to be found (Criteria for searching: Instruction, parameter or program line number included) can be defined. Indirect levels are not considered as criteria for searching here.

The search is carried out from the cursor position down to end of program.

Replace...

This instruction is connected with the above search function. First the command is analogue to the searching command mentioned above, yet, another input line is offered, where the expression is written, by which the searching expression can be replaced. Indirect levels cannot be used.

Find Text...

This function helps to search for texts, which can either be found in the commentary ranges or are parameter components of certain instructions (e.g. DISPLAY_TEXT).

Replace Text...

This instruction is connected with the search function mentioned above. Further, an expression can be defined, by which the search expression is to be replaced..

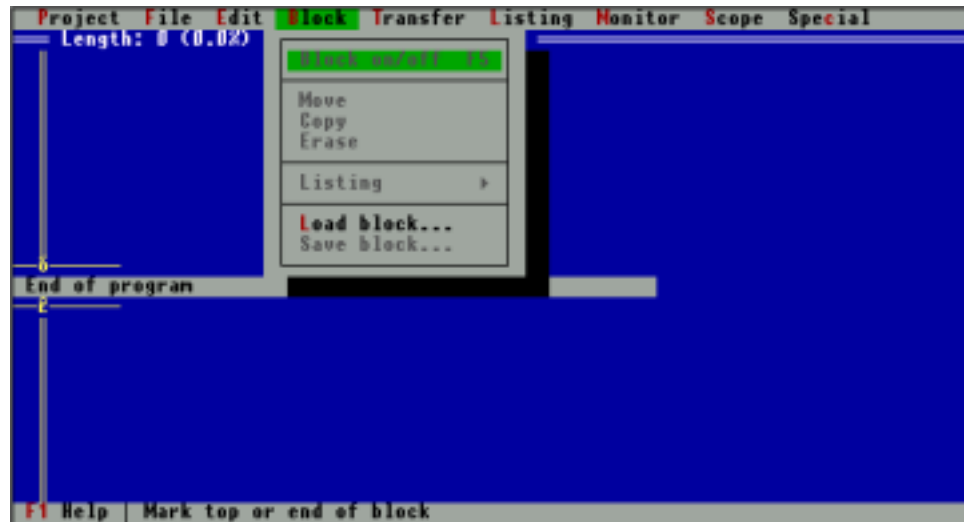
Next

This instruction also relates to the search function. Further, an expression can be defined, by which the search expression is to be replaced.

Restore Line

The line deleted last is restored by this instruction.

3.5.5 The "Block" Menu



Block on/off

Here the block function can be switched on or off. After activating, the block can be marked using cursor key ↓, starting from the present cursor position. After this, the marking mode is deactivated by repeated call-up of this function. Then the block can be used for further operation.

Move (Ctrl K-V)

The block is moved from its original position to the actual cursor position. Thus, after execution of the command, the block is removed from its original position.

Copy (Ctrl K-C)

The block is copied in front of the actual cursor position. After carrying out the instruction, the block will be on the place, where it has been marked, as well as over the actual cursor position.

Erase (Ctrl K-Y)

The marked block is deleted.

Listing

With the help of this function, program sequences marked as a block can be edited on a printer or stored in a file. This file has got the same format as the data transfer to the printer during the printing process. After activating the selection line "Listing", a window will appear, where the output to printer or file, the sheet length, and the left margin can be defined. The default values refer to the printing on continuous form paper.

Load block...

Here, a block is loaded from disk or hard disk. A window will be opened, where the name of the block to be loaded can be input. If this window is ignored and ENTER ↵ is pressed instead, a window with the files available for selection will appear. The structure of a block file is identical with the file of a complete program.

Save block ...

The block is saved to floppy or hard disk. After confirmation of the selection line, a window will appear, in which the name of the block can be defined. The structure of a block file is identical with the file of a complete program.

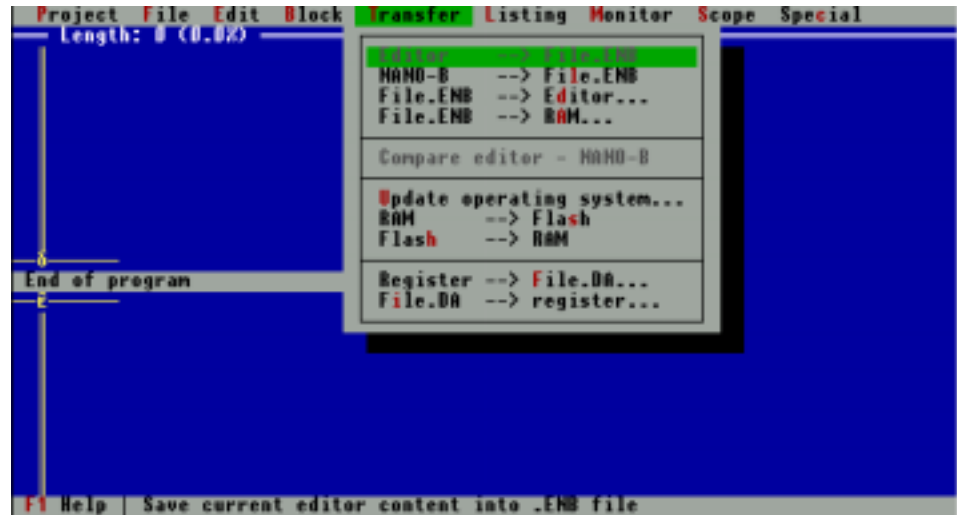
**Note:**

For the "Load block..." and "Save block..." functions, a path to a directory can be specified, which need not be defined again. For this purpose, the following instruction is to be given on DOS level (e.g. in the AUTOEXEC.BAT):

```
SET SYMPASBLOCKS="PATH"
```

where, for instance, "PATH" can stand for C:\BLOCK.

3.5.6 The "Transfer" Menu



Extension, by the example of NANO-B; it can be named differently, depending on the controller

Editor -> File.ENB

By this function the program is transferred into an object file the name of which can be defined in a window.

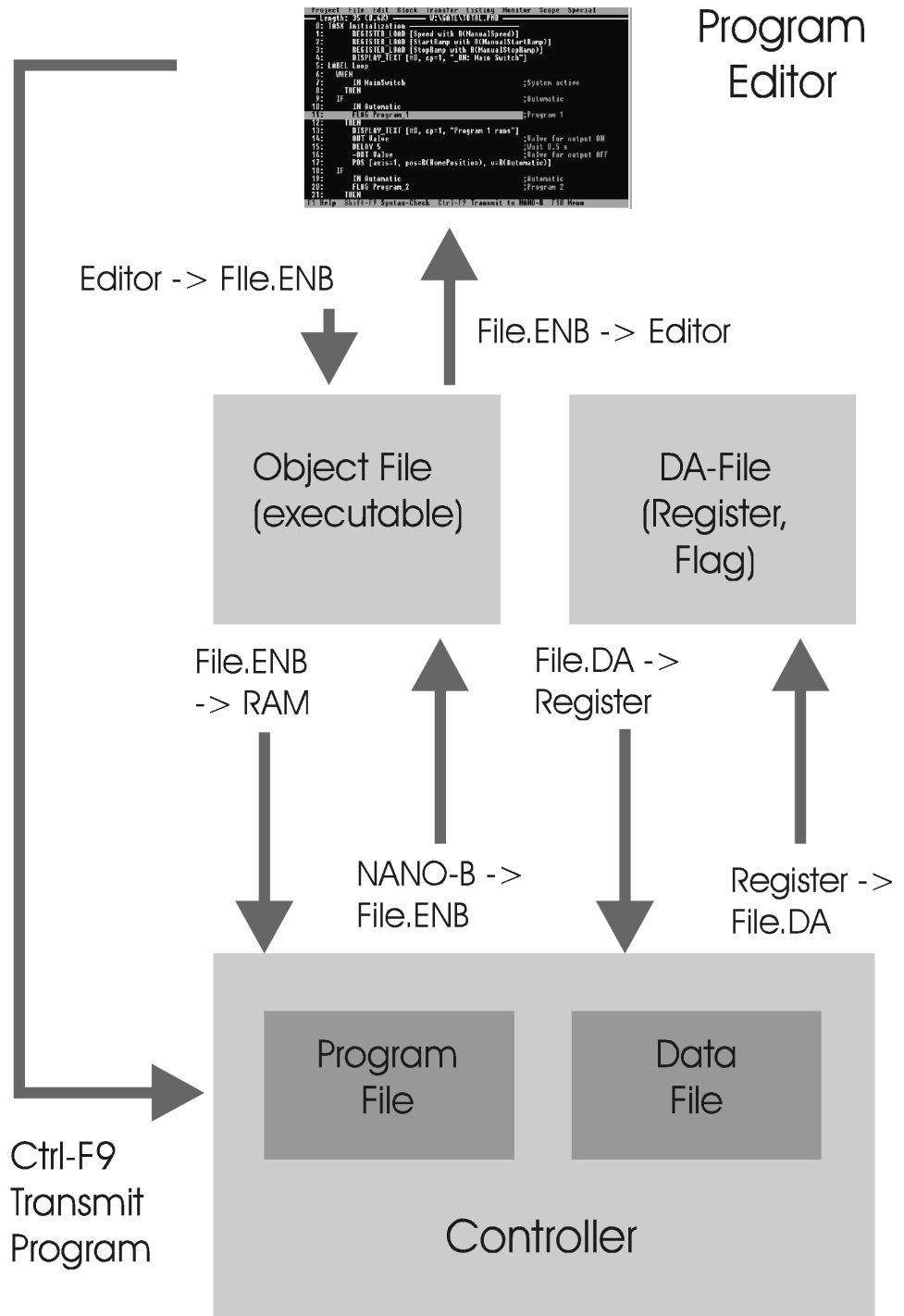
NANO-B -> File.ENB

Transfer of a program from the RAM of a controller into an object file the name of which can be defined in a window.

File.ENB -> Editor...

Program transfer from an object file into the program editor.

SYMPAS Program and System Files (Extension *.ENB, for NANO-B as an example)



File.ENB -> NANO-B...

Program transfer from an object file into the RAM of a controller. The program will be transferred to, yet not started in, NANO-B.

Compare Editor -> NANO-B

The program editor is compared with the editor in the RAM of the file. In a window, information is given, as to what degree both programs are identical.

Register -> File.DA...

You will be able to save self defined register and flag ranges as hard disk files. After function call-up a window will appear, where various ranges can be defined. These ranges can differ from each other in their type. Register and flag ranges are possible- 8 as a maximum. After all the desired ranges have been defined, the window can be left by the first menu line "All ranges defined, start transfer". Now a window will appear, where the file name can be defined. The extension ".DA" will be added by SYMPAS, and the file will be saved on hard disk.

File.DA -> Register...

The file described under "Register -> File.DA" which serves for the storage of register and flag ranges, is loaded into the controller from hard or floppy disk.

Thus, all register and flag ranges of the controller, which have been defined in the file, have been updated.

The DA File

Register and flag ranges are stored on PC or VIADUKT by the DA file

The DA file is an ASCII file which can be stored on the PC or VIADUKT hard disk and reloaded into the controller from there.

As an example, a DA file can look this way:

Header ->	<pre>SD1001 ; NANO-B DATA FILE - JETTER Automation Technique 71642 Ludwigsburg ; C:\SYMPAS\EXAMPLE\EXAMPLE1</pre>	Header Definition
Range of registers ->	<pre>RS 1 10 RS 2 20 RS 3 30 RS 4 40 RS 5 50</pre>	Register List
Range of flags ->	<pre>FS 1 0 FS 2 0 FS 3 0 FS 4 0 FS 5 0</pre>	Flag List

In the example shown above, the following register ranges, resp. flag ranges have been stored in the DA-file "EXAMPLE1.DA":

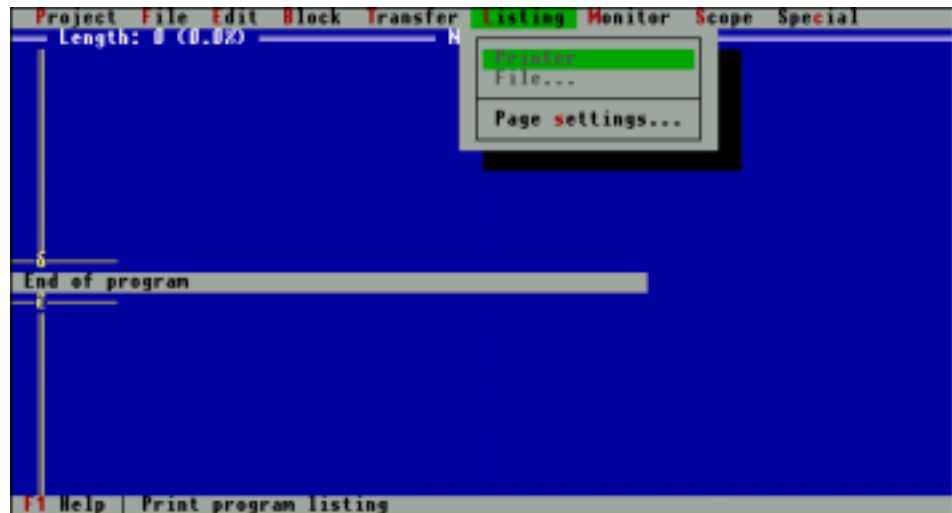
- Register 1 to 5 with the respective content
- Flag 1 to 5 with the respective status

Setup of a DA-file

The register-, resp. the flag list is designed as follows.

- 1st column: Identification RS for register, FS for flag
- 2nd column: Register-, resp. flag number
- 3rd column: Register, resp. flag status

3.5.7 The "Listing" Menu



Printer

The content of a program or symbol editor will be output as a program listing on the printer.

File...

The content of a program or symbol editor will be written as a program listing into a file. After activating the "File" selection line, a window will be open, where the file name can be defined, which is then be taken over into the program listing. The extension *.LST." will be added automatically.

Page settings...

Various settings concerning the page format can be made in this window.

Sheet length

Here, the sheet length can be defined. The default value refers to the printing of listings on continuous form paper.

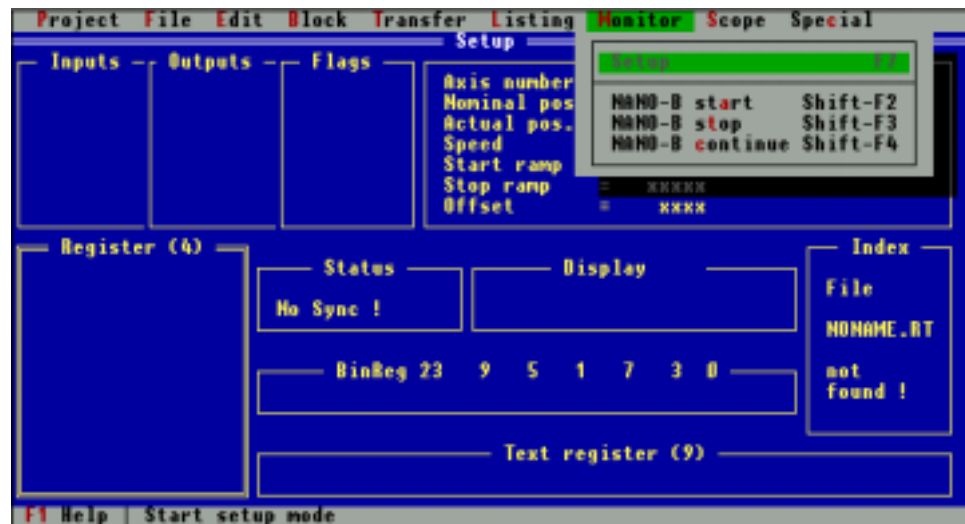
Left margin

Here the width of the left margin of the listing is defined. The input number refers to the number of blanks preceding the actual text.

Form feed

A form feed at the end of each printed listing page is generated by this function (if marked, it is activated). If the form feed function has been deactivated, blank lines will be printed.

3.5.8 The "Monitor" Menu



Setup

By this selection line, switching into the setup screen is caused.

NANO-B start

The program is started in the NANO-B by this instruction after having been transferred into the controller RAM, for example, by the selection "File.EPR -> RAM" out of the "Transfer" pull-down menu. It is started by this instruction.

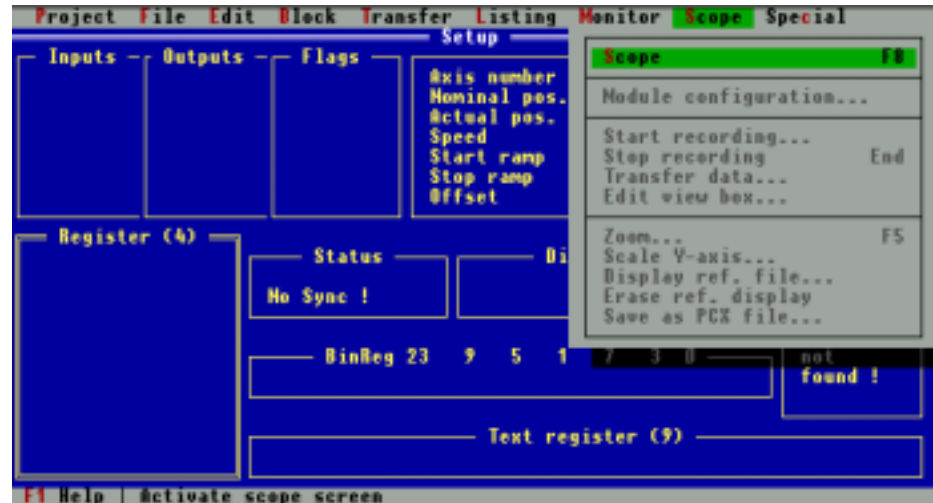
NANO-B stop

Processing of the user program will be stopped.

NANO-B continue

Processing of the user program will be continued at the point where it has been interrupted. The "NANO-B Start" function, though, would start at the beginning of the program.

3.5.9 The "Scope" Menu



Monitoring any register of various controller modules

Using the scope function, any register of the following modules can be recorded:

- PASE-E SV4 Plus (servo controller)
- PASE-E DIMA3 (digital servo controller)
- PASE-E PID4 (digital PID controller)
- PASE-E AD16 (analogue input card)

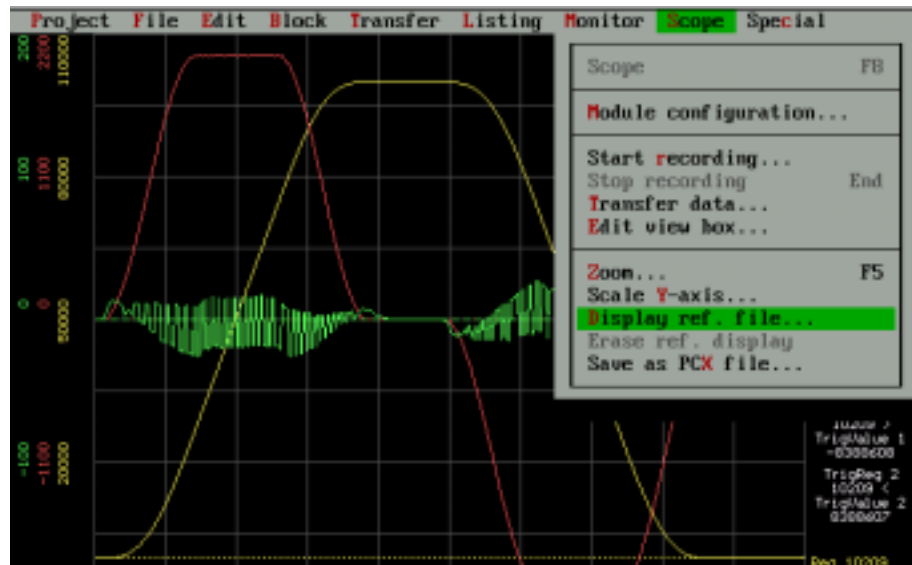
- NANO SV (Servo controller)
- NANO PID (Digital PID controller)

As all register contents can be logged, it is possible, to display speed and position of an axis in relation to time, or to display the graph of an analogue input, to give but a few examples.

3 graphs can be displayed simultaneously

Up to 3 graphs can be displayed simultaneously in the scope screen.

Press (F8)
to open the
Scope screen

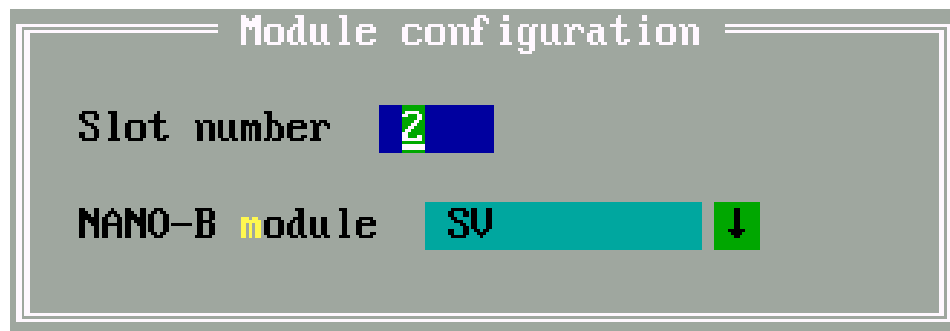


The following functions are available in the "Scope" menu:

Module configuration...

Input of slot
number and
module type

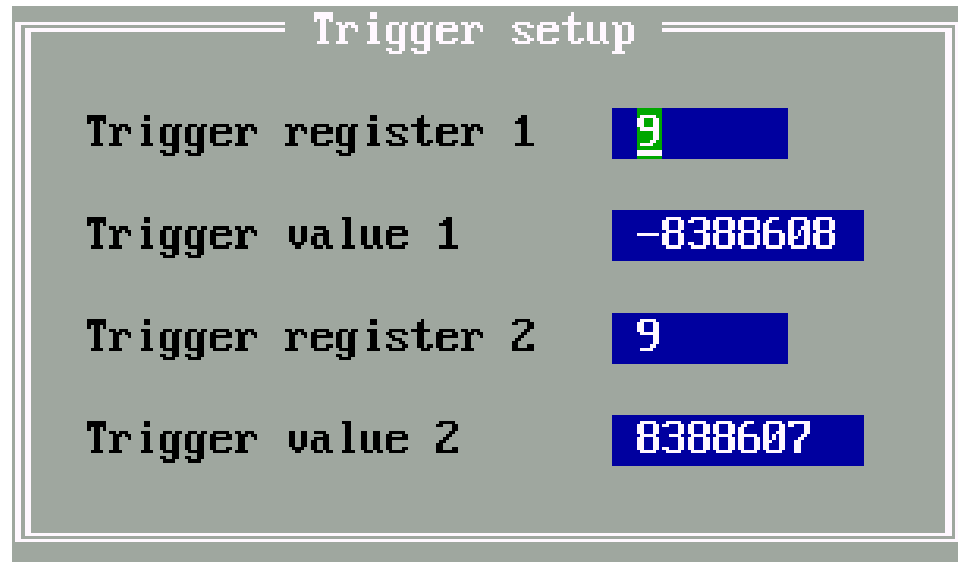
First, a window will appear, where number and type of the module to be monitored will be given.



Conditional start

Recording of the registers, which have been assigned to channels in "Module configuration", depending on the conditions defined in the selection line "Trigger setup".

The following window will be opened:

Trigger setup

The screenshot shows a window titled "Trigger setup" with a grey background and a white border. It contains four rows of configuration data, each with a label on the left and a value in a blue box on the right:

Label	Value
Trigger register 1	9
Trigger value 1	-8388608
Trigger register 2	9
Trigger value 2	8388607

Both trigger conditions must be fulfilled

The condition for trigger register 1 is:

- Trigger register 1 > Trigger value 1

The condition for trigger register 2 is:

- Trigger register 2 < Trigger value 2

Both trigger conditions must be fulfilled.

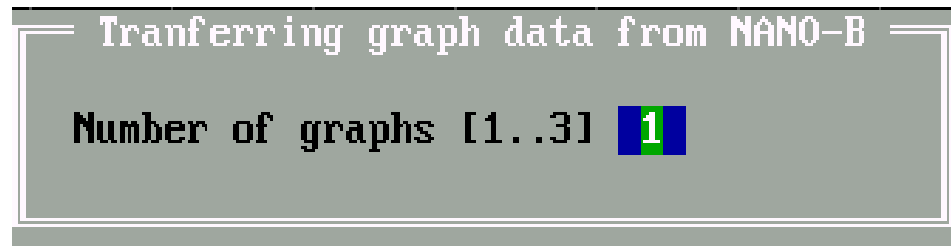
Stop recording

Recording the register values is stopped.

Transfer data...

Input the number of curves to be displayed

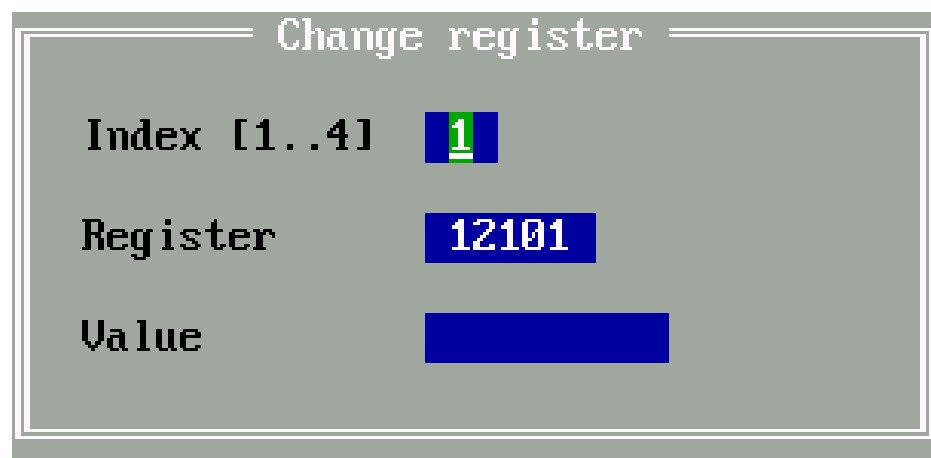
The following window serves for defining the number of curves to be read by SYMPAS from the controller memory and to be displayed on the Scope screen.



Edit view box...

The contents of 4 additional registers can be displayed

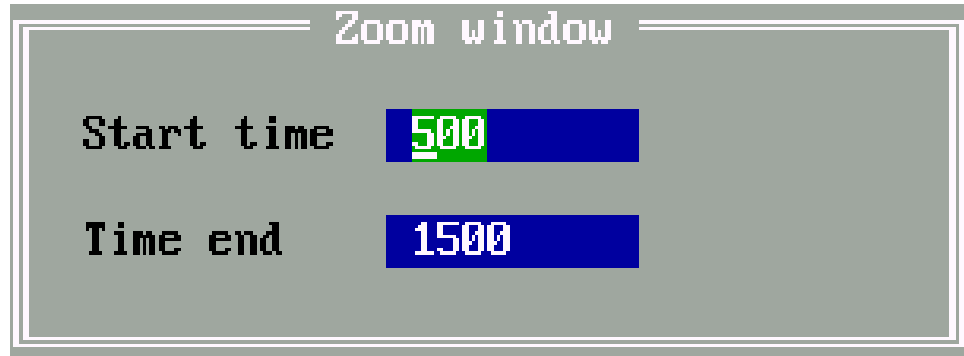
This function helps to display and change up to 4 controller registers of any kind in the top right corner of the Scope screen.



Zoom...

The x-axis can be scaled by the "Zoom" instruction

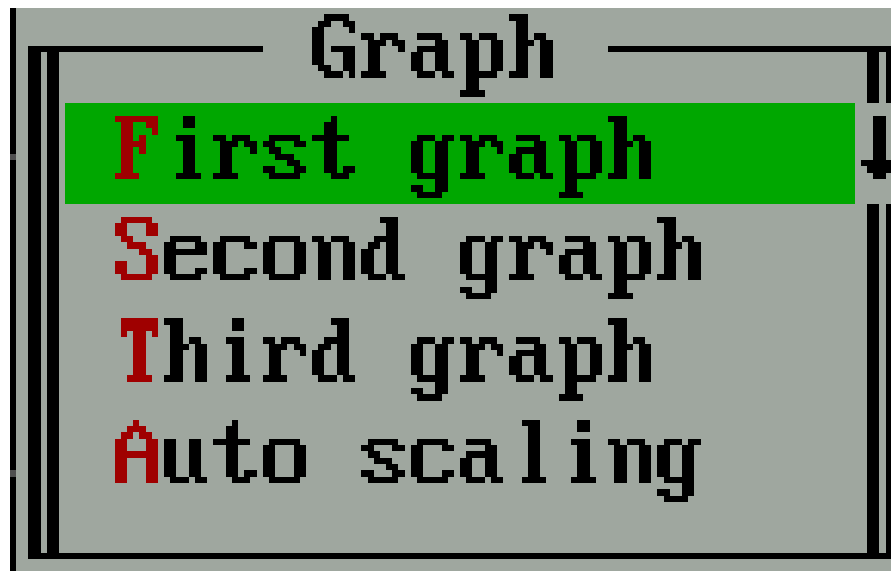
The range of the time axis to be displayed over the entire screen can be defined by the "Zoom" instruction.



Scale Y-axis...

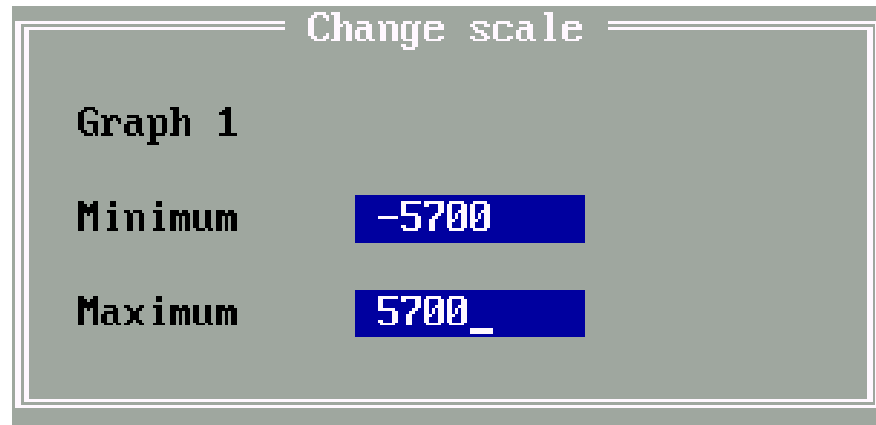
Y-axis scaling of each graph can be selected individually

First define, for which graph the display of the y-axis is to be scaled.



Now input the new value range which is to be displayed.

The display area of each graph can be selected individually



Display ref. file...

The monitor display can be saved by "File / Save" (*.SCP). By "Display ref. file..." a reference file, which has been stored on hard disk, will be displayed on the screen again.

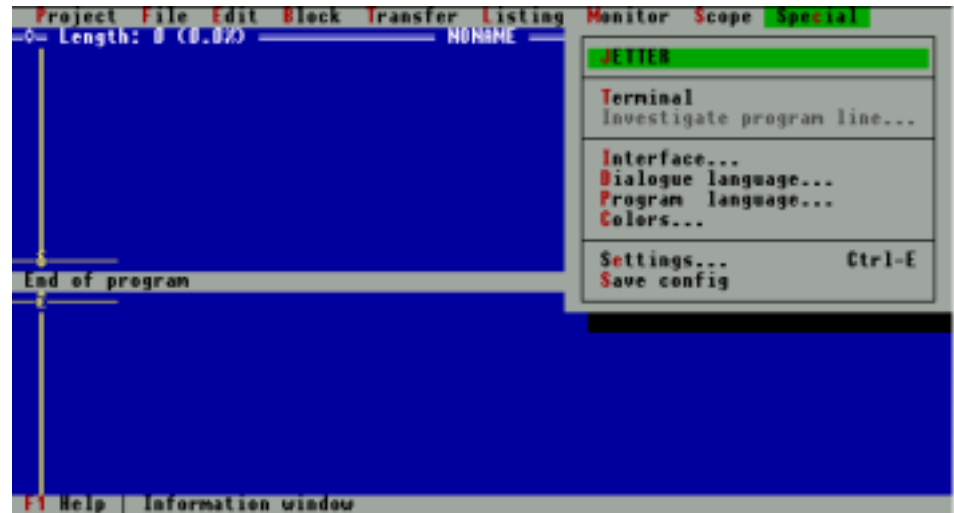
Erase ref. display

The monitor display of the reference file (*.SCP) will be deleted again.

Save as PCX file...

The monitor display will be stored as a PCX file on hard disk.

3.5.10 The "Special" Menu



JETTER

After giving this instruction, an information window will appear, from which the following information can be taken:

- Program version
- Our phone number
- Current interfaces: COM1, COM2, or JETWay
- Controller state: online or offline
- Save environment: ON or OFF
Under this heading information about the switch position of the "Auto save environment" function is given.
- Syntax-Check: ON or OFF
Under this heading information about the switch position of the "Syntax Check" function is given.
- RAM capacity is still available in the PC

Terminal

A terminal is simulated by this function. In the upper section of the screen there will be the data sent via interface, in the lower section there will be the data which have been sent back to SYMPAS.

These functions are reserved for internal use by the JETTER company.

Investigate Program Line

Investigate actual program line (task pointer, special function).

Interface...

Interface
Timeout time
Baud rate

In the window which will appear, the interface can be defined, by which the connection to the controller is made up. A choice can be made between COM1 and COM2 of the PC or the JETWay interface (see *Chapter 2.4 SYMPAS for Several Networked Controllers (JETWay-H)*). In addition, can be defined a timeout for the selected interface. The change between various screens (program editor, symbol editor, setup-screen) can be speeded up using this function, if the controller is not connected. Further, the baud rate for the DA file, resp. for program transfer, will be given.

Dialogue Language...

Here, the dialogue language, e.g. the language of the pull-down menus, help windows, etc. can be defined. A choice between English and German can be made.

Program Language...

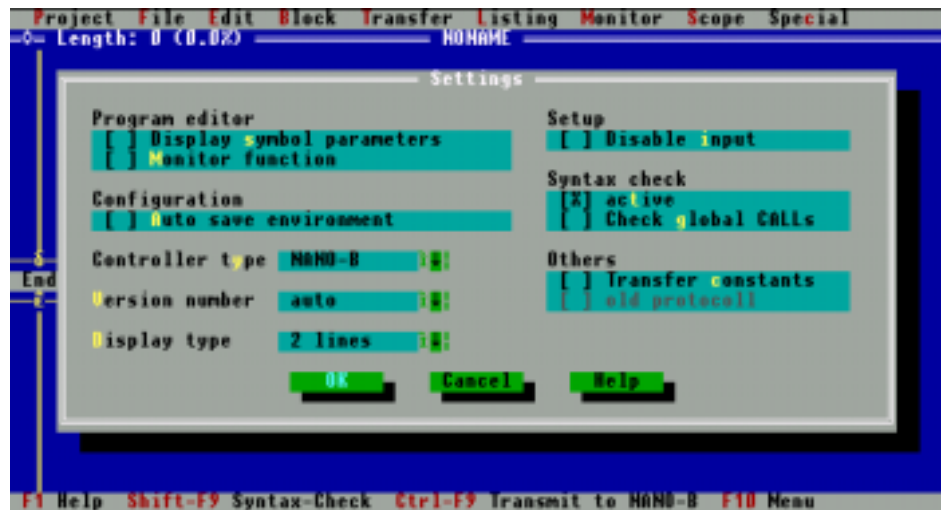
Here, the programming language, i.e. the language, in which the instructions will be displayed, can be selected. A choice between English and German can be made.

Colors...

In this selection line the colour settings for the entire programming environment can be determined. After confirming the selection line a window will appear, where again a choice of 4 subordinate windows is offered. With the help of the TAB key, the windows "Group", "Item", "Foreground" and "Background" will be activated one after the other (distinguished by double frame). By the SHIFT-TAB key combination, the windows are activated in reverse sequence. One can move between the individual windows by the cursor keys. For each line of the "Group" window, there are one or more sub-divisions in the window "Detail", which can be assigned a certain colour in the windows "Foreground" and "Background". Colour setting will be interrupted by pressing the ESC key, while it is confirmed by the ENTER ↵ key. Underneath the "Background" window there is a test text, where a preview of the colour setting is given in an exemplary text.

Settings...

In a window, the following settings can be defined:



Program Editor

- Display of the symbol parameters (Ctrl-Alt-S)

When symbolic expressions are used for parameters of the programming language instructions, the numeric value of the parameter will additionally be displayed in the program text by this function.

If the following line is found in the program

```
REG rNumberParts
```

in the line below, the number of the controller register will be added by the function:

```
REG rNumberParts
    100
```

That way, the assignment of symbolism to physically existing registers can be checked.

- Monitor Function (Ctrl-Alt-M)

The register contents are displayed in the program editor.

If, for example, there is the following program line

```
REGISTER_LOAD [100 with R(200)]
```

the "Monitor function" will lead to the following result:

```
REGISTER_LOAD [100 with R(200)]  
                  0                  23
```

The contents of the respective registers are displayed in the line below and are continuously actualised.

Configuration

- Auto save environment

If this switch is active, all environment settings (as described in *Chapter 3.5.3*) are saved under SYMPAS.DSK. When SYMPAS is started afresh, all settings are restored, when the switch "Auto save environment" is being kept by the instruction "Save environment".

- Controller type

Here, the desired controller type can be set.

With the settings

- o number
- o auto
- o ignore

Compatibility with former versions

- Version number

Here, the version number of the operating system of the controller mentioned above is input.

Auto: Error report, if for program transfer the controller version does not contain the instruction set that has been used (if it is too old).

Number: Error report, if for program transfer the controller version does not contain the instruction set that has been used (if it is too old).

- Display-Type

Here the information is given, whether a 2 or 4 line LCD display is being used (only for the setup-screen).

Setup

- Disable input

Register contents, respectively input, output and flag conditions can be displayed, but not changed..

Syntax - Check

- active

By this function, "Syntax check" is switched on and off. The program is checked by "Syntax check" for the following criteria:

- Has TASK0 been defined?
- incomplete comment
- double flag or task
- conditioning has not been finished
- incomplete conditioning
- brackets have not been set correctly
- task instructions without corresponding task
- condition without corresponding flag
- subroutine without corresponding flag
- GOTO without SUBROUTINE command
- error in instruction syntax
- completeness of task
- number of subroutine levels (20 are possible)
- main program is running in subroutine
- go into a non-corresponding task
- local subroutine has been called up by a non-corresponding task
- error in arithmetic or Boolean syntax

If the "Syntax Check" switch is set to "ON", a syntax-check will be carried out for the following actions:

- before program transfer by the selection line "File.ENB... -> RAM" in the "Transfer" pull-down menu.
- Before automatic program transfer and program start from the program editor with key combination Ctrl-F9.

Independent from the switch position the syntax check is carried out:

- with the SHIFT-F9 key combination in the program editor.
- Check global CALLs

Per definitionem, global calls stand at the end of the last task. Any different positioning will be remarked by the Syntax check. If this function has been deactivated, global calls can be placed anywhere.

Others

- Transfer constants

The constant data defined in the symbol file are transferred to the controller.

- old protocol

For the programming interface (RS232) the old protocol is used (applies to PASE-E PLUS only.)

Save config

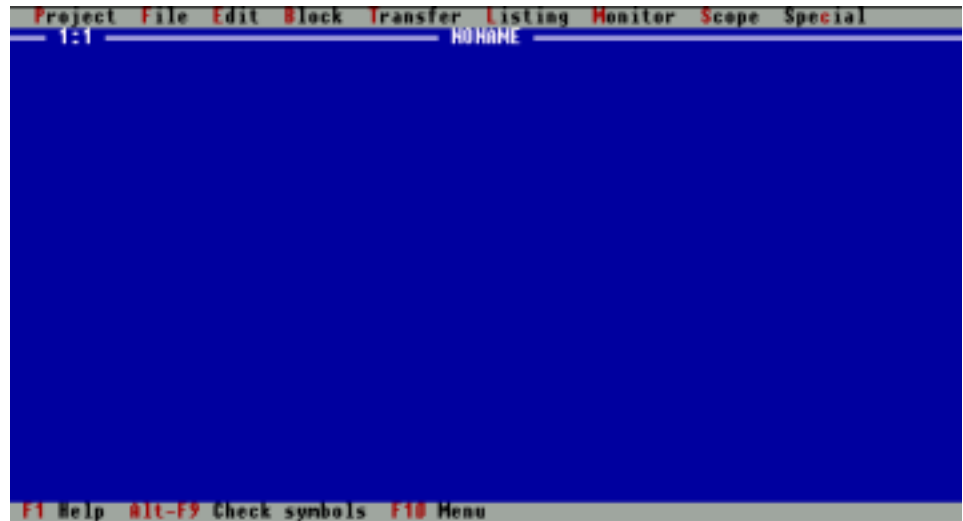
Using the function "Save config", the following configurations are kept:

- Dialogue language
- Programming language
- Interface
- Number of lines per page

This setting refers to output of block and program listings.

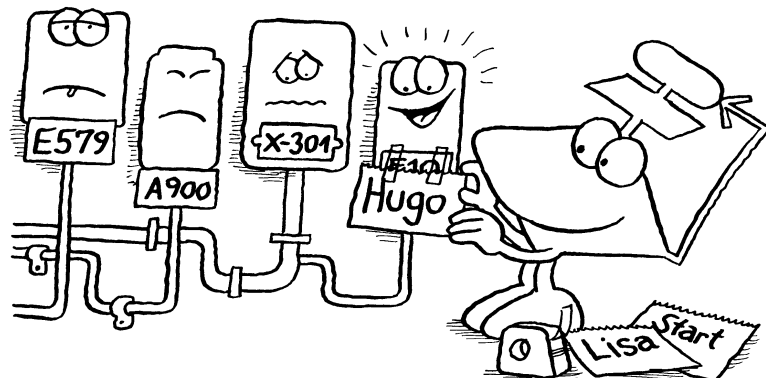
- Margin
This setting refers to the output of block and program listings.
- Switch Positions
 - o Auto save environment
 - o Syntax check
 - o Formfeed at end of page
 - o Timeout
- Colours

3.6 Symbolic Programming - the Symbol Editor



Switch
between
program and
symbol editor
by pressing
(F4)

Using the F4 function key or the corresponding selection line "edit" in the pull-down menu you will get into the symbol editor. In this editor a file to assign symbolic names to all instruction parameters of the programming language can be created. Thus, for example, input "IN102" will become "IN iStart". Each numeric parameter of the instruction language can be replaced by such symbolic naming, which means more clarity of the program and laees maintenance after completion. The following order should be observed: First, create the complete symbol file, in order to write the corresponding



program into the program editor.

3.6.1 Keys and Functions in the Symbol Editor

Some rules for the creation of the program symbolism must be considered.

Any available ASCII character, starting from the ordinal number 32, can be input.

Valid symbols must start in column 1 and must be separated from the following parameter by at least one blank. The symbol length is limited to 15 characters.

Parameters of the type "string" (DISPLAY_TEXT) have to be enclosed by one of the following characters: ", ', or # (e.g. "Hello World"). The string length is limited to 24 characters.

INCLUDE files can be included after the following pattern:

```
#INCLUDE file name
```

INCLUDE files must also start in the first column.

Commentaries must be preceded by semicolon or at least one blank.

Cursor Movement:

Key:	Function:
Cursor up	one line back
Cursor down	one line forward
Page up	page back
Page down	page forward
Ctrl-page up	go to first line
Ctrl-page down	go to last line
Cursor left	one column back
Cursor right	one column forward
Home	go to beginning of line
End	go to end of line
Ctrl-cursor left	previous word
Ctrl-cursor right	next word

Editor Instructions:

Key:	Function:
ENTER	new line
BS	delete character in front of the cursor
DEL	delete marked character
Ctrl-Y	delete line

Block Operations:

Key:	Function:
Ctrl-K B	mark top of block
Ctrl-K K	mark end of block
Ctrl-K V	move block
Ctrl-K C	copy block
Ctrl-K Y	delete block
Ctrl-K R	load block from disk
Ctrl-K W	write block onto disk
Ctrl-K P	print block
Ctrl-K H	switch off block
Ctrl-K L	mark line
Ctrl-Q B	search for top of block
Ctrl-Q K	search for end of block

Program Labels:

Key:	Function:
Ctrl K 0..9	0 to 9 cursor positions of the symbol text are stored.
Ctrl Q 0..9	cursor positions 0 to 9 are searched for in the symbol text.

3.6.2 Creating a Symbol File (in the Symbol Editor)

The numeric parameters of the programming language can be replaced by symbolic names.

REG 100 will become

REG **rNumberOfParts**

A symbol file is created according to the following pattern

- A valid symbol has to begin in the first column. If a line starts with ";" or a blank " ", this line is interpreted as a commentary line.

rNumberOfParts

- This symbolic parameter `rNumberOfParts` is now given its numeric equivalent, which will appear in the same line only being separated by at least one blank " " from the symbolic name.

`rNumberOfParts 100`

- A commentary can be added now. It must be separated from the parameter by at least one blank or semicolon.

```
rNumberOfParts 100 ;Commentary: The symbol  
"NumberOfParts" is related  
to the numeric parameter  
"100".
```

REG 200 = REG rNumberOfParts

register REG 100 , which is physically existant in the controller is given the symbolic name "NumberOfParts".

After writing the symbol listing, the creation of programs in the program editor can be supported as follows: For example, a register with symbolic naming will be input into the program editor. After input of the "RE" short form a window will appear, in which register numbering, respectively symbolic naming can be carried out. If in this window the first letter of the symbolic name is written, and after this the key combination SHIFT-? is pressed, a window of all symbolic names starting with this letter will appear. Now you can easily select the respective name by cursor key. This way, symbols that have been defined once, will not have to be typed over and over again.

Example of a Symbol File:

```

Symbolisting von „prog01“ V1, 28.04.1996 15:13      Seite 1

JETTER PROCESS-PLC NANO--B

Customer/Project: sympas manual
Place           : Ludwigsburg
Date            : 28.04.1996 15:13
Version         : 1

;**** TASK ****
;
tControlTask    0           ;The process is controlled by the task
tAutoTask       1           ;Automatic-Task
tDisplayActualpos 2         ;Display-Task Actual position
tEMERGENCY_STOP 3         ;EMERGENCY STOP-Task

;**** LABELS ****
;
sLoop           40          ;Flag 40
sDriveLeft      41          ;Flag 41: Program sequence drive left
sDriveRight     42          ;Flag 42: Program sequence drive right
sRefDrive       43          ;Flag 43: Program sequence reference run

;**** INPUTS ****
;
iEmergStopSwitch 105        ;Switch Emergency Stop Condition
iEmergDoor        106        ;Switch Emergency Stop Condition;
                               ;Emergency Door is Open
iAutomatic        107        ;Switch Automatik/Hand
iStartButton      108        ;Button „Start“
iStop_Button      201        ;Button „Stop“
iRef_Run/Button   202        ;Button „Reference Run“

;**** REGISTER ****
;
rSM_Status        11100      ;Status register SM-Control
rCommand Register 11101      ;Command register SM-Control
rSM_Speed         11103      ;Nominal speed register SM-Control
rActualPosition   11109      ;Actual position register SM-Control

;**** FLAG ****
;
fReferenceOK      1          ;Flag: Reference Drive has taken place
fAutomaticTask    2          ;Control Flag Automatic-Task
fArrowLeft       217        ;LCD Cursor Key left
fArrowRight      218        ;LCD Cursor Key right

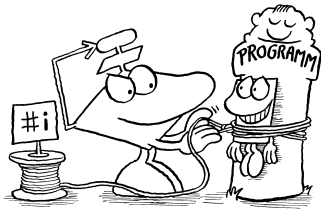
```



Note:

The path of the symbol file must be identical to the path of the corresponding program file. The relationship between program file and symbol file will be demonstrated extensively in *Chapter 6. Demonstrating Example: Handling-System*.

3.7 INCLUDE Files



SYMPAS programs respectively program parts can be included in a SYMPAS program. Thus, a complete program can be combined out of a pool of SYMPAS modules. `#INCLUDE` instructions can be part of both program and symbol files. The maximum number of `INCLUDE` files has been restricted to 32 per editor.

Structuring of programs

Function libraries

Enlargement of maximum program length

INCLUDE files are used for

- modular structuring of programs
- Combining `INCLUDE`-files in instruction libraries
- Avoiding restrictions in the maximum program length

3.7.1 INCLUDE Files in the Program Editor

`#INCLUDE` instruction as a place holder for the text of the `INCLUDE` file

`INCLUDE`-files are integrated into the text of the main file by the `#INCLUDE` instruction. This instruction line is functioning as a place-holder for the program text, which is written in the `INCLUDE`-file. Exactly the program text which is in the `INCLUDE` file is logically placed in the main file, where the `#INCLUDE` instruction has been inserted under the name of the respective file.

Define Main File

32 INCLUDE files are possible

In the menu "File/Main file..." the main file will be defined. In this main file the INCLUDE-files will be inserted..

No nesting of INCLUDE files



Figure 9: Up to 32 INCLUDE files can be defined in the main file.

The #INCLUDE Instruction

The INCLUDE file is inserted by the #INCLUDE instruction

With the help of the #INCLUDE-instruction, the INCLUDE file will logically be integrated in the program text.

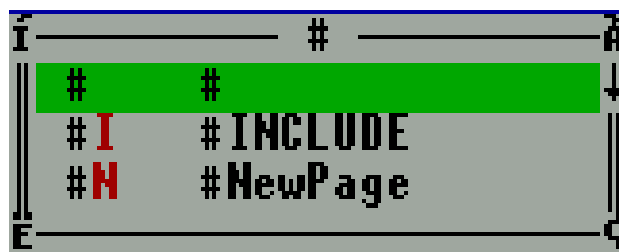


Figure 10: The INCLUDE file is inserted by the #INCLUDE instruction.



Note:

- 32 INCLUDE files can be defined in the main file.
- Nesting of INCLUDE files is not permitted. In one INCLUDE file no further INCLUDE files must be defined.

Example:

The result might be similar to the following one:

Main file:
TOTAL.PNB

INCLUDE file
PUMP01

```

Project File Edit Block Transfer Listing Monitor Scope Special
0 Length: 35 (0.62) W:\GATE\TOTAL.PNB
22: DISPLAY_TEXT [#0, cp=1, "Program 2 runs"]
23: OUT Ventilator ;Ventilator ON
24: DELAY T0 ;Waits 1 s
25: POS [axis=1, pos=R(Magazine), v=R(Automatic)] ;Put off
26: WHEN
27: AXARR axis=1
28: THEN
29: -OUT Ventilator ;Ventilator OFF
30: DELAY T0 ;Waits 1 s
31: DISPLAY_TEXT [#0, cp=1, "Axis in Position"]
32: TASK TPumpControl
33: #INCLUDE C:\SMPAS\PUMP01
34: GOTO TPumpControl
End of program
    
```

Figure 11: The INCLUDE file PUMP01 has been inserted into TOTAL.PNB

Determine with the help of the pick list, which file is to appear in the program editor

The Pick List

With the help of the pick list (file / pick list ...) selection between main file and INCLUDE files can be made. For this purpose, first load the necessary files (New File) into the pick list. From then on, the file will appear in the program editor, which has been mouse-clicked upon in the pick-list.



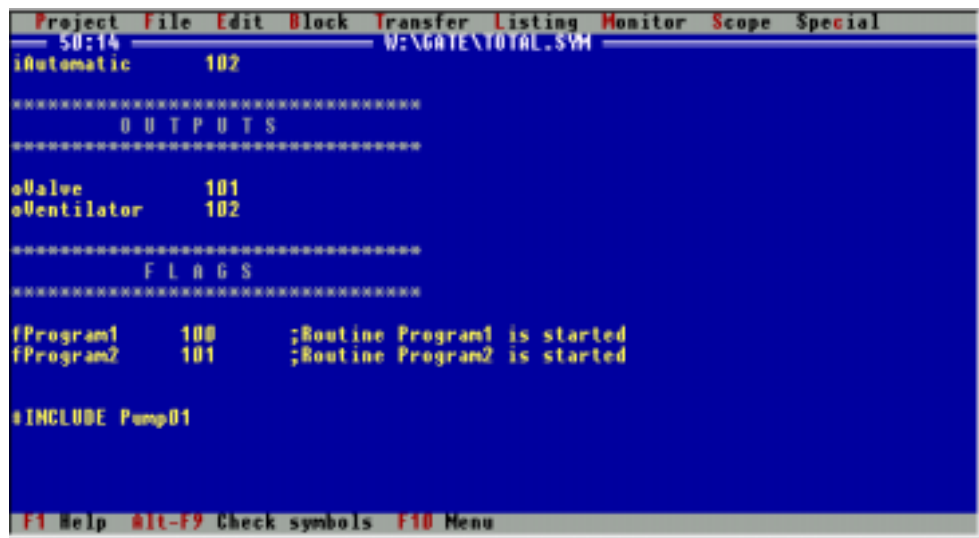
Figure 12: Integrate files in the list by "Open". The file which has been mouse-clicked upon in the list will appear in the screen..

3.7.2 INCLUDE Files in the Symbol Editor

By **#INCLUDE** further symbol files can be integrated in the symbol text

A symbol file serves for integrating further files as INCLUDE files. Thus, a library of pre-designed, application related symbol files can be created, which can be integrated into the symbol text if necessary. In the symbol editor, the following line can be read, for example:

#INCLUDE
instruction →



**INCLUDE files
can be
defined
together
with a path**

The INCLUDE-file can be defined together with a path. Nesting on several levels is not possible. The respective pieces of information are taken from the files on hard disk and will not appear in the symbol editor.



Note:

- 32 INCLUDE files can be defined in the symbol file.
- Nesting of INCLUDE files is not permitted. In an INCLUDE file, no further INCLUDE files must be defined..

INCLUDE Files:

**The file name
and the file
itself must be
o.k.**

For the use of INCLUDE files, two aspects have to be considered. First, the file name must be identical with the name that is on the hard disk. Secondly, the content of the respective INCLUDE-file must be o.k. In case there are errors in the INCLUDE-file or in the file name, leaving the symbol editor is not possible any more. The corresponding include files have to be cleared, if an error has occurred.



Note:

If the symbol editor cannot be left because of a faulty INCLUDE file, the errors of the INCLUDE files must be cleared (with ;). Then, the INCLUDE file which has been cleared of the errors can be left.

The design of
the **INCLUDE**
file is identical
to the design
of a symbol
file

May we remark in conclusion, that the design of an INCLUDE file is identical to that of a symbol file. Thus, any existant symbol file that has been saved on hard disk, can be included in any further symbol file as an INCLUDE file, which then must not contain any further INCLUDE files, though.

3.8 Error Messages

The following SYMPAS error messages are meant to support the program editor, the symbol editor, as well as general programming of PROCESS-PLC controllers.

Call the syntax
check by
(SHIFT) (F9).

The program with its corresponding symbol file is checked by the syntax check. This will (if activated; *Chapter 3.5.10 The "Special" Menu*) be activated before program transfer into the controller, by the key combination SHIFT-F9.

Check the
symbol text by
(ALT) (F9)

Further, there is the possibility in the system editor to check the symbol text for syntactic correctness by the key combination ALT-F9.

In another window there will be information on numbers and categories of errors. After confirming with ENTER ↵, the cursor will be placed at the error position of the program editor. A red error line will provide further information.

If several errors have been reported in the window mentioned above, one error after the other can be corrected by calling the syntax check function as often as necessary by SHIFT-F9. A context-related help message per error will appear in the red error line.

The following error messages are possible:

Error Messages of Symbol Errors:

1 Symbol not found

A symbol placed in the program text has not been defined in the symbol file.

2 Symbol already exists

A symbol has been defined several times; for instance INPUT 102 twice.

3 Invalid parameter

An invalid parameter has been assigned to a symbol in the symbol editor: numeric parameter: max. \pm 8388606.

4 Exceeds valid value range

The instruction parameter is outside the valid value range.

5 Symbol is not a register

One symbol has been defined for both a numeric and a text parameter.

6 Invalid string constant

An invalid string parameter has been defined. String parameters must stand between " or ' or ' characters. Their length can be 24 Bit as a maximum.

Error Messages in the Syntax-Check:

9 Function definition (X) not found

A function called in the program text has not been defined in the program heading.

10 Function call does not match

The number of parameters is not equal in call-up and definition

11 Too many labels

There are too many relative flags (managed by SYMPAS itself)

13 First instruction has to be TASK 0

The first instruction in a PROCESS-PLC program must be TASK 0.

14 TASK(X) already exists

Task number "X" has already been defined in the program.

15 LABEL (X) exists already

Flag number "X" has already been defined in the program.

16 TASK not found

A task defined as `TASKBREAK`, `TASKCONTINUE`, `TASKRESTART` does not exist.

17 TASK (X) is no endless loop

Task number "X" has not been closed by a `GOTO` instruction. Each task must be closed in order to form an endless loop.

20 GOTO label not found

The `LABEL(X)` or `TASK(X)` relating to the `GOTO(X)` instruction do not exist.

21 GOTO into another TASK

It is not possible to give `GOTO` instructions for jumps into other parallel branches.

22 GOTO into procedure not allowed

`GOTO`s into functions (from outside) are not permitted.

23 GOTO from procedure not allowed

`GOTO` out of functions (to the outside) is not permitted.

24 Global subroutines only at the end of program text

There are differences between local and global subroutines. Local subroutines are only used by one task. They are placed at the end of a task.

Global subroutines are used by various tasks and have to be placed at the end of the entire program text, that is, after the text of the last task. If this structure cannot, or is not to be maintained, the syntax check can be deactivated with the help of the respective switch

25 Only 20 subroutine levels valid

20 subroutine levels are permitted.

26 RETURN without SUBROUTINE

A RETURN instruction that has not been preceded by CALL has been found by SYMPAS.

27 Main routine runs into subroutine

The main program will turn into a subroutine.

28 CALL not found

There is no FLAG(X) corresponding to a CALL(X).

30 WHEN not allowed here

No WHEN instruction is permitted by the program syntax at this position.

31 IF not allowed here

No `IF` instruction is permitted by the program syntax at this position.

32 ELSE without IF..THEN

`ELSE` without a preceding `IF..THEN` instruction has been detected by SYMPAS.

33 ELSE, IF, WHEN, THEN too far from IF

The program text in a conditioned decision, this is, between `IF` and `THEN`, or between `IF` and `ELSE` or to the final instruction belonging to `IF - THEN`, `IF, WHEN -` is too long. This problem can be solved by shortening the respective program text.

34 IF, WHEN, THEN too far from ELSE

The program text in a conditioned decision, here between `ELSE` and the corresponding final instruction - `THEN, IF, WHEN -` is too long.

35 THEN expected

At this point, the instruction `THEN` is waited for by the compiler.

37 Allowed only in input condition

These operators are only allowed between `IF` (`WHEN`) and `THEN`.

38 Allowed only in output instruction

These operators are only allowed after `THEN` and `ELSE`.

39 Form syntax error

The operators `=`, `+`, `-`, `*`, `/`, `WOR`, `WAND`, `WXOR`, `ACTUALPOS`, `ND`, `NB`, `NH` have been used in a wrong context in this operation.

41 Numeral or variable expected

At this point a numeral or variable is expected by the compiler.

42 "=" expected

At this point an equal sign is expected by the compiler.

43 Boolean expression expected

At this point a Boolean expression is expected by the compiler.

44 Arithmetic compare operator expected

At this point an arithmetic compare operator is expected by the compiler.

45 ")" without "("

Parentheses have not been set completely

46 ")" expected

Brackets have not been set completely.

47 Only 3 parenthesis levels valid

3 parenthesis levels are valid as a maximum.

50 Function definition only allowed before TASK0

Functions must be defined before the first task (TASK 0).

51 END_DEF without DEF_FUNCTION

END_DEF has been specified without DEF_FUNCTION. In END_DEF , a function is concluded by DEF_FUNKTION.

52 END_DEF expected

A function definition has not been concluded by END_DEF.

53 RETURN expected

Before an END_DEF, RETURN is missing.

Miscellaneous Errors:

55 Unknown instruction

An unknown instruction has been detected by the compiler.

56 Program too large for controller memory

The program memory of the controller is too small for the program that is to be transferred.

57 "}" without "{"

Commentary parentheses have not been set completely.

58 "}" not found

Commentary parentheses have not been set completely.

59 Cannot open file

DOS error in the context of INCLUDE-files ("File not found" or "Too many open files").

60 Insufficient RAM space

In the PC memory there is not enough space for the INCLUDE file.

61 Only single nesting depth allowed for INCLUDE files

In an INCLUDE file, there is another #INCLUDE.

62 Only 8 include files allowed

Not more than 8 INCLUDE files may be defined in one main file.

63 INCLUDE files only allowed if main file defined

INCLUDE files can only be defined in a main file.

64 Unexpected end of file

System error message.

65 GOTO distance larger than 32 kByte

A label handled by SYMPAS is too far from the GOTO instruction. Reduce distance.

66 Controller version x.xx needed

An instruction has been used, for which a later operating system version is needed than the one that has been defined in "Settings...".

3.9 Files, Extensions, etc.

Please find the compilation of files, which are provided by SYMPAS, in the up-to-date README file, which can be found on the SYMPAS disk to be read on DOS level.

All files generated by SYMPAS while working in it, will be shown in the survey below.

NAME.PPE (PASE-E), **NAME.PPM** (MIKRO),
NAME.PPD (DELTA), **NAME.PNA** (NANO-A),
NAME.PNB (NANO-B)

These are the names of the program files, in which the program text is to be stored.

NAME.BKE (PASE-E), **NAME.BKM** (MIKRO),
NAME.BKD (DELTA), **NAME.BNA** (NANO-A),
NAME.BNB (NANO-B)

This way, the backups for the corresponding program files are named.

NAME.SYM

These are the symbol files of the corresponding program files. The name of the program file need not be identical with the name of the symbol file. The symbol files must be found in the directory of the corresponding program files.

NAME.BKS

The backups of the symbol files are named this way.

SYMPAS.CFG

This is the configuration file, where all the settings are stored, which are selected in the "Special" pull-down menu "Save Settings".

SYMPAS.DSK

This desk file will be considered by SYMPAS during startup, when the switch "auto save environment" has been stored in activated state in the SYMPAS.CFG file. According to the design of SYMPAS.DSK, the environment will be restored after startup.

NAME.DSK

In this desk file, all settings have been stored, which can be addressed by the selection line "save environment in the "file" pull-down menu. Besides the SYMPAS.DSK file, the user can still create further files to store the configuration of the environment.

NAME.SUE (PASE-E), **NAME.SUM** (MIKRO),
NAME.SUD (DELTA), **NAME.SNA** (NANO-A),
NAME.SNB (NANO-B)

In this file, the settings of the setup screen will be stored.

NAME.LST

In this file, printer outputs will be stored, which are to be transferred into a file.

NAME.RT

System file, the existence of which is essential for the functioning of the index window in the setup screen.

NAME.EP (PASE-E), **NAME.EPR** (PASE-M),
NAME.EPD (DELTA), **NAME.ENA** (NANO-A),
NAME.ENB (NANO-B)

This object file is created with the help of the "Editor -> File.EP" in the "Transfer" pull-down-menu.

NAME.DA

Register and flag range file. With the help of the "Register -> File.DA..." selection line in the "Transfer" pull-down-menu you can store register and flag ranges, which you have defined yourself, in the above mentioned files on floppy or hard disk.

NAME.SIT

Sympas Include Table contains already included symbols in binary form.

3.10 Miscellaneous

3.10.1 Indirect Addressing



Activate the indirect level in the definition window by (CTRL) (R) or (SPACE)

Indirect addressing will be defined in the opened window, where one, or more than one, instruction parameters are defined. For this purpose, press the CTRL-R key combination or the SPACE key. Now, an "R" will appear in front of the parameter line, or - after pressing the keys twice - "RR" will appear, if doubly indirect addressing is possible in this instruction.

3.10.2 Commentaries

There are three ways of inputting commentaries into the program editor:

- press the ";" key and input the respective commentary as a program line. Confirm by pressing ENTER <↓.
- to add a commentary at the right hand side of the program text, press the cursor key → and input the respective commentary.
- Further, it is possible to integrate commentaries into the program text by writing them in braces { ... }. All characters in braces will be interpreted as belonging to a commentary and will thus not be compiled (commenting of program passages).

3.10.3 Call-up by the /o Switch (Laptop, Notebook)

SYMPAS has been programmed in overlay technique and thus needs minimum space in the working storage of the PC. On the other hand, frequent access to the hard disk of the PC will be necessary when this technique is used. Normally, this is no problem, as those activities will hardly be realised.

If you have installed SYMPAS on a disk drive (access time is problematic), or in a laptop or a notebook (in case of battery operation, operation time will be reduced), though, SYMPAS can be called by

```
SYMPAS /o
```

on DOS level. Now the overlay-buffer in the working storage of the PC will be enlarged, so that no access to hard disk will be necessary any more, as all SYMPAS program parts will continuously be in the RAM. Thus, the problems mentioned above will be solved from their very roots.

```
SYMPAS /oxxxx
```

Expand overlay range by xxxx Bytes.

As an alternative to using the switch, 20 free 16 kByte blocks EMS can be installed on the PC (see DOS manual).

3.10.4 The NOSYMPAS.EXE Program

The NOSYMPAS.EXE program is a reduced version of the SYMPAS.EXE program.

The programs are meant for end users, who are to be granted only very restricted possibilities of manipulation. Programs can be loaded from hard disk and out of the controller (RAM), yet they cannot be edited. As usual, access to the setup screen has been provided, although data manipulation is not possible after call-up. After release of the disable in the "settings" dialogue, register values, inputs, outputs, etc. of the PLC controller can be changed.

If a customer is to be supported, the following aspect might be helpful: Program name (e.g. *.PNB) and/or setup file name (e.g. *.SNB) can already be defined on DOS level, when SYMPAS is started.

```
NOSYMPAS /NANOB PROG01.PPE SETUP05.SUE
```

NOSYMPAS.EXE will function in the following order:

- load *.PNB
- load *.SNB
- activate setup screen, after value input resp. change have been disabled first (release with the help of selection line "disable input" in the "Special/Settings..." pull-down menu).

3.10.5 Switching to DOS

By the Alt-F5 key combination the DOS screen is moved into the foreground, that has been active before SYMPAS call-up, respectively after leaving the DOS shell by EXIT. By pressing any key, the SYMPAS screen can be called again.

3.10.6 Password

Activation

In order to define a password, SYMPAS is to be given the call-up parameter /p (/pv). In this case, the password is input by a window at the beginning of SYMPAS. A choice between two varieties can be made:

/p = the characters that have been input will not be readable in the display; for the sake of security, the input must be repeated once more.

/pv = the characters that have been input can be read; thus, repetition of the input is not necessary.

Definitions

- a password can have a maximum length of 8 characters, and it must have a minimum length of 5 characters.
- if there is no error report after password input, the password is valid.

Application

- the coded keyword will be written into the program header or directly into the controller (Ctrl-F9).
- if by the "File.ENB -> Editor" selection line a program is recompiled, and if a password has been defined and taken up into the program head, this must be input now, otherwise no compilation of the program out of an EPROM file will be possible. If the password, which has been input at the program start of SYMPAS, is identical with the password in the program header of the control program, a second password input will not be needed.

3.10.7 SYMPAS Version 3.09 ff, and MIKRO up to 2.10

If SYMPAS from version 3.09 onwards is executed together with any MIKRO of an operating system version earlier than 2.10, the syntax check belonging to SYMPAS will not be able to recognise an excession of the 3 subroutine levels that are permitted as a maximum.

3.10.8 SYMPAS and PASE-J (up to version 4.04)

If a controller of the PASE-J type is used, SYMPAS must be started by the following call line:

```
SYMPAS /J
```

3.10.9 SYMPAS in the Network (PASE-E up to version 4.04)

If SYMPAS is used in the network, the path to the SYMPAS(_M).CFG configuration file is to be defined by the DOS command line

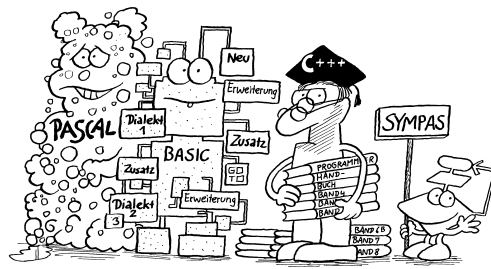
```
SET SYMPAS_CONFIG=Path.
```

3.10.10 Further Command Line Parameters (Call-Up Switches)

/L	high contrast colour chart for laptops
/S	symbols in selection windows, in alphabetic order.
/B	input windows appear under the cursor line.
/Bxxxx	input of a fixed baud rate (which has been set in the controller).
/PASEE+	SYMPAS setting: PASE-E PLUS
/MIKRO	SYMPAS setting: MIKRO
/DELTA	SYMPAS setting: DELTA
/NANO A	SYMPAS setting: NANO-A
/NANO B	SYMPAS setting: NANO-B (Default, if no controller has been defined).

II. SYMPAS Programming

1. Overview



For the execution of a **controller task** with the help of a PROCESS-PLC, a program adapted to the problem is needed besides an apt hardware configuration. The controller is caused by the program to execute the respective controller task.

SYMPAS follows the motion sequences of the machine

The PROCESS-PLC programming language is unconventional in a certain sense. It is adapted to the motion sequence of the machine to be controlled, respectively of the process to be controlled, and not to a contact plan, as this would usually be the case.

This makes a vast difference: For programming a controller task it is **not** necessary to consider traditional means first (contactors, contactor relays ...). The motion sequence can be transferred almost directly into the programming language of the PROCESS-PLC.

The process is described by plain-text language

In addition to the mere descriptive language, there is also floating point arithmetic, data management and multitasking of up to 32 tasks.



During program execution, the PROCESS-PLC differs from conventional controllers in so far, as no cyclic storage run takes place. This way, the reaction time is independent of the program length, as only those input conditions, which are necessary for continuation of the controlling process, yet not any other condition, will permanently be tested.

2. Fundamentals of Programming

2.1 Principles of Program Setup

The clear difference between the programming language and the standard PLC languages has a certain effect on the basic program structure.



Normally, several parallel partial processes have to be controlled, which, in general, will run sequentially.

Thus, the program structure should follow the arrangement of the parallel programs as closely as possible. It is helpful to define a basic program, which is to function as a main program and which will activate and connect the "sub"-programs, called subroutines, via flags.



In addition, parallel programs for execution of asynchronous instructions given by, for example, user elements, by the central controller or by VIADUKT, will normally be needed.

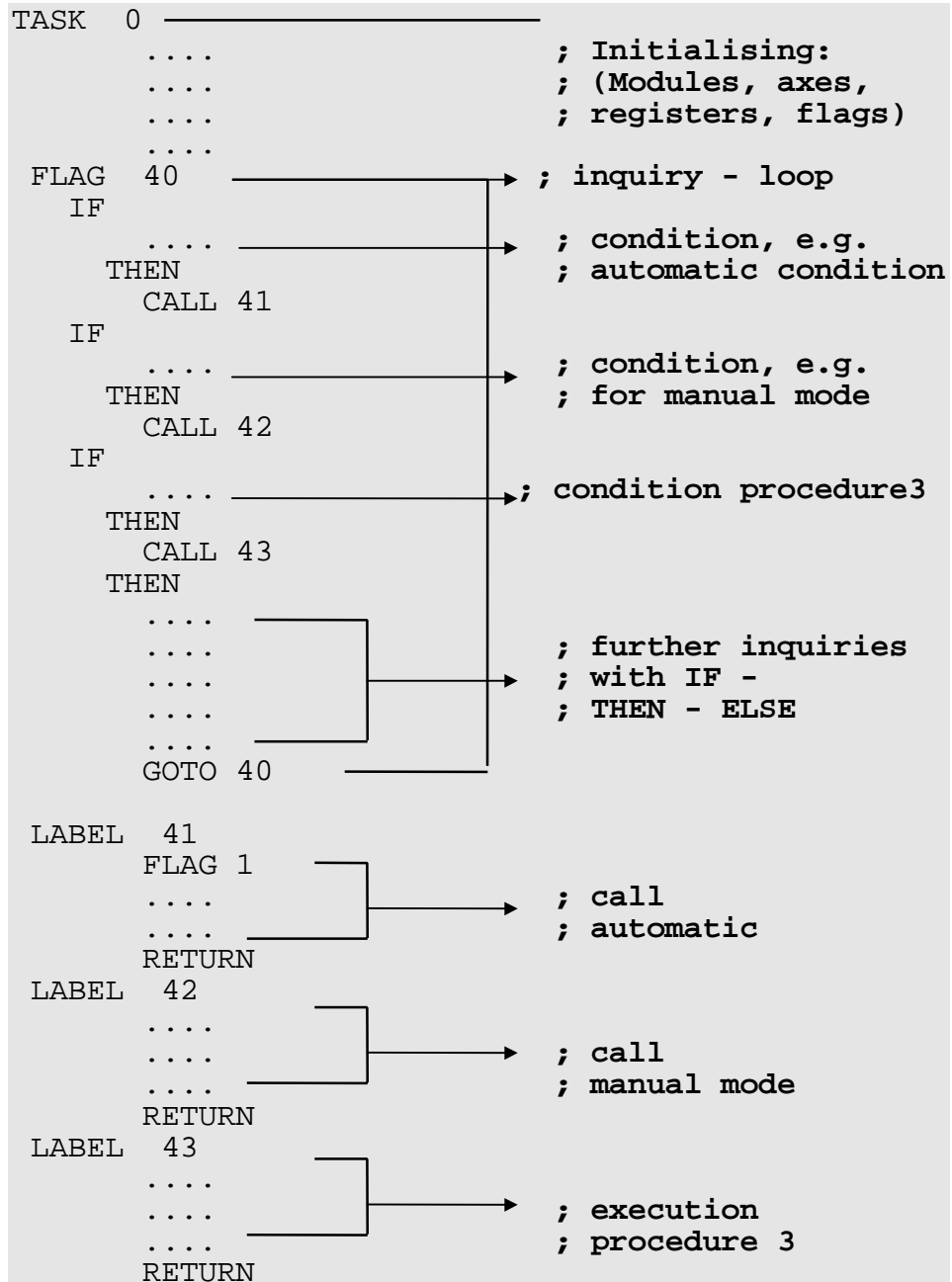
In this language, the processing time is not dependent on the program length, but on the number of parallel programs (tasks) that have been used. Thus, not too many of them should be running simultaneously.

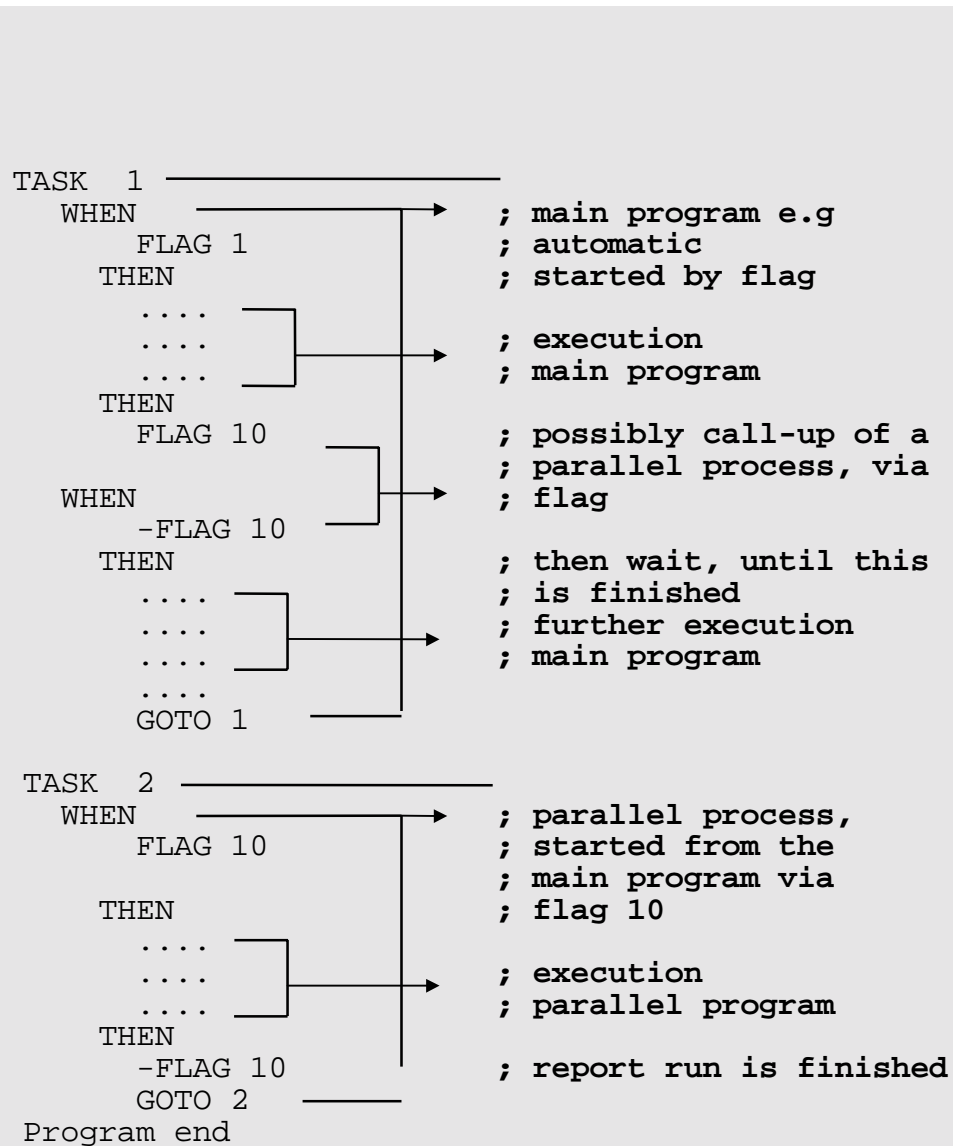
Basic
Instruction
WHEN
...
THEN
...

The basic instruction **WHEN**...(condition)... **THEN** ...(output).. can thus be used very well for direct process description, as (in contrast to the IF/THEN instruction for BASIC or PASCAL) meeting the condition is **waited for**, and only then the output is carried out. This means, that, other than with the known PLC languages, the order of inputs corresponds to the program sequence.

If meeting the condition is not to be waited for, but if only a logic decision is to be made, **IF . . . THEN . . . (ELSE)**... must be used. This directly corresponds to the BASIC instruction **IF . . THEN . . . (ELSE)**... , i.e. depending on the logic state of the condition, one of the two outputs will be made, which can also be blank. (The **ELSE** branch can be left out totally, even including the "**ELSE**".)

A basic program structure can, for example, be made up as follows:





Description of the individual parallel tasks and of their coordination:



TASK 0

TASK 0
must always
be present

In **TASK 0**, first an initialisation of additional boards and of the outputs, flags and registers is made. Normally, a reference run is necessary for axis control, which can also be programmed at this stage.

TASK 0:
Initialisation,
request
loop for
inputs, keys,
etc.

After this, there will be a number of **IF** conditions. This way, inputs can be enquired on. This **enquiry loop** will continuously be run through; in consequence, no **WHEN** - instruction must be written there. This way it is possible to recognise a met condition and to react immediately.

Then a jump to the respective flag will be made at once (**FLAG 41**, **FLAG 42** etc.), where those instructions, which must be executed, are written. Those can also be instructions for an automatic control task or something similar. Yet, there must not any instructions be given at this position, which will need a long time for execution. Otherwise, the inquiry loop will not be run through any more, neither will any new inputs be considered further.

In the case of "call-up automatic" a task could be started, for example, by which automatic controlling could be carried out. The same way another task could be started under "call-up manual mode", by which a manual controlled task could be carried out.

TASK 1**TASK 1:
Automatic
task,
control of the
main functions**

In this task the **main controller program** has been written. This main program will be started by `FLAG 1`. This could be started out of `TASK 0 (LABEL 41)`. Thus, the condition for this process could be a start key (input). As soon as this input has been activated by pressing the key, `FLAG 1` will be set and thus the start function be triggered.

Then the main program will carry out the instructions, possibly start another parallel program (here by `FLAG 10`), and will then come to an end by defined conditions. This main program must be seen as an endless loop, if it really is in automatic mode. By certain interrupt conditions, for example by a stop key or an error report, this endless loop will be left and the end will be reached.

From the end a jump will be made to the beginning of the task, for the task to be ready to be called again. If `FLAG 1` has not been reset in the meantime, the controller will be restarted immediately, which might even have been desired. Otherwise the flag would have to be reset to the beginning before `GOTO`.

TASK 2

This is an example of a simple task called via flag to, for instance, control operator guidance.

2.1.1 Rules for Program Structure - Task Structure

Definition of a Parallel Branch (Tasks)

By 32 tasks
parallel
program parts

For programming more complex processes and for the realisation of subroutines, there is the possibility in PROCESS-PLC to write several parallel program parts. Altogether, 32 independent program parts are possible. These can also be connected via `FLAG` oder `REGISTER`, which means they are made dependent on one another. One of these program parts can be called "**parallel branch**" or "**task**".

`TASK 0`
must always
be there

A parallel branch always has to start with a `TASK` instruction. The `TASKs` for parallel branches have got numbers 0 to 31 (inclusively). `TASK 0` is always there. If no further `TASK` of a number smaller than 32 is input, "parallel branch" 0 will remain the only one; thus, no parallel processing will be carried out by the controller.

Ascending
and complete
numbering

If a parallel branch is needed, a `TASK` number 1 must be input. For all parallel branches that are needed further, the next number must be selected in ascending order and without leaving one number out.

In the editor, `TASKs` are marked by a line. This line is to distinguish between two programs.

GOTO destinations only in the context of one task; never exceeding one task

A parallel branch is a program that is complete in itself. This means, that out of a parallel branch there must not be a GOTO to a LABEL, which is also used by another parallel branch. At the end of a parallel branch, a GOTO to a LABEL must be part of a parallel branch. Otherwise, the parallel branch will run "into the next one", which will lead to malfunctioning of the program.

Place global calls at the end of the program text

Subroutines ("calls") and functions are an exception. They can be called from another parallel branch without any problems. Basically, even simultaneous call-up of the same subroutine is possible from several parallel branches - yet, it should be placed at the end of the last task (place global calls at the end of the entire program text).

TASKS, which have been positioned at the beginning of a parallel task (0..31) can be used - as any other label - as destinations for GOTO instructions (GOTO).

A program without multitasking, i.e. without parallel branches, can be structured as follows:

```

TASK 0 -----
    ...
    ...
    ...           ; Program Text
    ...
    ...
    ...
    LABEL 50
    ...
    ...           ; Program Text
    ...
    GOTO 50
    
```

The parameter numbers of the TASK or FLAG instructions can be given symbolic names

This means, that at the end a jump to label 50 has been made. This label can, of course, be given any number from 32 to 999 or a symbolic name. The jump at the end must be made to a LABEL that is already available. A jump can also be made to TASK 0 directly.

Example of structuring a program with three parallel branches:

```
TASK 0 -----
...
LABEL 101      ; All set labels are, of course,
                ; only meaningful,
LABEL 105      ; if they are jumped to.
                ; The jumps can be programmed to
LABEL 102      ; go in all directions to your liking,
                ; even forward,
                ... ; yet only inside
                ... ; of one
                ... ; parallel branch (Task).
GOTO 101
TASK 1 -----
...
LABEL 200
LABEL 201
...
GOTO 201
...
GOTO 1
TASK 2 -----
...
...
...
GOTO 2
```

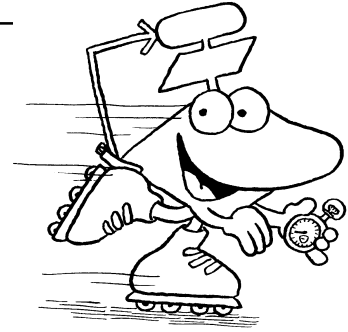

Parallel Processing (Task Switching)

Rules for task switching

Parallel processing of individual parallel user program branches is managed by the operating system of the controller. There are certain rules, according to which the processing of individual program parts is started and changes from one parallel branch to another can be made. To know these rules might be interesting for the advanced programmer:

Rule 1:	Processing of the user program always starts with <code>TASK 0</code> .
Rule 2:	Switching between parallel branches will always be carried out after the following instructions or in the following situations: <ul style="list-style-type: none">• DELAY• USER_INPUT• In case of a WHEN - condition, which has not been met
Rule 3:	The following three further conditions for task switch can be influenced by the user with the help of the flag (flag number in the controller manual). The default value after reset will be written in square brackets. <ul style="list-style-type: none">• After a certain time (Task switch-Timeout) [active]• After GOTO - instruction [active]• After an IF condition that has not been fulfilled [deactivated]

Remarks on Rule 3



Task switch
after a
defined time

a) Task Switch Timeout (Flag 2056)

After a certain time the task will be changed, when the present instruction has been carried out, without meeting any other requirement. The time, which has been stored in the "task-timeout time" register (in units of ms), can be changed by the user.

After reset if flag 2056 = 1, (if it is set) the "task timeout time" is loaded with 20 (ms)

Without changing flag 2056 and special register "task timeout time" this task switch timeout is activated. Thus, switch to another task will be made after 20 ms as the latest.



Note:
An output instruction cannot be terminated by timeout and will thus be processed completely.

Task switch
before GOTO
instruction

b) After GOTO - instruction (flag 2057)

If flag 2057 has been set, the task will be changed after each GOTO instruction.

This flag is also = 1 after reset, this means, it has been set.

c) IF condition, not fulfilled (flag 2058)

Task switch
before
IF condition
that has not
been met

If flag 2058 has been set, task switch will be carried out after each **IF** condition, if it has not been met (This means, that the **THEN** branch will be preferred to the **ELSE** branch).

This flag will be = 0 after reset, this means, it will not be set.

d) **Flag 2056 to 2058**
<---> **Special register "task switch conditions"**

Overlapping
flags 2056 to
2058 with
special
register

These three flags are overlapping with the "task switch conditions", where the three bits represent the three register bits of lowest value (flag 2056 <--> Bit 0, etc.), while the register bits of higher value are not being used.

The defined values after reset result in value 3 for the special register "task switch conditions".

By the instruction

```
REGISTER_LOAD [ Task switch conditions with 0]
```

the three conditions can make an exception from rule 3.

2.1.2 Special Registers / Flags for Task Control

Flag 2056

If this flag has been set, the task will be changed after a certain time, which has been written into the "task timeout time" special register, without any other condition having come true. Of course task switch during execution of an instruction cannot be made, yet immediately after this. After reset, this flag is set; this means, the task switch condition is active.

Special Register "Task-Timeout Time"

In this register, the task-timeout time is defined in ms (milliseconds). After reset, this register will be set on 20 with the consequence, that the task will be changed after 20 ms as a maximum.

Flag 2057

If this flag is set, **the tasks will be switched after each GOTO instruction**. This flag is set after reset; this means, that without changing the flag, the tasks will be switched after each GOTO instruction.

Flag 2058

If this flag has been set, **the tasks will be switched after each IF condition that has not been fulfilled**. This means that the THEN branch is preferred to the ELSE branch.

After reset, the flag will not be set.

The **special register "taskswitch conditions"** is overlapping with flags 2056 to 2063, flags 2059 to 2063 excluded. Thus, only the three bits of lowest value are used. By assigning a certain value to this register, all task switch conditions can be defined as desired. After reset, the register value will be 3, as can be concluded from the values of the flags mentioned above.

In the following example the same task switch conditions are set in two different ways:

Exemplary
numbering
DELTA

Example:

```
REGISTER_LOAD [ 61467 with 4]    -FLAG 2056  
                                  -FLAG 2057  
                                  FLAG 2058
```

At the left, values 0, 0 and 1 will be written into flags 2056, 2057 and 2058 of register 61467. At the right, the same is done with the help of the three `FLAG` instructions.

2.2 Symbolic Programming

Symbolic name instead of numeric parameter:

IN iStart

instead of

IN 101

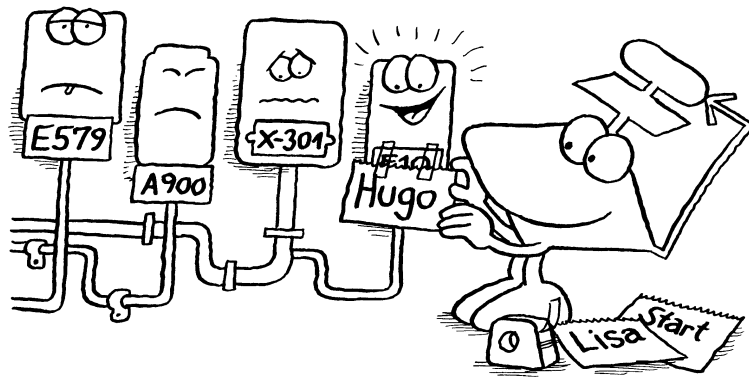
The numeric parameters of the programming language can be replaced by symbolic names. This helps for clarity of the program and makes getting acquainted with the source text easier. It is important to replace only the numeric part of the parameter by symbolism. All other parameter components stay the same.

Thus, for example,

Input 101	can be named	Input Start
or		
Register 100	can be named	Register No.OfPieces

Instead of the numeric parameters, which are not very meaningful, symbol names can be used to make the program easier to understand.

See also the explanations on the symbol editor in



2.2.1 Recommendations on Symbolic Notation

Standardising of symbolic names

It is helpful to set up some rules for the choice of symbolic names. Standardising of symbols for inputs, outputs, flags, registers, etc. should be made in the symbolic name, for example:

iStart for a start input, or

rDestinationPosition for a destination position register of an axis

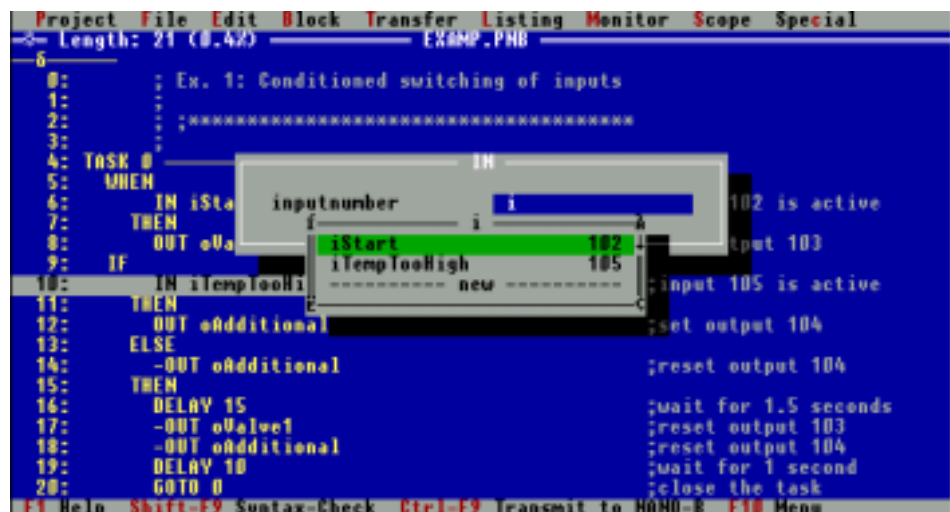
Placing standardising small letters in front of the actual symbolic names has got two important advantages:

Improved readability

- Readability of the program code is improved, as in each symbolic name there is information not only on its meaning resp. function, which is given by the naming itself, but also on its type (input, output, flag, register, etc.).

List of all symbols of the same type

- It is possible to have, for example, all inputs displayed in the dialogue window by inputting i?. You need not print or memorize the symbols any more; instead, you will be given uncomplicated online help.



The following pre-posed small letters are recommended for standardising:

Recommended contractions for standardising the symbolic names

Contraction	Type	Sub-Type
i	input	-
o	output	-
f	flag	-
fs		special flag
r	register	-
rs		special register
rf		floating point reg
rm		slave register on modules, boards
rt		text registers
rv		VIADUKT registers
l	labels	-
lm		for WHEN_MAX
le		error subroutines
t	task	-
n	number	-
ne		error report
nc		commands
nb		definition of a register bit
nv		call-up of VIADUKT masks
ax	axes	

2.2.2 Examples of Symbolic Notation

The following program sequence is to illustrate the recommendations on symbolic notation:


```

0:      ; *****
1:      ;
2:      ;   Example 1: Conditioned activating of outputs
3:      ;
4:      ; ;*****
5:      ;
6: TASK 0 -----
7:   WHEN                               ;Wait for
8:     IN iStart                         ;Input 10 activated
9:     THEN
10:    OUT oValve1                       ;Set output 8
11:   IF                                  ;Input 11 aktive?
12:     IN iTempTooHigh
13:     THEN
14:       OUT oRefrig                     ;Set output 9
15:     ELSE
16:       -OUT oRefrig                    ;Reset output 9
17:     THEN
18:       DELAY 15                         ;Wait for 1,5 seconds
19:       -OUT oValve1                     ;Reset output 8
20:       -OUT oAddition                   ;Reset output 9
21:       DELAY 10                         ;Wait for 1 second
22:     GOTO 0                             ;Close the task
Program end

```

The following symbol definitons have been made

iStart	10	;Input: Start input
iTempAddHigh	11	;Input: Temperature sensor
oAddition	9	;Output: Refrigerating set
oValve1	8	;Output: Suction valve

2.3 Remarks on the Program Examples



Note:

In the following program and instruction examples, both symbolic and numeric programming will be applied. Symbolism is not used, if it is better to communicate with the hardware directly via register number, or for didactic purposes, if symbolic presentation would complicate the description.

In case of numeric presentation the register numbers of the NANO-B PROCESS-PLC will be used; exceptions will be marked.

3. The Programming Language

In this chapter all instructions, which are available for programming the controller will be described (by one example or more), then it will be demonstrated (by one example or more), how they can be used.

3.1 Overview over Instructions

The instructions have been listed in the following table:

Arithmetic and Boolean Characters:

> = < + - * / () #

(*Chapter 3.3 Boolean Expressions and Chapter 3.4 Arithmetic*)

Commentary Characters:

;
(*Chapter 3.11.3*)

PROCESS-PLC

Instruction Set

DR	DISPLAY_REG	output of register contents onto LCD or printer
DT	DISPLAY_TEXT	output of texts onto LCD or printer
D2	DISPLAY_TEXT_2	depending on a register, one of two texts can be chosen
OU	OUTPUT NUMBER	setting, resetting, querying of a digital output
U	USER_INPUT	input of register values by the user, with the help of the LCD
BC	BIT_CLEAR	the bit of a register is cleared or queried for zero
BS	BIT_SET	the bit of a register is set or queried for 1
TH	THEN	IF...THEN...ELSE, WHEN...THEN
DF	DEF_FUNCTION	the beginning of a function definition is marked
ED	END_DEF	the end of a function definition is marked
IN	INPUT NUMBER	a digital input is queried
IF	IF	IF...THEN...ELSE
LI	LIMITS	1. it is queried, whether the register is inside certain limits (query) 2. a register is placed between certain limits by force (assignment)
AX	AXARR	1. it is queried, whether the axis has been stopped (query) 2. axis is stopped (assignment)
AP	ACTUAL_POS	the actual axis position is queried

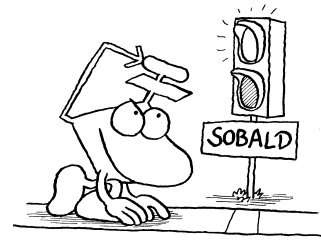
CO	COPY	a register area is copied
NP	NOP	this command is of no effect, yet, a processing time is needed (test purposes)
CF	CLEAR_FLAGS	a flag area is cleared
RL	REGISTER_LOAD	a value is written into a register (direct, indirect, doubly indirect)
LA	LABEL	GOTO label for program flow
F	FLAG	setting, resetting, querying a flag
NG	N-GET-REGISTER	a register of a slave controller is loaded into the memory of a master controller, JETWay, fieldbus
NO	NOT	logic NOT (an input condition is inverted)
NS	N-SEND-REGISTER	a register of a master controller is loaded into the memory of the slave controller, JETWay, fieldbus
OR	OR	logic OR (input condition)
P	POS	an axis is positioned with speed v onto position pos
RD	REGDEC	a register value is decremented by 1
RE	REG	register command, e.g. REG 100 = 1234
RI	REGINC	a register value is incremented by 1
RC	REG_CLEAR	a register area is set to 0
RZ	REGZERO	a register is set to zero, or a register is queried for zero
RT	RETURN	a subroutine or a function is finished
WH	WHEN	WHEN . . THEN

SF	specialfunction	call-up of certain special functions, e.g. trigonometry
WM	WHEN_MAX	WHEN_MAX . . THEN; additionally a time can be input, after which a subroutine (e.g. bugfix) can be called
EL	ELSE	IF . . THEN . . ELSE
G	GOTO	control of program flow
ST	START-TIMER	a time register is started
TA	TASK	label for task start
TB	TASKBREAK	a task is broken
TC	TASKCONTINUE	a broken task is continued
TR	TASKRESTART	broken task is started from the beginning
CA	CALL	a subroutine is called up
DE	DELAY	task-processing is broken for a certain time
WO	WOR	OR linkage of registers
WA	WAND	AND linkage of registers
WX	WXOR	exclusive OR linkage of registers
TE	TIMER-END?	time-register is queried

PROCESS-PLC Numbers

Abbr.	Command	Remarks
NB	number (binary)	the numbers are input as binary numbers: b010101010101010101010101
ND	number (decimal)	the numbers are input as decimal numbers: 1234
NH	number (hexadecimal)	the numbers are input as hexadecimal numbers: hFA23CD

3.2 Basic Instructions



3.2.1 Waiting Condition **WHEN** ... **THEN**

Syntax:

```

WHEN
    <Condition>
THEN

```

WHEN
waits, until
condition has
been met

Meaning:

Fulfilling the <condition> is waited for. Only then the next instruction will be given (after **THEN**).

Elementary
conditions:

Input
output,
flag,
register bit,
arithmetic
comparison

The condition can either be a flag, an input, an output, a certain register bit or the result of an arithmetic comparison.

These "elementary conditions" can be combined into a Boolean expression, the result of which is to be the condition. For those expressions brackets can also be used. If the sequence has not been defined otherwise by brackets, the Boolean expression will be processed from the beginning to the end, while the result can be interpreted as a condition. (See also Boolean Expressions *Chapter 3.3 Boolean Expressions*).

Examples:

```

1)  WHEN
      IN iStart
      Flag fTaskEnable
THEN

```

When the input `IN iStart` is active, and when `FLAG fTaskEnable` has been set, the program will be continued by the instruction following after `THEN`.

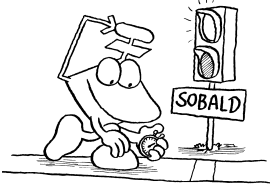
**Note:**

If in a Boolean expression nothing has been written between two or more elementary conditions, this will automatically be interpreted as an AND operation.

```
2)  WHEN
      REG rVoltage
      =
      50
      THEN
```

When the register `REG rVoltage` has got value 50, processing of the program is continued. (The value of `REG rVoltage` can, for example, be changed in another task, or it can represent an analogue voltage value.)

3.2.2 Waiting Condition `WHEN_MAX ... THEN`



Syntax:

```
WHEN_MAX [Max.Time=<Time>, Subroutine= <Subroutine>]
    <Condition>
THEN
```

WHEN
Fulfilment of
condition is
waited for.
In addition:
Timeout time

Meaning:

Fulfilment of <condition> is waited for; only then the next instruction will be given (after `THEN`).

If the maximum time has run out before the condition has been met, the subroutine will be called.

The condition is either a flag, an input, an output, a certain register bit, or the result of an arithmetic comparison.

Example:

```
1)  WHEN_MAX[Max.time=z5s, Subroutine.=leError]
      REG rPressureInCylinder
      >
      50
      THEN
      ...
      ...
      LABEL leError
      -OUT oHydraulicPump
      DISPLAY_TEXT[#0,cp=1,"_fix error"]
      DISPLAY_TEXT[#0, cp=25, "after this F1"]
      WHEN
      FLAG fKeyF1
      THEN
      OU oHydraulicPump
      DISPLAY_TEXT[#0, cp=1, "_"]
      RETURN
```

When the register `REG rPressureInCylinder` is greater than 50, the instruction after `THEN` will be continued. If the task remains at the `WHEN` condition that has not been met, the error routine `leError` will be called up. After the error routine has been processed, it will return to the `WHEN` condition by the `GOTO` instruction. This means `REG rPressureInCylinder` is waited for by the task to become greater than 50 - what should be the case now, as the error has been corrected by the error routine.

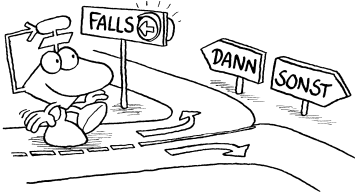
**Note:**

After execution of the error routine, the program will return to the `WHEN`-condition by the `RETURN` instruction.

```
2)  WHEN_MAX[Max.time=n2s, subrout.=leEmergStop]
      IN iPneuAxis
      THEN
      ...
      ...
      ...
      LABEL leEmergStop
      -OUT oPneuAxis
      AXARR Axis=X
      AXARR Axis=Y
      AXARR Axis=Z
      OUT oWarningTone
      DISPLAY_TEXT[#0, cp=1, "_EmergStop"]
      ...
      ...
      RETURN
```

When the input `iPneuAxis` is activated, further instructions will be given after `THEN`. If the task remains at the unfulfilled `WHEN` condition longer than 2 seconds ($8\text{max. time} = n2s$), the error routine `leEmergStop` will be called up. After being processed, the error routine will return to the `WHEN` condition by the `RETURN` instruction.

3.2.3 Branch Condition **IF ... THEN ... (ELSE)**



Syntax:

```

IF
  <Condition>
THEN
  <First instruction>
ELSE
  <Second instruction>
THEN or WHEN or IF

```

Meaning

If `<condition>` has been fulfilled, the first instruction will be carried out. If `<condition>` has not been fulfilled, the second instruction will be carried out.

After another `THEN`, a following `WHEN` or another `IF`, processing of the program will be continued.

For `<condition>` another Boolean expression - similar to the `WHEN` instruction must be written.

Both the first and the second instruction can consist of several sub-instructions. Yet, they can also be left out.



Note:

- The `THEN` branch will be closed by `ELSE`, or `THEN`, `WHEN`, `IF`.
- The `ELSE` branch will be closed by `THEN`, `WHEN`, `IF`.
- Both will **not** be closed by `FLAG`.

Examples:

```
1)  IF
      IN iStopper
      THEN
        FLAG fPartPosOK
      ELSE
        -FLAG fPartPosOK
      THEN (or IF or WHEN)
      ...
```

In this example it is demonstrated, what the IF - THEN - ELSE - structure normally looks like.

If input IN iStopper active (<condition> is fulfilled), FLAG fPartPosOK is set, the instruction in the ELSE branch is left out and the process continued after the second THEN.

If IN iStopper is not active (<condition> not fulfilled), the instruction in the THEN branch is left out, the flag is reset and the process continued by the instruction after the second THEN.

```
2)  IF
      FLAG fsKeyF1
      OR
      FLAG fsKeyF2
      THEN
      ELSE
        OU oLamp
      THEN (or IF or WHEN)
      ...
```

If the condition has been met, (FLAG fsKeyF1 or FLAG fsKeyF2 have been set) no steps are taken (THEN branch is not used). If the condition has not been met, output OU oLamp is switched on (ELSE branch).

```
3)  IF
      IN iStart
      THEN
        OU oHydAxis
        OU oWarningTone
        REGISTER_LOAD [ rLength with 25]
      ELSE
      THEN (or IF or WHEN)
      ...
```

If input IN `iStart` is active, the outputs OU `oHydAxis` and OU `oWarning` tone are set, while value 25 is loaded into register REG `rLength`.

If IN `iStart` is not active, though, the program will jump to the second THEN to continue with the next instruction.

```

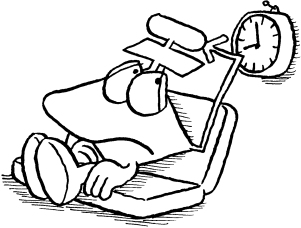
4)  WHEN
      FLAG fStation1
    THEN
      IF
        IN iKey1
      THEN
        OUT oPump
        DELAY 5seconds
        -OUT oPump
      ELSE
        OU oSignalTone
        DELAY 2Seconds
        -OUT oSignalTone
      IF
        FLAG fError
        IN iStopper
      THEN
        OUT RelaisOFF
      ELSE
        WHEN
          -IN iStopper
        THEN
          GOTO 100

```

In Example 4) a small program part is shown, where the structure of the WHEN-THEN instruction and of the IF-THEN-ELSE branch have been illustrated.

Recognition of these structures is supported by different indentation positions of the instructions in the SYMPAS editor.

3.2.4 The **DELAY** Instruction



Indirect
addressing
of the delay
is possible

Unit of **DELAY**
parameter:
100ms

The

DELAY <Time of delay>

instruction

serves for programming of a defined time, while processing of the task is to be held for this time. The program is simply to be inactive during the delay time. The time of delay is the only parameter which must be input.

The delay time can also be defined indirectly via register.

Generally, the **DELAY** parameter is defined in units of **100 ms**. A delay value of 10 means one second.

This unit can be defined by changing the "users' time base". Please be careful not to use too small units, as the operating system would be occupied with too many management functions. These mechanisms can be looked up at the description of the time registers. At this place they shall not be dealt with any further, as they will only be needed for very special applications.

Example:

```
DISPLAY_TEXT [#0, cp=0, "_Wonderful !"]
DELAY 50
DISPLAY_TEXT [#0, cp=0, "_"]
```

In this example, a text is displayed in the user interface, then there is a delay of 5 seconds before the display is cleared again.

Time Registers

In connection with the `DELAY` instruction it is important to mention the time registers as well: These are the registers the parameter value of the delay instruction is written into. When such a register is not zero any more, it will be decremented by one after each time unit. The delay instruction only serves the purpose of loading this register and then to wait, until its value is zero.

Please note about the time registers: Each task has got its individual time register (the number of which can be looked up in the respective controller manual). In the following example the task-time registers are shown as examples of the NANO-B.

Exemplary
numbering in
the NANO-B

Example: Task-Time Register

Task 0	Register 2300
Task 1	Register 2301
...	
...	
Task 31	Register 2331

In some applications a time register is to be activated, while still further instructions are being executed by the program. This can be managed by writing into the respective register using the `REGISTER_LOAD` instruction. Later the time register can be integrated into a comparison by simple enquiry.



Note:

Please be careful not to use the `DELAY` instruction simultaneously with the time register of the same task, as this might cause situations, where there is no time for the delay to expire any more.

This danger can easily be avoided: Just take a time register of a task instead, in which a `DELAY` instruction will never occur. This way, both the instruction and one (or even more) time register can be used in the same task.

Examples:

```
1a)  TASK 0 -----  
      ...  
      ...  
      DELAY 10  
      ...
```

```
1b)  TASK 0 -----  
      ...  
      ...  
      REGISTER_LOAD[rsTaskTimeReg with 10]  
      WHEN  
      REGZERO rsTaskTimeReg  
      THEN  
      ...
```

In their function, both programs are identical. First, the time register is set and then it is waited for to become zero. That is exactly the function of the `DELAY` instruction.

3.3 Boolean Expressions

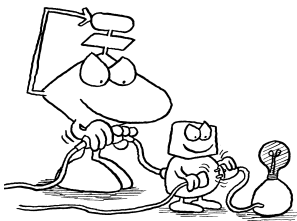
Boolean expressions are either true (1) or false (0)

All those expressions are conditions which have either been fulfilled or not. If the condition has been fulfilled, value 1, if it has not been fulfilled, value 0 is assigned. The Boolean expressions always appear in a `WHEN` or `IF` branch as input conditions, this means, never after `THEN` or `ELSE`!

Everything that is written between `IF` or `WHEN` and the next `THEN`, will be understood by the controller as a Boolean expression

Elementary Conditions

These are very simple expressions, which can consist of only one instruction, and which in the following will be called **Elementary Conditions**. Some of them are:



- flags
- inputs
- outputs
- single register bits
- arithmetic comparisons (this means, `REGZERO`)
- `AXARR`

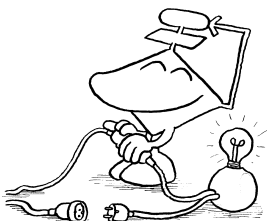
Flags, inputs, outputs, and register bits can be (logically) connected with each other.

For elementary conditions, the following characters are also permitted:

`= < > () #`

and the following instructions for connections:

NOT OR XOR AND



Elementary conditions as default, connected by **AND**

If there is no link instruction between two elementary conditions, they will automatically connected by **AND**! In output conditions, **AND** must be used.



Note:

Three levels of brackets are allowed for Boolean expressions.

3.3.1 Phrasing Elementary Conditions

Flags

Flags have got the value 1 or 0

Generally spoken, a flag is a **one-bit register**. It can have value 1 or 0. A set flag corresponds to value 1. A flag that has not been set corresponds to value 0. By input of a negative sign in front of the flag number into the input field, easy access can be made onto:



Examples:

```
1)      IF
          FLAG fStartTask
```

FLAG fStartTask set? (= 1 ?)

```
2)      IF      -FLAG PosOk
          IF      NOT
                FLAG PosOK
```

FLAG PosOK not set? (= 0 ?)
Both phrasings have got the same result.

Inputs and Outputs

Access to inputs and outputs, e.g. with
IN 101
 or
-OU 108

These have either got value 1 or 0 (set or not set). As it is the case for flags, access to the deactivated input, respectively output, can also be made here by preposing a negative sign.

Examples:

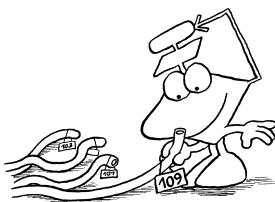
```
1)  IF
      IN iStart
```

Input IN iStart active? (= 1 ?)

```
2)  IF          IF
      -OUT oLamp      NOT
                        OUT oLamp
```

Output OU oLamp not active? (= 0 ?)

Both phrasings have got the same meaning.



Access to register bits by the instructions
BIT_SET
 and
BIT_CLEAR

Access to register bits can be made by the **BIT_SET** and **BIT_CLEAR** instructions (see also chapter 3.6.4 Register Bit for these instructions).

BIT_SET is 1, if the defined bit has got value 1, otherwise it is 0.

BIT_CLEAR is 1, if the defined bit has got value 0 (if it is cleared), otherwise it is 0.

Example:

Register 1 is to have (binary) value 100110 = 38.

```
BIT_SET [Reg.1, Bit 2] is 1 (Bit is set)
BIT_CLEAR [Reg.1, Bit 2] is 0 (Bit has not been cleared)
BIT_CLEAR [Reg.1, Bit 4] is 1 (Bit is cleared)
BIT_SET [Reg.1, Bit 0] is 0 (Bit has not been set)
```



Arithmetic Comparisons

Arithmetic comparisons are also either true or false

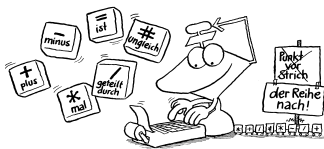
Arithmetic comparisons are also always either true or false. They are given value 1 for true and value 0 for false.

Examples:

```
1)      IF
        REG rCounter
        =
        0
        THEN

        IF
        REGZERO rCounter
        THEN
```

If the register REG rCounter has got value 0, this expression is true (= 1). If REG rCounter is not 0, the expression is false (= 0). This example exactly corresponds to REGZERO rCounter, as it is queried by the counter, whether the value of the given register is 0.



```
2)      IF
        REG rNumber
        >
        10
        THEN
```

If the content of register REG rNumber is greater than 10, this expression is true. If the register value is smaller than or equals 10, the expression will become false (= 0).

**Note:**

The `REG` instruction or arithmetic and logic connections of registers or results of functions without a comparison operator are implicitly compared with 0.

Example:

```

IF          IF
    REG rTestReg    REG rTestReg
    THEN           #
                   0
                   THEN

```

**Remark:**

In arithmetic comparisons it is possible to use **arithmetic expressions** and **word processing instructions** right and left of the comparison operator (see to this aspect *Chapter 3.4 Arithmetic*).

Example:

```

    WHEN
        REG 100
        >
        REG 1000
        *
        135
        +
        REG 1001
    WAND
    h00F080
    THEN

```

At this part of the program the value of register 100 is waited for to become greater than the result of the arithmetic and logic connections on the right hand side.



Note:

The combinations "> =" resp. "< =" are **not** permitted.

3.3.2 Examples of Connected Expressions

```

1)  IF
      FLAG 1
      FLAG 2
      OR
      IN 101
      THEN

```

Boolean Expression:

(FLAG 1 AND FLAG 2) OR IN 101.

This expression is true, if both flag 1 and flag 2 have been set, or if input IN 101 is active.

2a)

```

      REGISTER_LOAD [ rCounter with 10]
LABEL lLoop
      ...
      REGDEC rCounter
  IF
      NOT
      REGZERO rCounter
  THEN
      GOTO lLoop
  THEN
      ...

```

2b)

```

      REGISTER_LOAD [ rCounter with 10]
LABEL lLoop
      ...
      REGDEC rCounter
  IF
      REGZERO rCounter
  THEN
      ELSE
      GOTO lLoop
  THEN
      ...

```

In this example a loop has been realised, which is being run through 10 times. For this purpose, first the loop counter (Register `REG rCounter`) is loaded with the number of loop runs and is decremented by 1 in a loop (`REGDEC rCounter`). At the end of a loop, the loop counter (`REG rCounter`) is checked, whether 0 has already been reached. If this is not the case, the loop must be run through once more (Go to `LABEL lLoop`). In this case the comparison is made on two different ways: In one case the condition is negated and thus the jump is made from the `THEN` branch, in the other case the loop is closed from the `ELSE` branch. Both programs are identical referring to their function.

```
3a)      WHEN
          NOT
          (
          IN 101
          OR
          -IN 102
          )
          REG 100
          <
          20
          THEN
```

If the Boolean expression is '1' (this means, it is true) the program can be continued by `THEN`.

The expression:

```
NOT (IN 101 OR IN -102) AND REG 100 < 20
```

To let this expression be true, both partial expressions (before and after `AND`) must be true.

Although the first partial expression is a little confusing, it becomes clear after more detailed investigation: This partial expression is true, if (`IN 101 OR IN -102`) is false, if - **IN 101** is deactivated and **IN 102** is active.

The second partial expression is fulfilled, if the content of register **REG 100** is smaller than **20**.

To let the entire expression become true, both partial expressions (before and after the **AND**) must be true.

```
IN 101=0   AND   IN 102=1   AND   REG 100<20
```

```
3b)      WHEN
          -IN 101
          IN 102
          REG 100
          <
          20
```

This program part has got exactly the same function as the former one.

Thus it can also be realised that by simplifying such expressions a better overview can be acquired. If, on the other hand the machine is considered, it is easier to understand, if the more complicated expression is kept.

3.4 Arithmetic Expressions

These instructions can be used for making an input condition after `WHEN` or `IF` (arithmetic comparison; see former *Chapter 3.3 Boolean Expressions*), as well as in an output instruction (assignment of a calculation result to a register).

The design and the evaluation of a formula is identical in both cases, except the fact, that in an arithmetic comparison left of the comparison operator, a number or connection may, assignment to a register can, only be made to one register.

A value is assigned by an equal sign.

Assignment: =

The following instructions serve the description of arithmetic / logical expressions:

arithmetic operators: + - * /

logic operators: WAND, WOR, WXOR

Numbers: Binary Numbers
Decimal Numbers
Hexadecimal Numbers

Variable: REG <Reg.Number>

3.4.1 Numbers

Numbers can be input in three different ways:

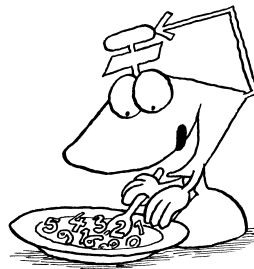
1. Decimal numbers by **ND** (Number Decimal)
2. Binary numbers with **NB** (Number Binary)
3. Hexadecimal numbers **NH** (Number Hexadecimal)

In the program text differentiation of the three number formats is made by writing 'b' in front of binary numbers and 'h' in front of hexadecimal numbers.

Floating point numbers are input in indirect mode

Only **integers** can be input. Direct input of a floating point constant is not possible.

Referring to the internal integer format (23 Bit and signs), numbers of seven decimal places, numbers of six hexadecimal places and numbers of twenty-four binary places can be input.



3.4.2 Arithmetic Expressions

The characters `+` `-` `/` `*` can be directly selected on the keyboard and input this way.



Note:

10 levels of brackets can be used for arithmetic expressions !



Syntax:

```

REG x
=
REG y
+
REG z
/
5
-
20

```

Meaning:

Register `x` is assigned the value of the following operation:

$$((\text{Register } y + \text{Register } z) / 5) - 20.$$

It can be seen that calculations are **not made according to the "./: before +/- "** rule, instead, one step after the other will be carried out following the sequence of operations.

An operator with the following number (operand) will always be selected by the controller to evaluate this

"partial calculation" in order to gain an intermediate result. Basing on this intermediate result, the next operator and operand will be selected in order to evaluate this partial calculation now. This procedure will be followed, until the last number value will have been selected.

Thus, an expression like $X = A + B - C / D - E * F$
 has got the following meaning: $X = (((A + B - C) / D) - E) * F$

In the following examples a difference is made between integer registers and floating point registers, as partially different possibilities are offered.

3.4.3 Assignment to Integer Registers

Decimal places are omitted, when a number is assigned to an integer register

When a number is assigned to an integer register, the decimal places are simply omitted, i.e. the numbers will not be rounded!

This means: $\text{Reg 0} = 10 / 3 = 3,333\dots$

-> Reg 0 contains value 3

$\text{Reg 0} = - 10 / 3 = - 3,333\dots$

-> Reg 0 contains value -3

Examples:

```
1)   REG 100
      =
      h0000A8
      +
      b00000000000000001001001110
```

Register REG 100 is given the sum of hA8 (=168) and b1001001110 (=590). The result is 758, what will then be the value of REG 100. (REG 100 = 758)

```
2)   REG 200
      =
      100
      /
      3
```

Register REG 200 is assigned the value 100 / 3 (= 33.333....). As in REG 200 only integers can be stored, the decimal places will be omitted. After this, the register value will be 33. (REG 200 = 33).

3)

The assignment $A = A - (B + C) / 2$ is to be calculated, which cannot be achieved by following the "./: before +/-" rule.

Solution:

An alternative expression (operation) must be phrased for the controller to calculate the expression step by step in the desired sequence.

Transformation: $A - (B + C) / 2 = 0 - B - C / 2 + A$

The expression at the right hand side exactly corresponds to the needed expression

(REG 1 = A, REG 2 = B, REG 3 = C). An operation cannot be started by "-" (negative sign), thus, a small trick has been applied (0 - ...).

```

REG 1
=
0
-
REG 2
-
REG 3
/
2
+
REG 1

```

REG 1 (A) is decremented by the value (REG 2 + REG 3) / 2 (average between B and C). Again, the new value is stored in register REG 1.

3.4.4 Assignment to a Floating Point Register

More detailed description of floating point registers can be found in *Chapter 3.6 Registers and Chapter 4.1*). Below, there are just some short remarks:

Numbers from -10^{15} to $+10^{15}$ are stored in the floating point register.

The preciseness of operations is 7 post-comma places, as the numbers are stored in a 32 Bit wide store.

The following floating point register ranges are supplied by various PROCESS-PLC:

PROCESS-PLC	Floating Point Register Range
PASE-E Plus	8960 to 9215
DELTA	62208 to 62463
NANO C	65024 to 65279
NANO B	? to ?
NANO A	-
MIKRO	-

In a program, no floating point numbers can be input, though (e.g. DELTA register: REG 62208 = 1,456 can **not** be input directly.)



Note:

Value 1,456 can be loaded into a floating point register by the assignment REG x = 1456 / 1000.

The rules for the calculation of floating-point expressions are exactly the same as for integer registers. 10 bracket levels, no "./: before +/-" operation.

Examples :

Numbering
of registers
demonstrated
by
DELTA

```
1)  REG 62208
     =
     12345
     /
     1000
```

Value 12,345 (= 12345 / 1000) is assigned to the floating point register (DELTA) REG 62208 .

```
2)  REG 62248
     =
```



```
REG 0
/  
REG 1  
+  
100  
*  
REG 62208
```

The value of the following operation is assigned to floating point register (DELTA) 62248.

```
REG 62248 = (  $\frac{\text{REG 0}}{\text{REG 1}}$  + 100 ) * REG 62208
```

3.5 Tasks, Labels, Jumps and Subroutines

3.5.1 Tasks, Flags and Jumps



The following instructions belong together and serve unconditioned jumps in a program. With the help of the **IF** instruction they can be turned into conditioned jumps.

TASK	LABEL	GOTO
-------------	--------------	-------------

The labels are to mark certain program points, which can be accessed by a **GOTO** instruction.

TASKS 0 to 31

Tasks must start with 0 and be numbered in continuous and ascending order

These labels serve the **marking of parallel branches**. In the program they must be applied in ascending order; this means, start with **TASK 0**, then mark **TASK 1** etc. Tasks must not be left out. They are presented as follows:

TASK 0 _____



LABELS 32 to 999

Labels serve as markers for jumps, or they mark the beginning of a subroutine

These labels serve as mere junctions, or - in connection with subroutines - to mark the start of a subroutine. They are displayed without the horizontal line:

LABEL 32

Jumps

GOTO
instructions
can be
defined
indirectly



After a `GOTO` instruction, processing of the program will be continued at the label the number of which is defined as a parameter in the `GOTO` instruction. The number of the label can also be defined indirectly, this means, by `REG x`, causing the jump to go to the label the number of which has been stored in register `REG x`.

Please be especially careful not to jump to a label, which has been allotted to another task.

Note:

Define special label numbers for each task, for example

```
TASK 0:  Labels 100 to 199
TASK 1:  Labels 200 to 299
TASK 2:  Labels 300 to 399
.....
```

Syntax:

```
TASK 0 _____
...
LABEL 100
...
...
GOTO 100
...
...
GOTO 0
```

Meaning

Program execution always starts with `TASK 0`

Program execution starts with `TASK 0` and then executes the instructions, until the `GOTO 100` instruction will have been reached. This way `Label 100` is jumped to. If this `GOTO` instruction stands in the output branch of an `IF` condition, the `GOTO` instruction will be carried out, while the program is going on, until `GOTO 0` will be reached.



Note:

Please be careful not to leave out this second jump for any reason. If there are more than one tasks, this could mean that `TASK 0` would go over into `TASK 1`.

Example:

```

1)          REGISTER_LOAD [ rCounter with 10]
          LABEL  lLoop
              ...
              ...
              ...
          REGDEC rCounter
          IF
              NOT
              REGZERO rCounter
          THEN
              GOTO lLoop
          THEN
              ...
  
```

In this example, a loop has been realised, which is run through 10 times.

Register `REG rCounter`, which was assigned value 10 in the beginning, is decremented by one in each loop (`REGDEC rCounter`). After this, a comparison will be made to find out, whether value 0 has already been reached. If it has not, another jump to the beginning of the loop (`LABEL lLoop`) is made. If value 0 has been reached, the program will be continued after the second `THEN`.

3.5.2 Subroutines

The instructions

CALL **RETURN**

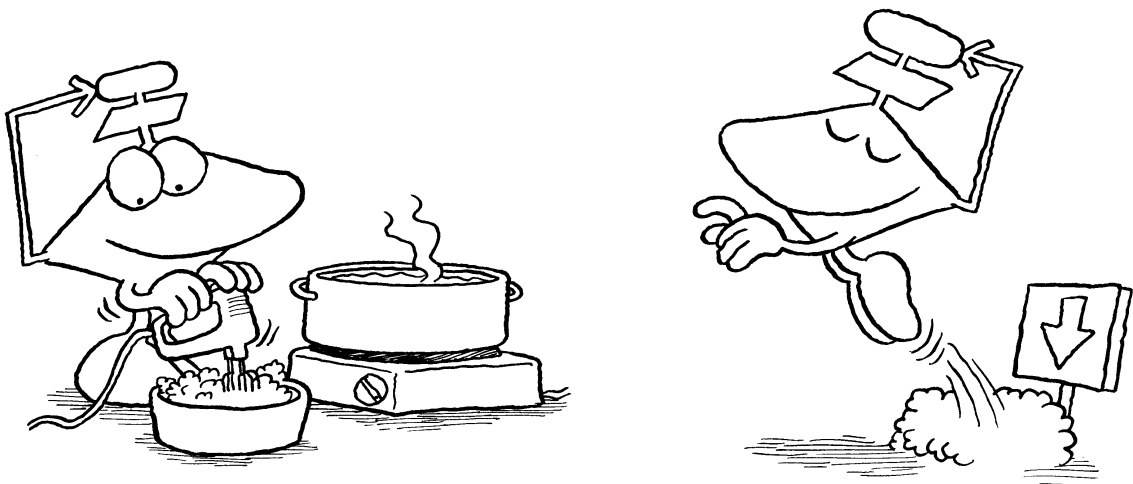
also belong together and serve the realisation of subroutines.

Subroutines are program parts which can be jumped to from any position in the program. Then, these partial programs will be processed. After completion the program will return to the position the subroutine has been called from.



Note:

20 subroutine levels are permitted.



Using subroutines make program texts clearer and more compact

As a return from a subroutine is always made to the program position the subroutine has been called from, the program parts, which are needed often and at various positions in the program, only need to be written once. Thus, memory can be saved and the program design becomes clearer.

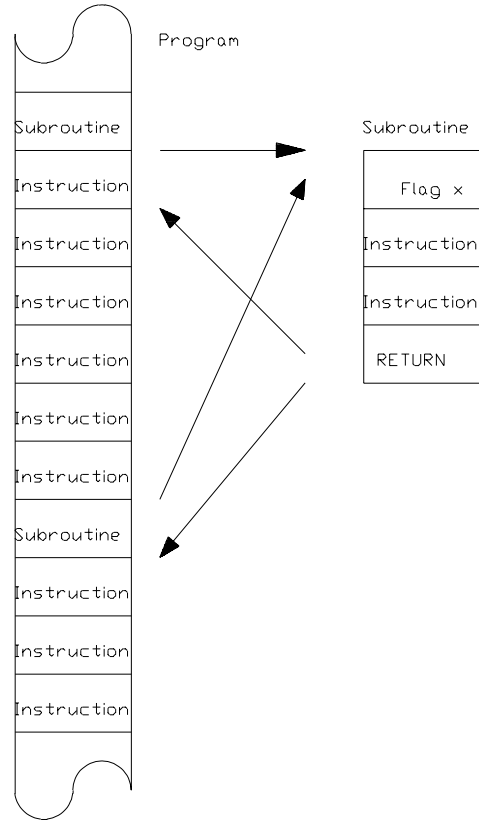


Figure 13: Subroutines

Syntax:

```

    THEN
        CALL leErrorHandling
        ...
        ...
    > LABEL leErrorHandling
        ...
        ...
        ...
        RETURN
    
```

Interpretation

When the subroutine reaches the `CALL leErrorHandling`, an immediate jump will be made to `LABEL leErrorHandling`. There, the instructions will be carried out, until the `RETURN` instruction will be reached to go back immediately to the position, where the subroutine has been called up and then to continue with the next instruction.

The subroutine will return to the call-up position after execution.

The difference between `RETURN` and `GOTO` instruction is, that, in the `RETURN` instruction, the call-up position is memorised by the program, in order to return there after execution of the subroutine.

Indirect addressing is also possible after giving the `CALL` instruction. Thus, a subroutine can be called up as follows:

Indirect call-up of subroutines

```
CALL R(100)
```

or

```
CALL R(rPointer)
```

In this case, the program starting at the label the number of which has been stored in register `REG 100` resp. `REG rPointer`.

Rules in Connection with Subroutines

1. **20 subroutine levels** are permitted.
2. Jumps out of subroutines are not permitted. A subroutine must always be finished with a `RETURN` instruction, otherwise new subroutines cannot be called up any more.
3. One subroutine can be jumped to from several parallel branches. Depending on the program, this can even be done simultaneously.

Example:

```
TASK Initialising _____
...
...
CALL lGlobal
...
...
TASK Automatic _____
...
...
...
CALL lGlobal
...
TASK Input/Output _____
...
CALL lGlobal
...
GOTO 2

LABEL lGlobal
...
...          * Subroutine Text *
...
RETURN
End of program
```


In this example, the subroutine is called once out of each of the three branches at the end of the last task (TASK In-/Output). This is absolutely legal, as has been mentioned before, yet, depending on the subroutine, it can lead to unwanted results in the case of **simultaneous** call-ups.

To give an example, display of a "false" register can be caused, if in a subroutine various registers are to be displayed, which should not be a problem, when indirect register definition is used (by the DISPLAY_REG instruction).

**Note:**

Global subroutines, i.e. subroutines, which can be called up by several tasks, must be written at the end of the last task.

Thus, special care should be applied, if the same subroutine is called up several times simultaneously. If necessary, subroutine calls can also be coordinated with the help of flags.

3.5.3 Functions



Functions are defined in the program header

The instructions

```
DEF_FUNCTION[<Function>, xy]           END_DEF
```

As it is common for high level languages, the function is defined by the programmer in the program header, in order to call it up in the program text whenever it is needed.

- Functions can be called up by transfer parameters.
- Functions can be defined by return parameters.
- Functions have got local variables and labels.
- Functions can be applied as Booleans or in output instructions.
- Functions do not differ in their call-up from system instructions.

Instruction libraries can be set up with the help of the functions

The programmer can create his own application specific instruction libraries with the help of the functions.

Definition of a Function

Functions are defined before **TASK 0**

The functions are defined at the beginning of the program (before **TASK 0**). First, the data framework is specified in a definition window (call-up by hotkey **(D)**, then **(F)**).

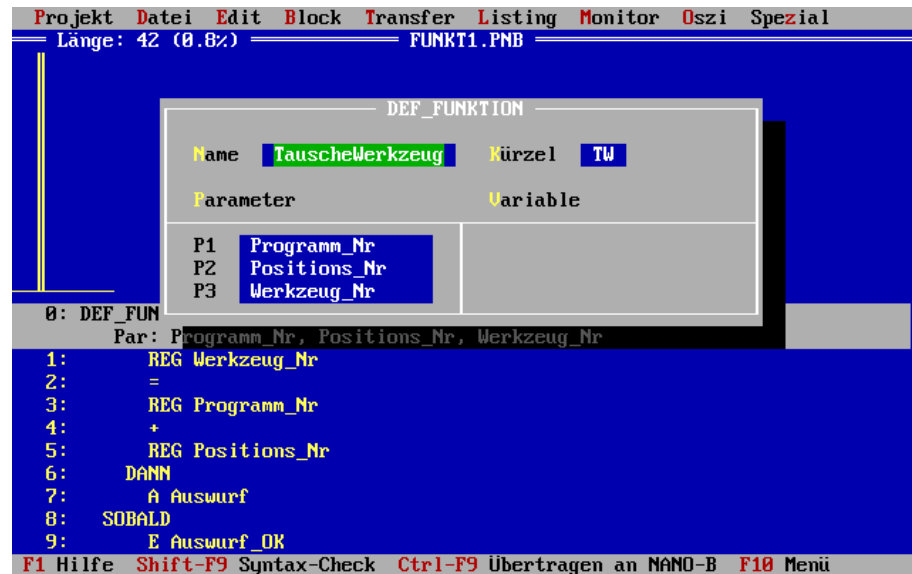


Figure 14: With the help of the definition windows the functions are parameterised

Definition of the Function Text

Functions are closed by `END_DEF.`

After confirming with (⌘), the function header will be displayed on the screen; after this, the function text can be input. It must be closed by `END_DEF.`

Function Call-Up in the Program Text

With the help of the instruction contraction, the functions are called up in the program text

Using the contraction of the function name, which has been input in the definition window (see figure above), the function is called up in the program text.

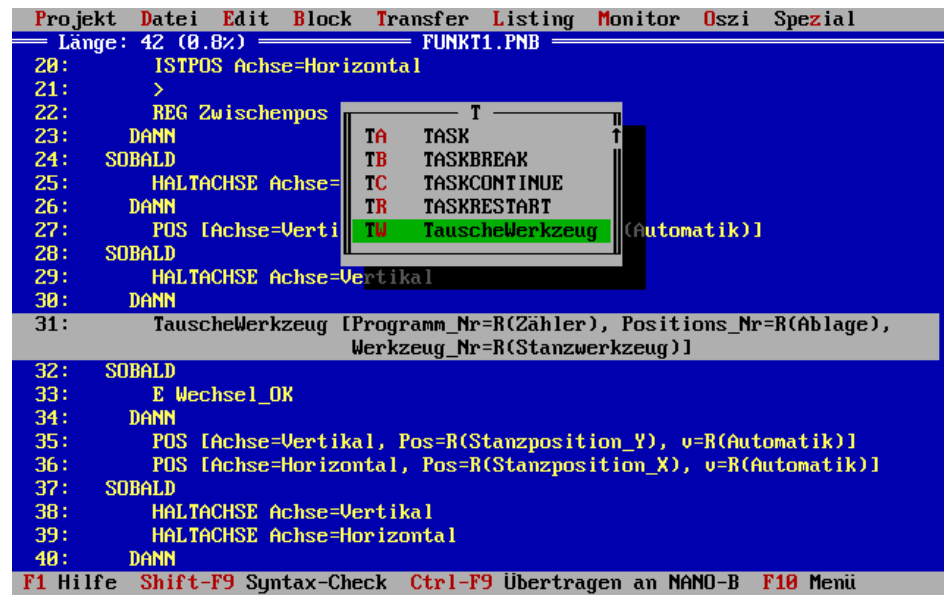


Figure 15: The call-up is made with the help of function (instruction) contractions the same way as in instruction call-ups

Example 1 (in an output condition):

The function begins with DEF_FUNCTION

```

0: DEF_FUNCTION [Exchange Tools, ET]
    Par: ProgramNo., PositionNo,
        ToolNo
1:     REG rToolNo
2:     =
3:     REG rProgramNo
4:     +
5:     REG rPositionNo
6:     THEN
7:     OUT oEject
8:     WHEN
9:     IN iEjectOK
10:    THEN
11:    OUT oFeedTool
12:    RETURN
13: END_DEF
14: TASK tInitialisation -----
15:    WHEN
16:    IN iFeedback
17:    THEN
18:    POS [Axis=axHorizontal,
        Pos=R(rPut_Down),v=R(rAutomatic)]
19:    WHEN
  
```

The function definition ends with END_DEF

The function is called up by Exchange-Tool

```

20:      ACTUAL_POS Axis=axHorizontal
21:      >
22:      REG rIntermediatePos
23:      THEN
24:  WHEN
25:      AXARR Axis=axHorizontal
26:      THEN
27:      POS [Axis=axVertical,Pos=R(rPutDown)
           v=R(rAutomatic)]
28:  WHEN
29:      AXARR Axis=axVertical
30:      THEN
31:      ExchangeTool
           [ProgramNo=R(rCounter),
           PositionNo=R(rPutDown),
           ToolNo=R(rPunchingTool)]
32:  WHEN
33:      IN iExchangeOK
34:      THEN
35:      POS [Axis=axVertical,
           Pos=R(rPunchingPositionY),
           v=R(rAutomatic)]
36:      POS [Axis=axHorizontal,
           Pos=R(rPunchingPositionX),
           v=R(rAutomatic)] 37:  WHEN
38:      AXARR Axis=axVertical
39:      AXARR Axis=axHorizontal
40:      THEN
41:      OUT oPunch1
42:      ...
End of program

```

Example 2 (in an input condition):

In this example enquiry is made, whether the starting condition `StartCondFulf` has been fulfilled. When the condition has been fulfilled, the task `TASK tPunchHole` is started.

```
0: DEF_FUNCTION [StartCondMet, SE]
    Par: rMinimumPos
1:   IF
2:     IN iDoorLocked
3:     IN iStartSignal
4:     FLAG fGlobalEnable
5:     REG rMinimumPos
6:     >
7:     1000
8:   THEN
9:     REG StartCondMet
10:    =
11:    1
12:  ELSE
13:    REG StartCondMet
14:    =
15:    0
16:  THEN
17:    RETURN
18: END_DEF
19: TASK tPunchHole -----
20:   WHEN
21:     StartCondMet
22:     [rMinimumPos=rActualPosition]
23:   THEN
24:     ...
End of program
```

3.6 Registers and Flags

The following instructions serve dealing with registers and flags. They will be explained in this chapter.

REGISTER_LOAD

COPY

specialfunction no.x

REGDEC

REGINC

REGZERO

REG_CLEAR

BIT_SET

BIT_CLEAR

FLAG

CLEAR_FLAGS

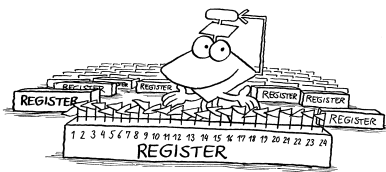
3.6.1 Basic Information on Registers

Register definition by numeric parameter or symbolic variable name

The registers are the numeric stores of the PROCESS-PLC. They can be used like variables. A value can be assigned to, and later be read from, them. A difference is made between integer registers, floating point registers and special registers. All registers are marked by a number or a symbolic variable name.

With all registers there is the possibility of indirect addressing. This means, that the number of a required register is in another register. On this subject, please see the REGISTER_LOAD instruction.

Integer Register:



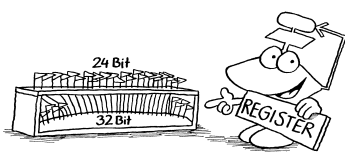
All these registers are 24 Bit wide registers, in which an integer number between **-8388608** and **8388607** is stored.

The following integer register ranges are supplied by the various PROCESS-PLC:

PROCESS-PLC	Integer Register Instruction
PASE-E Plus	0 to 8191 with additional memory expansion
	MEM128: 200000 to 232767
	MEM512: 200000 to 331071
	MEM1024: 200000 to 462143

DELTA	0 to 20479
NANO-C	0 to 1999 20000 to 27999
NANO-B	0 to 1999
NANO-A	0 to 199
MIKRO	100 to 1099

These integer registers can also be used as parameters for various instructions. Instead of a certain parameter, R100 is written, for example. The consequence for the instruction is, that the present memory content of register R 100 is used as a parameter.



Floating Point Register:

These registers are 32 Bit wide and can store real numbers, i.e. floating point numbers in a range between

$$-10^{15} \text{ to } + 10^{15}.$$

The smallest possible number has got an amount around $1,2 \cdot 10^{-15}$

Due to the 32 Bit wide display of numbers, a calculation preciseness of up to 7 decimal places will result.

The following floating point register ranges are supplied by various PROCESS-PLC:

PROCESS-PLC	Floating Point Register Range
PASE-E Plus	8960 to 9215
DELTA	62208 to 62463
NANO C	65024 to 65279
NANO B	-
NANO A	-
MIKRO	-

Special Registers:

The special registers, and thus the functions of the operating system are to be handled carefully

The special registers contain parameters, which are being used by the operating system. A big range of possibilities of influencing the functioning of the controller is offered; thus, it should only be used very carefully!

Slave Registers:

These registers are on intelligent modules resp. boards of various PROCESS-PLC. They serve the communication between the CPU and the processor of the module, respectively of the board. Thus, instructions and parameters are written into these registers by the controller program, and the status report is read. Access to the slave registers is made by the same instructions as for any other register. In some of these registers conditions or present values (e.g. the actual axis position) have been stored and have for this reason only been designed as **read-only** memories. Writing into these registers is not permitted!

PROCESS-PLC	Slave Register Range
PASE-E Plus	Slot 1: 11100 to 11799 Slot 2: 12100 to 12799 ... Slot 32: 42100 to 42799
DELTA	Slot 1: 21000 to 24999 Slot 2: 31000 to 34999 Slot 3: 41000 to 44999
NANO-B	CPU: 11100 to 11999 Module 2: 12200 to 12999 Module 3: 13300 to 13999 Module 4: 14400 to 14999
NANO-A	-
MIKRO	1100 to 1149

3.6.2 Instructions for Register Loading



The instruction

```
REGISTER_LOAD [ x with a ]
```

serves loading of number values (or contents of other registers) into a register.

Description:

In the instruction mentioned above, x stands for the number of the register number a is to be written into.

Indirect and Doubly Indirect Addressing

By pressing the **(SPACE)** key (one or two times), the indirect levels are selected

For the "x" and the "a" in the upper instruction, there cannot only stand a number, but also a register can be specified: By pressing the space key, an "R" can be placed in front of the register number.

If "Ry" is written instead of "x", value "a" is written into the register, the number of which is written in register y.

If "Rb" has been written instead of "a", the result will be, that not the value itself, but the content of the specified register is loaded into register x (or Ry).

If for "a" "RR" (SPACE key twice) and then a number (b) is input

```
REGISTER_LOAD [ x with RR(b) ]
```

the result will be the following: First the value of register b is read. This value now serves as a register number. In the register of this number a new value is written and finally stored in register x.

Example:

1) Load a number into a register

```
REGISTER_LOAD [rNewPosition with 1280]
```

Value 1280 is loaded into REG rNewPosition.

2) Copy a register onto another one

```
REGISTER_LOAD [rVoltage with R(rVoltage1)]
```

The value that is written in REG rVoltage1 is loaded into REG rVoltage. In other words, the content of REG rVoltage1 is copied into REG rVoltage.

3a) Load by doubly indirect addressing

```
REGISTER_LOAD [rVoltage with RR(rU_Pointer)]
```

The value, which is in the register of the number written in register REG rU_Pointer, is loaded into REG rVoltage.

3b) Example for Double Indirect Addressing Using Numbers

Register Occupation	Value
REG 64	211
REG 211	70035
REG 5000	4711
REG 4711	arbitrary

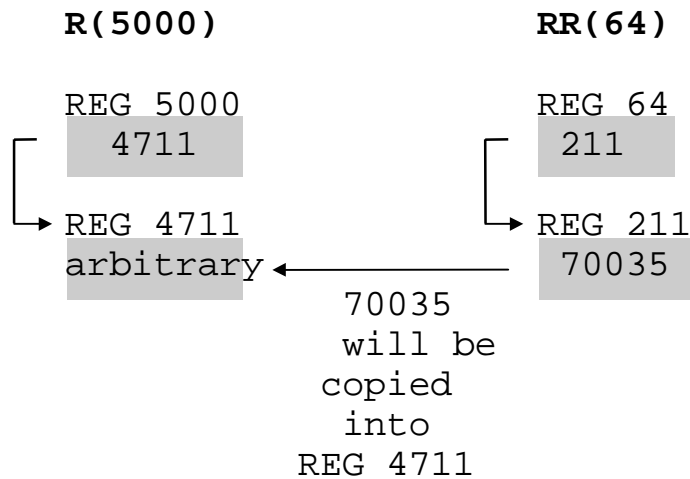
with the help of this occupation the following instruction is carried out:

```
REGISTER_LOAD [R(5000) with RR(64)]
```

The following register values will result:

- Register 64 = 211 (remains the same)
- Register 211 = 70035 (remains the same)
- Register 5000 = 4711 (remains the same)
- Register 4711 = R5000 = RR64 = R211 = 70035

Graph:





The instruction

`COPY [n=<Num of Reg> from <SourceReg> to <DestReg>]`

serves for copying whole register blocks. Only the number of registers has to be input: the number of the first register that has to be copied, and finally the number of the register, into which the first register is to be copied.

Indirect
parameter
input is
possible

All those three parameters can be input in **simple indirect mode**.

Example:

`Copy [n=5, from 100 to 200]`

Reg.Nr.	Content	Execution	Reg.No.	Content
100	77	—————>	200	77
101	3198	—————>	201	3198
102	791	—————>	202	791
103	86320	—————>	203	86320
104	13629	—————>	204	13629

The above presentation is to illustrate what is happening after the COPY instruction: Five registers are copied ($n=5$). The first register to be copied is REG 100 which is copied into REG 200. In the example arbitrary values were assumed for REG 100 to REG 104. It is important that after the copy instruction the same values appear in REG 200 to REG 204.

Example:

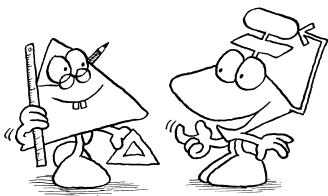
1) `Copy [n=100, from 0 to 1000]`

Registers REG 0 to REG 99 are copied onto registers REG 1000 to REG 1099.

2) `COPY [n=100, from 700 to 650]`

REG 700 to REG 799 are copied onto REG 650 to REG 749. Here, registers REG 700 to REG 749 are given new values, while the former ones get "lost". (Now they are stored in REG 650 to REG 699.)

**SPECIAL-
FUNCTION 1**
for axis
initialisation



The instruction

```
SPECIALFUNCTION [ #1, p1=a, p2=b]
```

also serves the copying of register contents.

There are two more special functions, which have been explained in *Chapter 3.11.4 Special Functions*. Special function 1 serves for copying and will thus be described below.

Special function 1 has been developed for the initialisation of axis cards, as a great number of registers have to be loaded there. The function has got two parameters: In the first parameter (p1) the number of the first register of a description block is defined, in (p2) the first register to be written into (b) is defined. Parameters p1 and p2 can also be indirectly addressed. In this description block all details are contained of how many and which registers are to be copied after the following standards:

Description Block

Reg.No	Content
a	Number of registers to be copied
a + 1	1 st offset register number
a + 2	Content of the 1 st registers
a + 3	2 nd offset register number
a + 4	Content of the 2 nd register
a + 5	etc.

**Remark:**

For each register to be written into, two registers of the description block are needed, for example register $a+1$ and register $a+2$. Those two numbers result in copying value $R(a+2)$ into the register of the number $b+R(a+1)$. This procedure can be illustrated best by an example:

Example:

```
SPECIALFUNCTION [ #1, 100, 1000 ]
```

The description block starts with register 100. As an example the values of register 100 and the following are to be:

Reg 100 = 4	Reg 103 = 11	Reg 106 = 912
Reg 101 = 10	Reg 104 = 199	Reg 107 = 19
Reg 102 = 4500	Reg 105 = 15	Reg 108 = 9999

After carrying out the special function the following registers contain the values

```
Reg 1010 = 4500
Reg 1011 = 199
Reg 1015 = 912
Reg 1019 = 9999
```

The instruction

```
REG_CLEAR [from Reg<first reg> to <last reg>]
```

serves clearing of register contents. Register blocks of any size can be set to zero. The registers, the first and the last register being included, are set to zero.



Example

```
REG_CLEAR [100 to 200]
```

All registers from 100 to 200 are set to zero.

3.6.3 Calculating with Registers

Calculating with registers is extensively described in the chapter on arithmetic expressions (*Chapter 3.4 Arithmetic*). Here, especially the instructions

REG <RegNr>

REGNULL <RegNr>

REGDEC <RegNr>

REGINC <RegNr>

are to be explained. In all of those four instructions it is possible to indirectly specify which is the only parameter to be defined. Thus, for example, for RegNo., Reg 100 can be written. This means that for the instruction the register is selected the number of which has been written into register 100.



The instruction

REG

By this instruction a register value can be directly accessed and treated like a variable. In an output instruction the register that has been written on the left of the equal sign a value is attributed. In an input condition the content of the register is read. The register accesses written at the right of the equal sign result in both cases in reading of the register.

Example:

```
1)   THEN
      REG 1
      =
      REG 105
      *
      25
```

In this example an assignment (output assignment introduced by **THEN**) is shown. Register **REG 105** is read and its content multiplied by 25. The result of this operation will be stored in register 1. The content of **REG 105** remains unchanged.

```
2)   IF
      REG 1
      =
      REG 105
      *
      25
      THEN
```

In this case the expression **REG 1 = REG 105 * 25** is not written in an output instruction, but it serves as an input condition. In this program part the value of register 1 will not be changed. It will only be compared with the product **REG 105 * 25**. (also see *Chapter 3.3 Boolean Expressions*)



A register is set to zero by the REGZERO instruction, or an enquiry is made, whether a register value is zero:

REGZERO <RegNr>

When this instruction is used as an input condition, (after IF or WHEN) it has got the following meaning, which is to be explained by the example below:

Example:

IF	REGZERO 49	IF	REG 49
	THEN		=
			0
			THEN

In both program parts the same function is carried out. At the right, the comparison is carried out as a general arithmetic comparison, while at the left the special instruction REGZERO is used.



The instructions
REGDEC **REGINC**

These two instructions serve for decrementing a register by 1, respectively to increment it by 1. These functions are often used in loops for incrementing or decrementing counters and pointers.

Examples:

1a)	THEN	1b)	THEN
	REGDEC 100		REG 100
			=
			REG 100
			-
			1

These two program parts have got the same function. In both of them the content of register 100 is decremented by 1.



2a)	THEN REGINC 88	2b)	THEN REG 88 = REG 88 + 1
-----	---------------------------------	-----	---

Here both program parts have also got exactly the same result. Register 88 is incremented by 1.

```

3)          REGISTER_LOAD [rCounter with 10]
LABEL 55
          ...
          REGDEC rCounter
IF
          REGZERO rCounter
THEN
ELSE
          GOTO 55
THEN

```

This way a loop can be realised that is carrying out a certain number of runs. In the loop the counting register is always decremented by 1, and then it is evaluated by comparison whether it is zero (REGZERO rCounter). If it is zero, no step is taken after the first THEN; the program goes on to the second THEN instead to be continued from there. If register 1 is not 0, though, a return is made to the beginning of the loop.

3.6.4 Register Bit Instructions



Using the instructions

BIT_SET

BIT_CLEAR

individual register bits can be queried, set or cleared.

In this case, the instruction

BIT_SET [Reg. <RegNo>, Bit <BitNo>]

means as an **output instruction** after THEN or ELSE):

The described bit is to be set; it is to be given value 1.

means as an **input condition** (after IF or WHEN):

Enquiry is made, whether the described bit has been set; this means, whether it has got value 1.



BIT_CLEAR [Reg. <RegNr>, Bit <BitNr>]

means as an **output instruction** (after THEN or ELSE):

The described bit is set to zero.

means as an **input condition** (after IF or ELSE):

Enquiry is made, whether the described bit is zero. If it is, the expression will be true; otherwise it will be false.

The **register number** can also be given here in indirect mode, but not the bit number.

Bit Numbering:

Integer Registers (24 Bit):

23	22	21	20	19	18	17	16	15	14	13	12	...
----	----	----	----	----	----	----	----	----	----	----	----	-----

...	11	10	9	8	7	6	5	4	3	2	1	0
-----	----	----	---	---	---	---	---	---	---	---	---	---

Bit 0 is the bit of lowest value.

Bit 23 is the bit of highest value.

Internal numbering is made by complements of two

Internal numbering is made by complements of two, for example:

+10	=	0000...1010
+1	=	0000.....01
0	=	0.....0
-1	=	1111.....11
-10	=	1111...0110

Bit 23 is 1 for negative numbers

Bit 23 ist bei negativen Zahlen 1.

The value can be calculated as follows: $V = 2^{\text{BitNo}}$

Examples:

```
1) THEN
    BIT_SET [Reg.12, Bit 3]
```

The fourth bit of register 12 is set, this means, it will have value 1 after this. If all the other bits of this register are zero, register 12 will then have value 8.


```

2) REGISTER_LOAD [1 with 0]
   BIT_SET [Reg.1, Bit 9]
   BIT_SET [Reg.1, Bit 8]
   BIT_SET [Reg.1, Bit 6]
   BIT_SET [Reg.1, Bit 3]
   BIT_SET [Reg.1, Bit 2]
   ...

```

In this program, certain bits of register 1 are set. The value resulting for register 1 can be calculated as follows (summing up of the "set values"):

$$2^9 + 2^8 + 2^6 + 2^3 + 2^2 = 512 + 256 + 64 + 8 + 4 = \underline{844}$$

3a)

```

WHEN
  BIT_SET [Reg.1, Bit12]
THEN
  ...

```

3b)

```

WHEN
  NOT
  BIT_CLEAR [Reg.1, Bit12]
THEN
  ...

```

These programs have got exactly the same result. It is waited, until bit 12 of register 1 has been set (not cleared).

3.6.5 Flags and Flag Instructions

Flags have got value 1 or 0

Basically, flags are memories, which are only one Bit wide, though. This means they can either have value 1 or 0. For this reason one can say a flag has been set (= 1) or cleared (= 0). Just as the registers, the flags are also marked by numbers. The numbering of the flags can be looked up in the manual on the respective controller. Numbers 2048 to 2303 have been reserved for special flags, which are needed by the system. The special flags that can be helpful to the user have been described in *Chapter 4.1.2 Flags*.

Overlapping flag - register ranges

Further, there is a flag range overlapped by a register range. For numbering ,please see the manual on the respective controller. For further details see *Chapter 4.1 Basics on Registers and Flags*

The following flag instructions are available:

FLAG

CLEAR_FLAGS



The instruction

```
FLAG <FlagNo>
```

has got the following meaning:

as an **output instruction** (after **THEN** or **ELSE**):

The flag is set, this means it is given value 1.

The flag can be cleared by inputting "-" (negative sign) in front of the flag number.

as an **input condition** (after **WHEN** or **IF**):

Enquiry is made, whether the flag has been set. The result of this enquiry corresponds to the flag value and has got the meaning true (= 1) or false (= 0).

Here, a negative sign can also be input in front of the flag number to get access to the inversed flag number. This means, enquiry is made, whether the flag has been cleared. (Thus, the result is exactly the inverse value of the flag.)

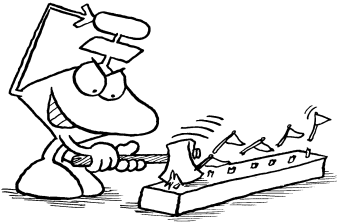
Example:

```
1)  THEN  
    -FLAG 2
```

By this instruction, flag 2 is cleared. After this, the flag value will be 0.

```
2)  WHEN  
    -FLAG 61  
    THEN
```

Flag 61 is waited for to be given value 0. This flag can, for example, be changed by another task; this means, it can be deleted by this task. If the flag is 0 already, when the program reaches this instruction, the rest of the program will be continued immediately.



The instruction

CLEAR_FLAGS

serves for clearing whole blocks of flags. This will be illustrated by the following example:

Example:

```
CLEAR_FLAGS [1 to 300]
```

All flags from 1 to 300 are cleared; this means all flags from 1 to 300 will afterwards have got value 0.

3.7 Inputs and Outputs

Inputs and outputs serve for inputs and outputs of binary signals.

3.7.1 Inputs



The inputs can be directly accessed by the program as a binary signal, this is, as 0 or 1. The inputs can, for example, be connected to a switch, which can be queried in the program by the input instruction.

For numbering of the inputs, please see the manual on the respective controller.

Inputs can be queried, but not set

The inputs have only got a function in input conditions. This means, they cannot be set by the software or be influenced in any other way. Only enquiries are possible..

This can be easily achieved by the input instruction:

IN 101

Enquiry is made by this instruction, whether input 101 has been set. If the input number is preposed by a negative sign, access can be made to the inverted signal, just as it is the case with flags.

Example:

```
WHEN
  -IN 108
THEN
```

In this program part, input 108 is waited for not being set any more. As soon as the input is not active any more, the program is continued.

**Several inputs
are combined
in registers**

Via special registers, several inputs can be read simultaneously. The register numbers can be taken from the respective controller manual. Besides that, they will be displayed as examples by *Chapter 4.1* .

3.7.2 Outputs



Outputs are digital switches to control valves, fuses, LED's or similar devices.

The numbering of the outputs can be taken from the respective controller manual.

The instruction for switching or querying an output is:

```
OUT 101
```

This instruction will have the following effect:

as an **output instruction** (after THEN or ELSE):

Output 101 will be set (activated)

as an **input condition** (after IF or WHEN):

Enquiry about the output: Has the output been set?
(This is an internal logical query. This means, that, for example, short circuits of the voltage output cannot be recognised that way.)

By input of a negative sign in front of the output number access can also be made to the inverse value ; this means, enquiries can be made, whether an output has not been set, or the output will be deactivated as an output instruction.

Here, the output number can also be specified indirectly.

Examples:

```
1)  IF
      IN 101
      THEN
      OUT 201
      ELSE
      -OUT 201
      THEN
      ...
```

Here, input 101 is queried first. If this input has been set, output 201 is to be set as well. If input 101 has not been set, output 201 is to be reset.



Remark:

Setting an output that is already active, as well as resetting an output that is not active will be to no effect.

```
2)  WHEN
      -OUT 202
      THEN
```

When output 202 is not active any more, the program is to be continued.

For outputs, there are also registers, by which simultaneous access to 8, 16 or 24 outputs can be made. For numbering see the respective controller manual.

3) Register 2540 (NANO-B) goes together with outputs OUT 101 to OUT 108.

Examples

1)

Register
number
NANO-B

```
REG 2540
=
b000000000000000011111111 (16 zeros, 8 ones)
```

2)

```
REGISTER_LOAD [2540 with 255]
```

The outputs (1 to 8) are set by these instructions. (After this, each one of the 8 LED's will be active). By the assignment of registers, the binary value can directly be input or - if this is required - the problem can be solved by one direct instruction like the REGISTER_LOAD instruction top right. For this purpose, the binary value must first be transferred into a decimal value. (See also *Chapter 3.11.4 Special Function*)

3.8 Display Instructions and User Input

In this chapter, the instructions for user interfaces will be explained. The devices have been described in a separate manual.

These are the instructions:

DISPLAY_TEXT

DISPLAY_REG

USER_INPUT

3.8.1 Display of Texts



The instruction

```
DISPLAY_TEXT[#<DeviceNo>,cp=<CursorPos> „<Text>„]
```

serves for text output on user interfaces or on a printer.

Meaning of the parameters

Device Number

For this parameter, 0 to 9 can be input.

User Interface **#0 to #4**

A user interface will be controlled

Printer **#8**

By this device number, the control system is caused to edit the text onto a printer.

Free
programm-
able
interface

#9

Editing is made via the free programmable PRIM interface.

Separate display on up to 4 simultaneously connected user interfaces is possible (a description can be ordered).

Cursor Position

In this parameter the cursor position is given, where the first character of the text is to stand. In this case, values from 0 to 127 are possible. The respective cursor positions can be taken from the manual on user interfaces.

Example on LCD9:

First line of the display: Cursor positions from **1 to 24**

Second line of the display: Cursor positions from **25 to 48**

Cursor position **0** has got a special meaning: If cursor position 0 is set, the text will be attached to the text edited last. The cursor will stand at exactly the same position, where it stopped after completing the former display instruction. (The function of cursor position 0 can be changed by a special register. Referring to this,

please see the description of the special registers in the controller manual.)

Text

_ and \$
are test marks

Here the text can be input which is to be displayed. The two characters "_" and "\$" serve as test marks:

The display is
cleared by _

"_" This character causes the display to be cleared first and the input text to be displayed (independently from the input parameter) then beginning with cursor position 1. This character only makes sense at the beginning of a text, as otherwise the first part of the text would be displayed first, yet would immediately be cleared again. This character has got the meaning **DELSCR** (Delete Screen). If this character is to be input, the character code for DELSCR can be changed in a special register of the controller.

For printing, this character has got the meaning **FORM FEED**.

By \$, the end
of the line,
starting from
the cursor
position
will be
cleared

„\$“ This character causes the rest of the line, starting from the present cursor position, to be cleared. It is also called **DELEOL** (delete end of line) and can also be replaced by another character (see description of special registers in the user's manual).

For printing, this character has got the meaning **LINE FEED**.

Examples:

1)

```
DISPLAY_TEXT [#0, cp=0, "_Actual_Pos:"]
```

First, the whole LC display will be cleared by this instruction, then "ActualPosition:" will be written on the upper line of the display (cursor position = 1). Instead of the cursor position, any other number could be written, as this will not be considered any more after input of the Delete Screen mark (DELSCR). The display will then look as follows:

```
ActualPos:
```

2)

```
DISPLAY_TEXT [#0, cp=25, "NominalPos:$"]
```

The text "NominalPos" is written, starting from the defined cursor position 25, which is the beginning of the second display line; then, the rest of the line is cleared.

3)

```
DISPLAY_TEXT [#0, cp=0, "FEHLER"]
```

Beginning at the present cursor position, the text "ERROR" is written. This means the text is simply attached to the former text.

4)

```
DISPLAY_TEXT [#8, cp=1, "This will be sent to the  
printer"]
```

The result will be, that the text "This will be sent to the printer" will be printed on the printer, starting from the beginning of the line. Details on printer output have been described in the controller manual. For editing on PRIM, the cursor position will be ignored.

3.8.2 Display of Register Contents



The instruction

```
DISPLAY_REG[#<DeviceNo>, cp=<CursorPos>Reg=<RegNo>]
```

serves for output of a register value onto user interfaces, modules, or a printer.

The parameters **device number** and **cursor position** have got exactly the same function as they have got in the `DIPLAY_TEXT` instruction (see above). Further, a **register number** must be input here. This is the register number the value of which is to be displayed. It can also be input by indirect addressing.

For register display there are two parameters to be changed, which are stored in the special registers "Field width for integer display" and "Alignment left/right". Following, the standard settings (after reset) will be described:

Setting after
reset

**Special Register "Field Width for Integer Display"
= 1**

8 places are used for register display; negative signs are displayed in the very beginning of the line and numbers flush right in the other seven places.

Setting after
reset

Special Register "Direction, Left/Right" = 0

Flush right display

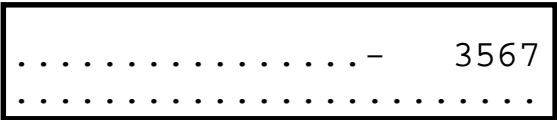
By PROCESS-PLC (not NANO-A) floating point numbers can also be displayed on user interfaces; special register "Field Width for Floating Point Display" (value 1..14).

Examples:

1)

```
DISPLAY_REG [#0, cp=17, Reg=100]
```

Register 100 is displayed by this instruction. If the special registers "Field Width for Integer Display" and "Arrangement; Right/Left" have not been changed since reset, the register will appear at the end of the first display line, as it is displayed in the following: (assumptions: Before the instruction the display was empty, while register 100 = -3567).



The dots are to mark the places, which, after giving the instructions, still have got the "old" contents.

2)

```
DISPLAY_TEXT [#0, cp=25, „ActualPos :$“]  
DISPLAY_REG [#0, cp=41, Reg=11109]
```

It is shown here how the two outputs/instructions can be meaningfully combined. First, the text "Actual Position" is written into the second line (left), and the rest of the second line is deleted (Dollar sign „\$“). Giving the second instruction, register 11109 is displayed bottom right in the display. The actual position of the stepper motor axis on the basic module NANO-B is stored by this register (assumptions: The actual position of axis 1 is to have value 5400.)

```
.....  
Actual position:          5400
```

The dots are to show the places, which will still have the "old" content after giving the instructions.

3) The following exemplary program is to demonstrate how it can become possible to lead a protocol of values directly to the printer.


```

TASK 5
REGISTER_LOAD [ rField width with 2]
DISPLAY_TEXT [#3, cp=1 "$"]
DISPLAY_TEXT [#3, cp=1 "nominal_pos"]
DISPLAY_TEXT [#3, cp=21 "actual_pos"]
DISPLAY_TEXT [#3, cp=41 "speed$"]
LABEL 100
WHEN
  FLAG 1
THEN
  DISPLAY_REG [#3, cp=3, Reg=rSollPos]
  DISPLAY_REG [#3, cp=23, Reg=rIstPos]
  REG 1
  =
  30
  *
  REG rActualPos
  /
  REG NumberEncoderLines
  DISPLAY_REG [#3, cp=43, Reg=1]
  -FLAG 1
  GOTO 100

```



This program has been programmed in a parallel branch, which is very practical, as this way printing of a protocol line can be triggered by setting flag 1 from any other parallel branch. At the beginning of this task (parallel branch) the title line is printed. This is done before the actual loop, so that in this case three columns are printed. In the first column, there is the nominal position, in the second one, there is the actual position, while in the third column there is the present speed in rev./min (if 1000 increments per revolution are evaluated).

**NANO-A: no
printer
connection**

The protocol printout could look as follows:

Nominal Pos.	Actual Pos.	Speed
15000	8433	2450
- 4800	1206	- 1207
250000	250000	0

3.8.3 Reading of Register Values by the Program

The instruction

```
USER_INPUT[#<DeviceNo>,cp=<Cursor_pos>,Reg=<RegNo>]
```

serves the writing of register values, which can be input via the keyboard of the display and keyboard module.

The same as for DISPLAY_TEXT instruction also applies to the two parameters **device number and cursor position**, yet with the following changes: In case of the device number, there is of course no printer that can be accessed, thus device number 8 is to be avoided in this case. If cursor position 0 is input, the value out of special register "Absolute Cursor Position for USER_INPUT" is selected as cursor position at the user input. If this value is also 0, though (which is the value the register has got after reset), the register will be written into at the present cursor position.

Indirect
addressing of
the destination
register is
possible

The **register number** is the number of the register the input value is to be assigned to. Here, simple indirect register addressing is possible as well.

It is important to know, that, for USER_INPUT there are normally 8 characters available. This value, which has been stored in the special register "Field width USER_INPUT" can also be changed. (See *Chapter 3.8.4 Special Registers for User Input*).

Example:

**Combination of
DISPLAY_TEXT
and
USER_INPUT**

To achieve effective operator guidance, USER_INPUT is often combined with the DISPLAY_TEXT instruction.

```
DISPLAY_TEXT [#0, cp=1, "_New Position ?"]  
USER_INPUT [#0, cp=17, Reg=100]
```

After giving those two instructions, the text "New position ?" will appear, and then the input of a number is waited for. This number, which is stored in register 100, serves as a new nominal position for a positioning run.

3.8.4 Special Registers for User Input

**Note:**

The special register numbers can be taken from the respective controller manuals, where a general survey of the special registers is given.

Special Register "Field Width for Floating Point Display"

Field width of the `DISPLAY_REG` instruction for floating point numbers. Value range 1..14, value after reset: 8.

Special Register "Field Width for Integer Display"

In this register the number of characters has been defined, which are to be displayed after a `DISPLAY_REG` instruction. After reset the value is set to 1, which corresponds to a display of 7 characters.

Special Register "Field Width for Flush Left Display"

In this register information is contained, whether a register to be displayed is to be written on the user interface flush left or right. The value after reset is 0 here.

In the following tables the various possibilities of combining the two registers and its effect on the display format are shown. The "*" (asterisk) will in the following stand for the output of a space. The "+" stands for a place holder of a positive sign; actually, a space will be displayed, though.

a) Special register "Flush Left Number Display" = 0;
default after reset

RegValue	1234	-345	7654321	-1234567
0	***+1234	***-345	+7654321	-1234567
1	+***1234	-***345	+7654321	-1234567
2	+**1234	-**345	+654321	-234567
3	+*1234	-*345	+54321	-34567
4	+1234	-345	+4321	-4567
5	+234	-345	+321	-567
6	+34	-45	+21	-67
7	+4	-5	+1	-7
8	+	-	+	-

b) Special register "Flush Left Number Display" = 1

RegValue	1234	-345	7654321	-1234567
0	+1234	-345	+7654321	-1234567
1	+	-	+	-
2	+1	-3	+7	-1
3	+12	-34	+76	-12
4	+123	-345	+765	-123
5	+1234	-345*	+7654	-1234
6	+1234*	-345**	+76543	-12345
7	+1234**	-345***	+765432	-123456
8	+1234***	-345****	+7654321	-1234567

c) For special register "Flush Left Number Display" = 2
see a), yet, no sign will be displayed.

Special Register "Field Width `USER_INPUT`"

The value of this register is the field width, which is presented to the user by the user interface, when the `USER_INPUT` instruction is carried out. The value of this register after reset is 8, which means that 8 characters are available to the operator for input. The first space for a character is reserved to the sign and will be occupied by a sign only.

3.8.4.1 Control Character for the DISPLAY_TEXT Instruction

Special Register "Delete Characters up to the End of the Line"

Default after reset: \$

In this register, the ASCII code of the **DELEOL** (DELEte End Of Line) character is contained. After reset it will have value 36, which is the ASCII code of „\$“ (Dollar sign). If, for example, a Dollar sign is to be used in a text display, thus, by this register, the function of "\$" can be transferred onto another character by inputting of another number.

Special Register "Clear Character Display"

Default after reset: _

In this register, the ASCII code of the **DELSCR** (DELEte SCReen) character is contained. Deactivation of the LC display is caused by this character. After reset, the register will have got value 95, which is the ASCII code of "_" (underline). This character can also be replaced by any other.

Register numbering
DELTA

Example:

```
REGISTER_LOAD [ 61462 with 38 ]  
DISPLAY_TEXT [ #0, cp=0, "&"]
```

The display is cleared by those two instructions. First, the DELSCR character will be changed; then the DISPLAY_TEXT will immediately be activated.

3.8.4.2 Controller Register for Cursor Position on the LC Display

Special Register "Absolute Cursor Position DT, DR"

The value of this register represents the cursor position after the `DISPLAY_TEXT` or the `DISPLAY_REG` instruction, in case cursor position 0 has been defined in the instruction. If the value of this register is 0 as well, which is the case after reset, the text to be displayed is simply attached to the last display. This means, the present cursor position is not being changed.

Special Register "Absolute Cursor Position U"

This register has got exactly the same function as the one mentioned above, but referring to the `USER_INPUT` instruction. The cursor position is defined by this register, where the input is to be made, if, at a user input cursor position 0 has been defined. This value will also be 0 after reset, which will result in annexing to the latest input or output.

Register
numbering
DELTA

Example:

```
REGISTER_LOAD [ 61649 with 25]
USER_INPUT [ #0, cp=0, Reg=100]
```

The register for indirect definition of the cursor position for user input is first loaded with value 25. At the beginning of the second display line a number, which is then assigned to register 100 can be input with the help of the following instruction.

3.8.4.3 LCD Display Time

The display of the user interfaces functions on two display levels. Normally, level one is displayed, which is also accessed by the user program; this means, here the `DISPLAY_TEXT-`, `DISPLAY_REG-` and `USER_INPUT` instructions are displayed.

By pressing the 'R' and 'F' keys, registers, flags, inputs and outputs can be queried and changed. This is carried out on level two (called monitor screen).

The time for switching back onto level one can be defined by the

Special Register "Display Time for Monitor Functions"

In this register the time of how long a register, a flag, an input or an output queried via keyboard is to remain on the display is given in seconds. This is the time before switching back the display of a user program.

After reset this register has got value 3, that is, three seconds.

3.8.4.4 Input Enable for Flag and Register Changes via User Interface Keyboard

A number of special registers serve for definition of register ranges, which can then be changed via keyboard. In this case, every two registers following one another make up a pair, by which a block is defined. The first three pairs refer to registers, the fourth pair to flags.

Special Register "First Changeable Register - Range 1" "Last Changeable Register - Range 1"

The lowest register that can be changed is defined in the special register "First Changeable Register - Range 1" (value after reset = 0), while the upper register is defined in the special register "Last Changeable Register - Range 1" (value after reset = 59999) the upper one (first register - input enable range).

Special Register "First Changeable Register - Range 2" "Last Changeable Register - Range 2"

The same applies to the same register range (values after reset: both registers are 0)

Special Register "First Changeable Register - Range 3" "Last Changeable Register - Range 3"

The same applies to the third register range (values after reset: both registers are 0)

Special Register "First Changeable Flag" "Last Changeable Flag"

In special register "First Changeable Flag" (value after reset = 0) the lowest flag that can be changed, in special register "Last Changeable Flag" (value after reset = 59999) the upper flag that can be changed is defined.



Note:

The values after reset have been chosen in such a way, that all registers and flags can be changed via keyboard. In most cases it is advisable to protect blocks of registers and flags this way, which contain important values, in order to avoid loss of important values due to a typing error on the keyboard.

Input enable for flags, inputs, outputs and registers can also be generally prohibited. (For this, see "Special Flags" in the controller manual.)

Register
numbering
DELTA

Example:

```
REGISTER_LOAD [ 61697 with 49]
REGISTER_LOAD [ 61698 with 100]
REGISTER_LOAD [ 61699 with 199]
REGISTER_LOAD [ 61703 with 300]
```

Assuming that the other registers have not been changed after reset, these attributions result in the following status:

```
Register  0 to 49:  input permitted
Register 100 to 199: input permitted
Flag      1 to 300: input permitted
```

All other registers and flags cannot be changed via keyboard any more. Thus, all special registers, as well as the axis module registers, are protected.

3.8.4.5 Restriction of the Monitor Functions

Special Register "Restriction of the Monitor Functions"

Flags 2096 (Bit 0) to 2103 (Bit 7) are overlapped by this register.

0 = Function blocked, 1 = Function available

Bit0 = 0	R, I/O keys without monitor function (flag will be set, though)
Bit0 = 1	R, I/O key with monitor function
Bit1 = 0	R, I/O key without monitor function flag input
Bit1 = 1	R, I/O key with function flag input
Bit2 = 0	R, I/O key without function, input of output number
Bit2 = 1	R, I/O key with function, input of output number
Bit3 = 0	R, I/O key without function, input of input number
Bit3 = 1	R, I/O key with function, input of input number

Bit4 = 0	= register contents cannot be changed by the key
Bit4 = 1	= register contents can be changed by the key
Bit5 = 0	= flags cannot be changed by the key
Bit5 = 1	= flags can be changed by the key
Bit6 = 0	= outputs cannot be changed by the key
Bit6 = 1	= outputs can be changed by the key
Bit7 = 0	= no access to inputs by the key
Bit7 = 1	= access to inputs by the key

3.8.4.6 Time for User Input

Setting after
reset:
no time limits

Special Register "Maximum Time for `USER_INPUT`"

The operating system is informed by the register of how much **time** there is (in seconds) for the **operator** to carry out a **`USER_INPUT`**.

After reset, this register will have value 0, which has got the following meaning: without any condition of time user input and its confirmation by ENTER is waited for. The number will be input and the processing of the program continued then. When the register is set to 10, termination of the `USER_INPUT` instruction after 10 seconds will be caused.

Flag 2053

This flag must be seen in the context of "Maximum time for `USER_INPUT`". After `USER_INPUT`, enquiries can be made via this flag, whether the input has been finished properly by the operator, or whether the user input has been terminated by a 'timeout'.

Flag 2053 = 1 means termination by timeout.

3.9 Instructions for Axis Controlling

In this chapter all available instructions for axis control will be described. The instructions are

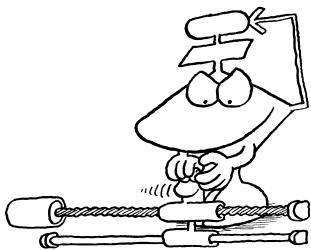
POS

ACTUAL_POS

AXARR

Using these makro instructions for axis control, program development will be made easier a great deal; the program will be clearer and easier to read.

3.9.1 Positioning



For axis positioning, that is, running the axis to a certain location, the following instruction is used:

```
POS [Axis<AxisNo>, Pos<ActualPos>, v<NomSpeed>]
```

Three parameters must be input, which serve for informing the axis board, where, and by which final speed, the axis is to be driven. Everything further will be controlled by the axis board on its own. In order to control the correct axis, the axis number must be input first.

The meaning of the parameters (the parameters can be indirectly addressed):

All parameters
can be
indirectly
addressed

Axis Number

Axis numbering can be taken from the respective manual.

Nominal Position

As an alternative
to the `POS`-
instruction,
position and
speed registers
can be written
into directly by
the
`REGISTER_LOAD`
instruction

Here, any number of an integer register can be input (this means, from -8388608 to 8388607). It depends on the whole process, whether all those values are useful. Normally, the range of numbers that is really useful for an application is smaller. (Please also see the descriptions of the respective servo controllers.)

In any case, the nominal position is given by this number; this is the position the axis is to drive to. The position input in the `POS` instruction corresponds to directly writing into the nominal position register.

Nominal Speed

In the number that has been input the maximum speed for this positioning is defined.

Both for nominal position and for nominal speed there can be doubly indirect register definition, for example `RR50`. For explanations on this doubly indirect register access see the `REGISTER_LOAD` instruction, *Chapter 3.6.2*.

This way, the meaning of the `POS` instructions can simply be the following:

For servo axes
sinus-shaped
start and stop
ramps

"Axis x : go to **position Pos** with **speed v**"

During acceleration the speed will slowly be increased (sine square shaped) and during deceleration when heading the destination position it will also be slowly (also in sine square shape) be decreased.

How fast the meaning of "slowly" is meant to be, that is, how high the steepness of the sine square function is to be, can be set using further parameters (start and stop ramps), which are directly loaded into the registers provided for this purpose.

Register
numbers by
the example
of NANO-B

In register 1xy05 there is the **start ramp**.

In register 1xy06 there is the **stop ramp**.



Remark:

Besides start and stop ramp, further parameter registers can be initialised on the axis boards in most cases. For this purpose the following instructions are preferred:

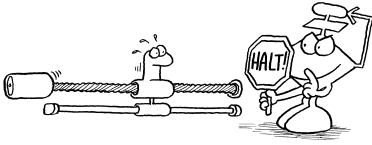
```
REGISTER_LOAD  
SPECIALFUNCTION [ #1, . . . ]  
COPY
```

These instructions are extensively explained in *Chapter 3.6.2*.

For positioning the

AXARR Axis<AxisNo>

instruction is also necessary. This instruction can be used as an input condition and an output instruction. The axis number can also be defined in simple indirect mode.



Deceleration with the help of the start ramp is caused by the AXARR instruction

As an **output instruction** (after THEN or ELSE) this means:

The axis is to stop immediately (i.e. without ramp) and the position is to be controlled at the present actual position.

As an **input condition** (after IF or WHEN) this means:

Has the axis reached the destination position (respectively the destination window)? If so, a 1 as a Boolean value (true), if not, a 0 (false) will be returned by the instruction.

If a negative sign is input in front of the negative number, the meaning of the instruction will be converted into its opposite.

In an **output instruction** this means, that an axis that has been stopped will continue, if its nominal position has not been reached yet.

In an **input condition** an enquiry can be made, whether an axis is still running.

**Remark:**

For positioning instruction the following should be considered: As soon as the instruction has been written and passed on to the axis board, the processor on the CPU will have fulfilled its task and thus go on to the next instruction. Axis positioning itself will independently be carried out by the micro processor on the axis controller module.

For this purpose enquiry will normally be made with the help of the `AXARR` instruction, whether the axis has already arrived in the destination window, before the next positioning is started. It is also definitely permitted, though, to start further positioning runs during one positioning run is already carried out. In this case, the axis will drive onto the nominal position that has been transferred last without stopping in between.

The destination window is an area that can be defined around the destination position. It can be set by register 7.

**Initialising
of the axis
controllers**

After the power supply of the controller has been switched on, all registers of the axis controllers will be loaded with the values they had at the beginning (reset values). In various SV modules, a relay will be switched off at this reset, which is going to transfer the analogue speed - nominal value output to the outside. By loading the number 1 into the instruction register (1) of the required module, this relay can be switched on:

**Referencing
of the axis**

Positioning is only possible, after a reference position has been loaded. For the beginning, this reference position can simply be defined. This is possible by writing number 3 into the instruction register:

```
REGISTER_LOAD [ rCommand with 3 ]
```

Normally, a special, exactly defined reference point is searched by reference run and then loaded into the register. In the instruction shown above, the present position is loaded as a reference point.

Example:

```
1) REGISTER_LOAD [rCommand with 3]
   REGISTER_LOAD [rCommand with 1]
   POS [axis=21, Pos=10000, v=500]
```

First the relay is switched on, then the reference point is set, and finally positioning is carried out:

"Axis 21 on the first axis module: Go to position 10000 with the speed 500‰!"

Axis 21 is to go to position 10000 with speed 500. This means that just the nominal position (10000) and the nominal speed (500) are loaded into the respective registers on the axis board.

The nominal speed is set on the output of the axis board as follows:

The speed rises up to its final value of 500 in a sine square shaped ramp (stepper motor: linear). When the axis is recognised to be near the nominal position, the speed is being decreased again, until the nominal position has been reached. There, the calculated speed will be 0.

Please find detailed information on the servo controller in the respective controller manual

(For a more detailed description, also on start and stop ramp, please see the servo controller and stepper motor chapters of the respective controller manual.)

```

2)      REGISTER_LOAD [rStartRamp with 50]
        REGISTER_LOAD [rStopRamp with 10]
        REGISTER_LOAD [rCommand with 3]
        REGISTER_LOAD [rCommand with 1]
        POS [axis21, Pos-40000, v1000]
    WHEN
        AXARR axis21
    THEN
        POS [axis21, Pos -30000, v200]

```

First, the ramps are defined (start/stop), then the reference is set, and finally the relay is switched on.

First, position -40000 is driven to with a speed of 1000%. After the position has been reached, the instruction is given to go to position -30000 with a speed 200%.

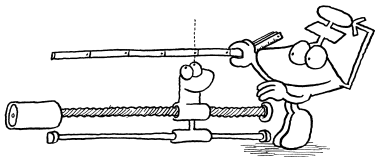
```

3)      REGISTER_LOAD [rCommand with 3]
        REGISTER_LOAD [rCommand with 1]
        POS [axis=21, Pos100000, v=300]
        DELAY 20
        AXARR 21

```

Here, the reference is set first, before the relay is switched on. Then, axis 21 (axis on the first axis module at slot 2) is instructed to go to position 100000 by a speed of 300%. After a delay of 2 seconds (definition by multiples of 100 ms), the axis run is interrupted by the AXARR instruction.

3.9.2 Enquiries on the Present Condition



To enquire about the present position, the

ACTUAL_POS

instruction is needed, which allows elegant access to the present axis position.

This position is in a register of the axis module (register 9) and could as well be found out by querying this register. The axis number can also be defined in indirect mode.

Example:

```
REGISTER_LOAD [rCommand with 3]
REGISTER_LOAD [rCommand with 1]
POS [axis=21, Pos10000, v=500]
WHEN
  ACTUAL_POS axis21
  >
  8000
THEN
  OUT 201
```

First, the reference is set, then the relay is switched on, and finally the instruction is given to go to position 10000 with a speed of 500%.

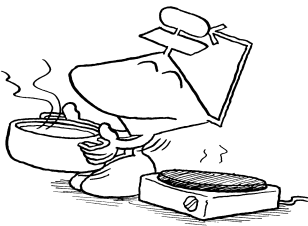
When the actual position is greater than 8000, output OU 201 is set.

3.10 Task Instructions

The instructions described in this chapter serve for mutual task control.

TASKBREAK	this way a task can be interrupted
TASKCONTINUE	this way a task can be continued
TASKRESTART	this way a task can be restarted

3.10.1 Taskbreak



By the instruction

TASKBREAK #<TaskNo>

processing of the defined parallel branch (task) is interrupted.

The parameter to be defined together with this instruction is the number of the parallel branch to be interrupted, that is, a number from 0 to 31.



Note:

Please mind here, that the controlling/positioning of intelligent slave modules will **not** be interrupted! If this should still be required, positioning must be explicitly terminated/ controlling must be interrupted.

With the help of this instruction, an automatic run, for example, can be interrupted at any position. Then, a manual mode or EMERGENCY STOP program can be processed.

3.10.2 Taskcontinue



The instruction

TASKCONTINUE #<TaskNo>

causes an interrupted parallel branch to continue processing.

3.10.3 Taskrestart



By the instruction

TASKRESTART #<TaskNo>

processing of the defined parallel branch is started afresh, that is, from the beginning of the task.

3.10.4 Examples of the Task Instructions

```

TASK 0 -----
    ...
    ...           ;e.g. manual mode
                ;program
    ...
TASK 1 -----
    ...
    ...           ;e.g. automatic mode
                ;program
    ...
TASK 2 -----
    ...           ;reference drive program
    ...
TASK 3 -----
    ...
    ...           ;e.g. further programs
    ...
TASK 4 -----
    WHEN
        IN -101 ;Emergency stop switch is
                ;pressed
    THEN
        TASKBREAK #0
        TASKBREAK #1
        TASKBREAK #2
        TASKBREAK #3
        AXARR axis21
        AXARR axis31
        AXARR axis41
    WHEN
        OUT 101      ;Emergency stop switch
                    ;deactivated
    THEN
        AXARR axis-21
        AXARR axis-31
        AXARR axis-41
        TASKCONTINUE #0
        TASKCONTINUE #1
        TASKCONTINUE #2
        TASKCONTINUE #3
        GOTO 4

```

For further examples and general information on multitasking see *Chapter 2.1*

3.11 Various Instructions

In this chapter, the following instructions will be described:

START-TIMER

TIMER-END?

NOP

;

SPECIALFUNCTION

LIMITS

Word Processing WAND, WOR, WXOR

3.11.1 Time Instructions

3.11.1.1 The Instructions START-TIMER and TIMER-END?

The instructions have got the following syntax:

```
START-TIMER [register no., value (time)]
```

```
TIMER-END? [register no.]
```

Those two instructions are written into together here, as they belong to the same function, that is, they depend on each other.

The **START-TIMER** respectively **TIMER-END?** instructions can be parameterised in indirect mode



The parameter of the **START-TIMER** instruction can be defined as a number or as a register number using indirect mode

With the help of **START-TIMER** and the **TIMER-END?** instructions, time can be monitored. In the **START-TIMER** instruction the required time, as well as the register the value is to be stored in, is contained, and the monitoring time is started in the running program by this instruction. The **TIMER-END?** instruction serves querying, that is, it is defined, whether the time set by the **START-TIMER** instruction has expired. Unlike after the **DELAY** instruction, the program will go on running for the defined time, even if it is in the same task. This function can, e.g. be used to limit the duration of processes, as, for example, warming up an item. There is no direct connection between the content of a defined register and a defined time. Thus, it is not easy to check how much time has already expired, that is, how much still remains!

All user registers can also serve as time monitoring registers.



Between the **START-TIMER** and the corresponding **TIMER-END?** instruction, no assignment must be made to the selected register, as otherwise the the **TIMER-END?** instruction will not render a result that is useful!

Internal Processing of **START-TIMER**, **TIMER_END?**

In case of the **START-TIMER** instruction the time given in the instruction is added to the content of a time base register that can be selected, while the sum is stored in the register defined in the instruction. The addition is carried out as a 22 Bit operation without sign. This means, that the maximum monitoring time can be 4 million time increments.

In the `TIMER-END?` instruction the stored value is compared with the present content of the time base register. If the time base register is still smaller than the stored value, the `TIMER-END?` instruction will have the result "false" (0). If the time base register is equal with, or greater than the stored value, bit 23 of the defined register is set (negative) and the result is "true" (1; time has expired).

This means that after the `START-TIMER` instruction, the `TIMER-END?` instruction has to render the result "true" at least once within a time of 4 million time increments, in order for the register to be set to negative, before number overflow takes place. On the other hand, the "timeout" status can be enforced by setting the used register onto a negative value.

The number of the time base register can be defined in one register. After switching on, the "runtime register" (in user increments) is used. Yet, any user register can be applied.

Example:

```
    POS [Achse=21, Pos=..., v=...]  
    START-TIMER [Reg=rMonitoring, time=100]  
    ...  
    DELAY 20  
    ...  
    ...  
WHEN  
    TIMER-END? Reg=rMonitoring  
    OR  
    AXARR 21  
    THEN  
    ...
```

3.11.1.2 Special Registers for Time Instructions

Special Registers "User Time Base in ms"

In this register, the **time increments** of the controller are defined, namely in units of ms (milliseconds). After reset this value is 100, that is, the time increments is 100 ms by definition. Into this register, values from 1 to 255 can be input, whereas values that are smaller than 10 should not be input.

In case of the `DELAY` instruction, the respective number is loaded into the time register of the task, which is then waited to become zero. The following two program sequences are to demonstrate this (Example NANO-B):

<pre>TASK 0 ... REGISTER_LOAD [2300 with 10] WHEN REGZERO 2300 THEN ...</pre>	<pre>TASK 0 ... DELAY 10 ...</pre>
---	------------------------------------

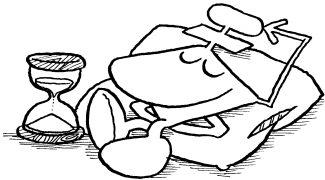
These two program arts have got exactly the same function. In some cases it may be useful, though, to use a time register, as between loading the time register and querying on zero still further instructions can be carried out.



Note:

Using time registers can also be quite tricky! Thus, it is very dangerous and not to be recommended to directly load the `DELAY` function, as well as the time register of the task, into the very same task. This way, "infinite" delays may result. For this reason, rather use `START-TIMER` and `TIMER-END?` instructions.

3.11.2 NOP



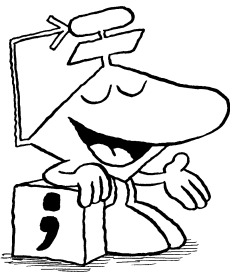
The

NOP

instruction is only of importance for the operating system.

As `NOP` is considered to be a "real" instruction, it will be processed in the program, which helps to have very short deceleration times in programs implying difficult timing.

3.11.3 The Commentary Character



The ";" (**colon**) actually is not an instruction, but it only helps to add a commentary line to the program text. Thus, more extensive commentaries can be written than those that can be filled into the commentary column behind an instruction.

As a mere commentary will follow, the entire line will be eliminated by the controller compiler during translation of the source program. Thus, neither memory space nor program processing time will be needed in the controller.

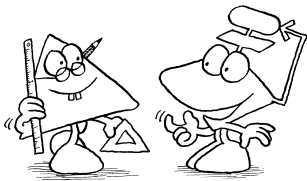
3.11.4 Special Functions

Special-
functions
only for PASE-E
and DELTA

Three internal controller functions can be called up by the

SPECIALFUNCTION

instruction:



Function 1: **Initialise axis**

Function 4: **BCD -> HEX transfer**

Function 5: **HEX -> BCD transfer**

Indirect
addressing
is possible

The parameters p1 and p2 can be indirectly defined for all functions.

Axis Initialisation

For axis initialisation use

SPECIALFUNCTION [#1, p1=<Par1>, p2=<Par2>]

This function is to serve initialisation of axis boards. Register values are copied from one memory range to another. This instruction has been described extensively in *Chapter 3.6.2*.

BCD -> HEX Transfer

The

SPECIALFUNCTION [#4, p1=<Par1> p2=<Par2>]

BCD switches,
for example,
can be
queried

serves the transfer of binary coded decimal numbers (BCD) into binary numbers.

This transfer can, for example be used for the location of values that have been written by BCD switches, that are connected to an input board.

The two parameters of this function are

Parameter 1 -> **Source Register Number**

Parameter 2 -> **Destination Register Number**

The bits of a source register are interpreted as a BCD number, then they are transferred into a binary number and written into the destination register. Four bits of the source register will make up a decimal place. Four places can be dealt with as a maximum.

BCD number in the source register:

Bit 0 to 3 -> last decimal place ("unit places")

Bit 4 to 7 -> second but last decimal place ("tens places")

Bit 8 to 11 -> third but last decimal place ("hundreds places")

Bit 12 to 15-> fourth but last decimal place ("thousands places")

Example:

Register 100 is to have the following value:

0101 1000 0011 0110 = 22582

but the value of the BCD number stored this way is

5 8 3 6 = 5836

The instruction

SPECIALFUNCTION [#4, p1=100, p2=101]

causes register 101 to have value 5836.

HEX -> BCD Transfer

SPECIALFUNCTION [#5, p1=<Par1> p2=<Par2>]

BCD-
controlled
displays, for
example, can
be controlled
this way

serves the transfer of binary numbers into binary coded decimal numbers (BCD). Thus it corresponds to the reversed special function 4.

Parameter 1 -> source register number
(binary number)

Parameter 2 -> destination register number
(BCD number)

3.11.5 The LIMITS Instruction



**LIMITS in
input
condition**

A very helpful instruction saving program codes:

```
LIMITS [reg.no, lower limit, upper limit]
```

This instruction can be applied in many ways:

1. LIMITS after IF or ELSE

Here the value of the register specified by the `LIMITS` instruction is checked on being in the interval which is defined by an upper and lower limit. The result of this operation is **true (1)** or **false (0)**.

**LIMITS in the
output
instruction**

2. LIMITS after THEN or ELSE

Here the value of the register specified as well by the `LIMITS` instruction is checked on being in the interval which is defined by an upper and lower limit. The result of this operation is the following, though:

- a) **The value is lower than the interval:**
In this case this value is replaced by the value of the lower limit.
- b) **The value is higher than the interval:**
In this case this value is replaced by the value of the upper limit.
- c) **The value is in between the limits of the interval:**
This value is kept.

Limits can also be defined by indirect or double indirect addressing.

3.11.6 Word Processing

In this chapter, the following instructions will be explained:

WAND

WOR

WXOR

With the help of these three instructions entire registers can be logically connected with each other bit by bit.

These logic connection instructions can be applied in the same way as the arithmetic operators + - * / .

They can be used in one and the same operation, yet there are no differences in priority.

Following, the instructions will be explained with the help of examples:

WAND

```
1)  REG 0
     =
     b010101010101010101010101
     WAND
     b001001001001001001001001
```

The first bit of the first number is connected by **AND** with the first bit of the second number.

The second bit of the first number is connected by **AND** with the second bit of the second number.

etc.

Using 0 the bits are reset.

Using 1 the bits are kept.

In **Word-by-word AND**-connection, the resulting bit will only be '1', where the corresponding bits of the first number **and** of the second number have been set (= 1).

By the **AND**-connection of a certain bit with '0', the result bit is set to '0'; by the **AND**-connection with '1', the status of the bit is taken over into the result.

The result of each connection is stored as the corresponding bit in register **REG 0**, in order for value `b000001000001000001000001 = 266305` to be written into **REG 0**.

```
2)  REG 0
     =
     REG 1
     WAND
     h0000FF
```

In this attribution the eight bits of lowest value (`h0000FF = b00000000000000000000000011111111`) that have been written into **REG 1** are taken over into **REG 0** just as they are. The bits of higher value belonging to **REG 0** are set to '0'.



Using 1 the bits are set.
Using 0 the bits are kept

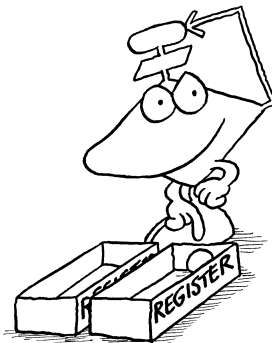
WOR

```
1) REG 0
   =
   REG 1
   WOR
   b0000000000000111100001111
```

In **Word-by-word OR**-connection, the resulting bits are set (=1), where the respective bits of the first number **or** the bits of the second number **or** the bits of both numbers are '1'.

In the **OR**-connection of a certain bit with '0' the status of the bit is taken over into the result bit; by connection with '1' the result is set to '1'.

The result, which has been stored in **REG 0** is designed as follows: **REG 0 = bxxxxxxxxxxx1111xxxx1111**. **x** is to define the bits, which are dependent on **REG 1**.



The bits are inverted by 1.
The bits are kept by 0.

WXOR

```
1) REG 100
   =
   46398
   WXOR
   123098
```

In the **Word-by-word EXclusive-OR**-connection those result bits are set to '1', where the respective bits of the two numbers have got different logic conditions. If the conditions are the same, the result will be '0'.

When a certain bit is connected with '0' by an exclusive-OR-connection, the status of this bit is taken over; in a connection with '1', the inverse bit value is written into the result bit.

```

46398 --> 0 0 0 0 0 0 0 0 1 0 1 1 0 1 0 1 0 0 1 1 1 1 1 0
123098 --> 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 1 1 0 1 1 0 1 0
  XOR
87524 <-- 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1 1 1 1 0 0 1 0 0

```

In register REG 100 value 87524 will stand in register REG 100 after the attribution.

```

2)  REG 100
     =
     REG 100
     WXOR
     hFFFFFF

```

Every bit of REG 100 is inverted by this attribution.

3.12 Network Instructions

**Mono-master-
network**

JETWay-R

**126 slaves
can be
connected**

A network is a connection of several controllers, remote I/Os or valve blocks. These can communicate with each other, that is, register values can be transferred from one unit to another one by the two following instructions. This network is a "mono-master network". This means, that one unit is the master (number 1), while the other units are the slaves (number 2 ...). This number can be defined in a register. Please note, that the two following instructions may only be input in the program of the master unit. Otherwise they will be considered as not defined.

The instructions

N-GET-REGISTER

N-SEND-REGISTER

serve taking register values from any controller into the master-controller and sending register values from the master-controller onto another controller.

**Easy
network
access by
50000-er
numbers**

Network operation by 50000-er numbers (*Chapter 3.12.3 Network Operation by 50000er Numbers*)

3.12.1 Sending Register Values to Slave Controllers



This is done by the following instruction:

```
N-SEND-REGISTER    [To    <network    no.>    from
reg.<SourceReg> to Reg.<DestinationReg>]
```

In this case, the number of the controller that is to be addressed must stand in the place of **network no.** This is the number of the slave controller, which is to be given the register value.

Indirect
addressing
is possible

Source Register Number is to define the register which must be read. This is a register belonging to the master controller. The parameter can also be defined indirectly (for example R10).

Destination Register Number is to define the slave controller register, into which the value must be written. This register number can also be addressed indirectly the pointer register being in the **master controller**.

Network errors
are indicated
by a special
flag

The "Network Timeout" special flag will indicate, whether an error (transfer error or timeout) occurred in the last transfer. For this reason, the flag should be checked after each transfer. If the flag is set, an error must have occurred, and data transfer can be repeated.

Example:

```

FLAG 100
  N-SEND REGISTER [To 2 from reg.100 to reg.200]
IF
  FLAG fsNetwork-Timeout
THEN
  GOTO 100
THEN
  ...

```

Transfer errors are indicated by special flag "network timeout"

In this example the value of register 100 is sent from the master controller to controller no. 2. If an error occurs, the special flag "network timeout" will be set, otherwise it will be reset; for this reason the program will return to label 100 to repeat the sending procedure. In this example, sending of the register value will be repeated over and over again, until the transfer has been carried out successfully.

3.12.2 Getting Register Values from a Slave Controller



This is done by the instruction

```

N-GET REGISTER [from <Net No>Reg.<SourceReg>, Reg
  here=<DestinationReg>]

```

The parameters to be defined have got the following meaning:

Network No.

is the network number of the slave controller out of which a register is to be read.

Source Register Number

Indirect addressing is possible

is the number of the slave controller register which is to be read (indirect addressing is possible).

Destination Register Number

is a register number of the "own" controller, into which the value is to be written (indirect addressing is possible).



Remark:

In the case of indirect addressing the registers, into which the source and destination register numbers have been written, are always on the **master controller**.

Spezial flag "Network-Timeout" is to report network errors

Here "Network-Timeout" will also be set in case of an error.

Examples:

1)

```

LABEL 100
  N-GET-REGISTER [from 2Reg.100, Reg here=200]
IF
  FLAG fsNetwork-Timeout
  THEN
    GOTO 100
  THEN
    ...

```

Here register 100 is read by the slave controller of network number 2 before being copied into register 200 of the master controller. With the help of special flag "network timeout" enquiry is made, whether an error has occurred.

If this is the case, register 100 of the slave controller is read again.

```
2)   REGISTER_LOAD [100 with 1000]
LABEL 100
      N-GET REGISTER [from 2Reg.R100, Reg here=R100]
      IF
        FLAG fsNetwork-Timeout
      THEN
        GOTO 100
      THEN
        REGINC 100
      IF
        REG 100
        <
        1030
      THEN
        GOTO 100
      THEN
```

By this program a whole register range (registers 1000 to 1029) are copied by the slave controller of network number 2 onto the master controller, as well as into registers ranging from 1000 to 1029.

A "collective report" of whether a transfer error has ever occurred, can be found in the special flag "Collective Report of a Network Error", which will be set in case of an error, yet will not be reset by the operating system. Resetting can only be done from the user program.

3.12.3 Network Operation by 50000er Numbers

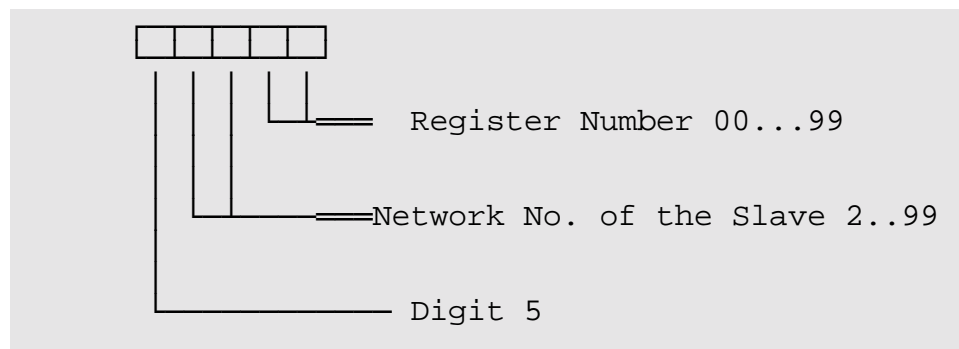
3.12.3.1 Addressing the Registers

The same access to master and slave registers

Access to control registers by a master controller only differs from an internal REGISTER_LOAD instruction in the parameter numbers. Apart from this number the program sequences for access to an internal register and a slave register are identical.

Register Number 00...99

The register number has got the following pattern:



With the help of these register numbers the master controller can have access to a window of 100 registers in the slave controller.

Example:

Register 62 of the slave controller with the network number 32 is addressed by the master controller by the instruction

```
REGISTER_LOAD [ 100 with R(53262) ]
```

By the numeric offset register, "windows" for slave register access are set

If access to a register is to be made, the number of which is greater than 99, a number offset value is to be input into the special register "number offset register". When the registers of the slave controller are accessed by the master controller, this value is added to the register number in the program of the master controller.

The instruction

```
REGISTER_LOAD [ 100 with R(53262) ]
```

in the program of the master-controller plus a value of 200 in the special register "number offset register" of the slave-controller of network number 32 has effective access to register 262 of the slave controller.

Special Register "Register Number Offset"

This value is added to the register number in the program of the master controller. The sum shows the number of the register, which, in the slave controller, is really accessed by the master controller.

Value after reset: 0

**Note:**

By the N-SEND REGISTER instruction the "register window" is set into the required range by the special register "register number offset". Then, operations can be made in this "window" with the help of the 50000er-numbers.



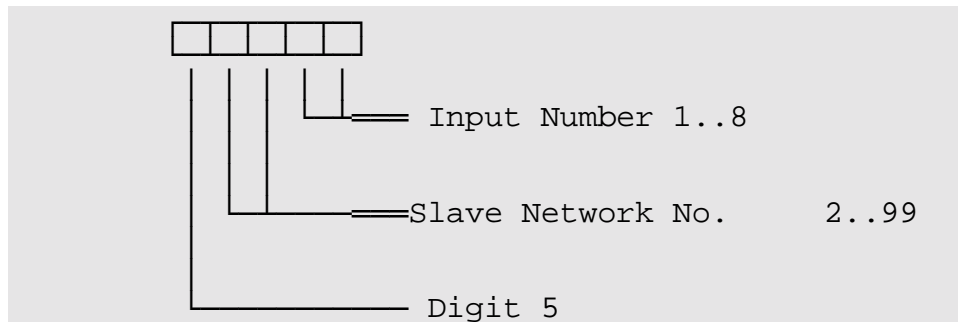
3.12.3.3 Addressing of Inputs, Outputs and Flags

Addressing of Inputs

The same access to master and slave inputs

Access to inputs of the slave-controller by the master-controller only differ from an internal master input instruction in their parameter number. Apart from this number, the program sequences for an access to a master input and a slave input are identical.

The input number is designed according to the following pattern:



"Windows" for the slave input access are set with the help of the number offset register!

The number value out of the corresponding number offset register for inputs is added to the input number defined in the input parameter. The resulting input will be addressed.

Special Register "Input Number Offset":

Number offset for inputs; the register is on the slave controller.

This value is added to the input number, which is in the program of the master-controller. The sum makes up the number of the input in the slave controller, which is actually accessed by the master controller.

Example:

Input 108 in the slave controller of network number 5 is addressed from the master-controller by the input

```
INPUT 50508
```

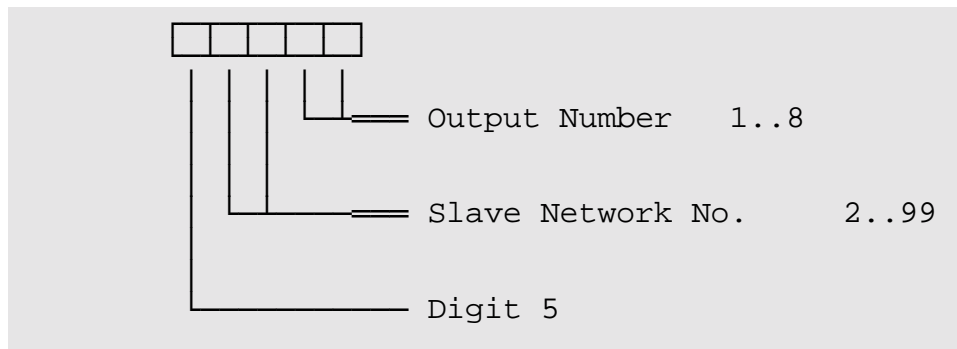
Before this,, value 100 must be written into special register "input number offset" (on the slave controller).

Addressing the Outputs

The same access to master and slave outputs

Access by the master controller to outputs of the slave controller only differ from an internal master output instruction in the parameter number. Besides this number, the program sequences for access to both a master and a slave output are identical.

The output number is made up according to the following pattern:



"Windows" for the slave output access are set with the help of the number-offset register!

The number value of the number offset register for outputs is added to the corresponding output number that has been defined in the output parameter. The resulting output will be accessed.

Special Register "Output Number Offset":

Output number offset: the register is on the slave controller.

This value will be added to the output number in the program of the master controller. The sum shows the number of the output, which, in the slave controller, is really accessed by the master controller.

Example:

Output 108 in the slave-controller of network number 5 is addressed by the instruction

OUTPUT 50508

of the master controller.

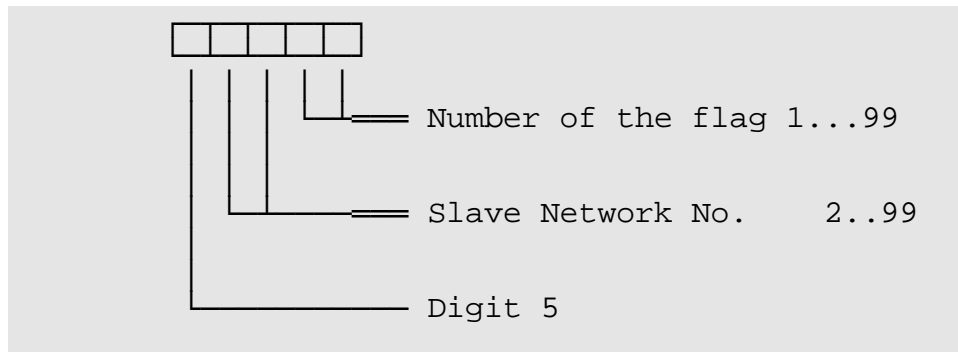
Before this, value 100 must be written into the special register "Output Number Offset" (in the slave controller).

Addressing the Flags

The same access to master and slave flags

Access to flags of the slave-controller by a master controller only differs from an internal master-flag instruction in the parameter number. Apart from this number, the program sequences for access to a master-flag and a slave-flag are identical.

The flag number has got the following pattern:



"Windows" for the slave output access are set with the help of the number-offset register!

The number value of the number offset register for outputs is added to the corresponding output number that has been defined in the output parameter. The resulting output will be accessed.

Special Register "Flag Number Offset":

Flag number offset; the register is on the slave controller. This value will be added to the output number in the program of the master controller. The sum shows the number of the output, which, in the slave controller, is really accessed by the master controller.

Example:

Flag 154 in the slave controller of network number 12 is accessed by the master controller with the instruction

```
FLAG 51254
```

Before this, value 100 must be written into the special register "Flag Number Offset" (in the slave controller).

3.12.4 Special Registers / Flags for Network Operation

Flag 2110

Errors in the latest network instruction are indicated (check sum or timeout).

Flag 2111 "Collective Report of a Network Error"

Errors that have occurred at a network instruction since reset of the controller (which is accumulating flag 2110).

Register "Network Number Network 1"

Network number network 1.

Register "Network Number Network 2"

Network number network 2.

Register "Network Reaction Time"

The network reaction time is defined in milliseconds. The timing is started, when a network instruction is carried out, and it ends, when the response of the other controller has arrived via network (this is mostly dependent on the load of the other controller). Any of these functions refers to the master network of a controller - this is the network of a controller to which this master belongs. A controller can only be master of one network at one time as a maximum.

Register "Execution Time Network Instruction"

The processing time of the network instruction carried out last has been written into this register (in milliseconds). This is the time written in the register "network reaction time" plus the time it will take, until the response of the other controller has been processed by the corresponding program task of the own controller, and until this task is continued (dependent on the load of the other and the own controller). Any of these functions refers to the master network of a controller - this is the controller network to which these masters belong. A controller can only be master of one network at one time as a maximum.

Register "Timeout Network Access"

Timeout time in milliseconds. Presetting: 250. After this time the network instruction will be terminated, as the accessed controller has not answered (it might have switched off).

Register "Number of Check Sum Errors in Network Reception"

Check sum error. The content is increased by one. To evaluate the quality of network transfer, this register can be read from time to time.

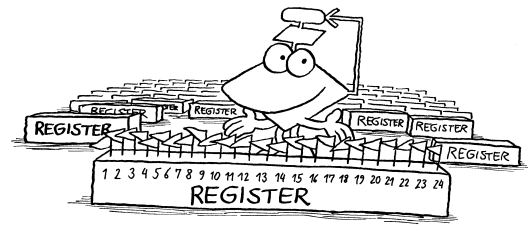
4. Description of the Memory

This overview has been kept in as general terms as possible and has not been specified on any certain controller

The memory that is available to the user of the control system is divided into registers and flags. This chapter is to inform about the design of the entire memory. Many registers used by the operating system will also be described, as they can be useful in some cases. Yet, please be careful when special registers are to be changed.

4.1 Basics on Registers and Flags

4.1.1 Registers



Registers are the memories of the controller, in which the data are kept

Registers are the memories of the controller, where all the necessary numeric values are stored. The operator can use part of these registers as well, namely to store values or to use them for calculating. The registers can either be loaded from the program, or with the help of SYMPAS by transferring whole data blocks to the controller via PC This can be helpful in many cases, as loading registers in a more complex program can take lots of space.

A difference is made between **integer registers**, **floating point registers** and **special registers**. All registers are marked by a number. Below, the three register types will be explained:

Integer Registers

Register width:
24 Bit ->
23 Bit plus sign

These registers are 24 Bit wide registers in which an **integer number** between - **8388608** and **8388607** has been stored. The sign of the number is stored by the Bit of highest value. The value of these registers can also be defined as a binary (b) or a hexadecimal number (h). For this, see the coding further below.

The specific
register
numbers can
be taken from
the respective
manual

The register numbers can be taken from the respective controller manuals. Here a general survey over the PROCESS-PLC registers is to be given:

The Structure of a Register:

In the 24th Bit the sign, in the other 23 Bits a number is stored, which corresponds to the binary value of these 23 Bits:

If the sign bit is zero, this value will exactly be the same as the register value.

If the sign bit is one, though, the number can be calculated by adding this binary value to -8388608 (= -h800000).

Examples:

Binary Number (24 Bit):	Hex Number	Dec Number
b00000000000000000000000001101111	h00006F	111
b01111111111111111111111111111111	h7FFFFFF	8388607
b10000000000000000000000000000000	h800000	-8388608
b11111111111111111111111111111111	hFFFFFF	-1
b100000000000000000000000011010011	h8000D3	-8388397

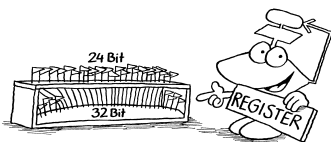
Floating Point Registers

There is no floating point register in NANO-A

These registers are 32 Bit wide and serve storing real numbers, which are, generally spoken, any fractures in the range of

$$- 10^{15} \text{ to } + 10^{15}$$

The amount of the smallest possible number is around 10^{-15} .



The accuracy of calculating is around 7 significant places, as only this amount of places can be stored in 32-Bit registers.

They serve detailed calculating, even of fractions. When fractions are assigned to an integer register, the decimal places will always get lost. If, for example, value -2,5 (result of a division) is loaded into an integer register, value -2 will be written there.

Another important function of floating point registers is the calculation of expressions, where results greater than 8 millions must be expected. In an integer register this can

lead to actually undefined values. The example below is to illustrate this problem:

The register numbers can be taken from the respective controller manuals. Below, a general overview over the PROCESS-PLC registers is to be given.

Example: Simple assignment

```
REG 1
=
2
*
5'000'000
```

When value 10'000'000 is assigned to the integer register 1, there will be the following result:

The number, which is presented as a binary number is loaded into the register. Yet, as the number is longer than the register, Bits from the beginning will get lost, or, to put it differently, Bit 23 (sign) is occupied. The result will be the following:

```
10'000'000 = h989680
           = b1001'1000'1001'0110'1000'0000
```

```
-> Reg 1 = -6'777'216
```

Special Register

Mainly, there are two kinds of special registers: One kind is placed on intelligent expansion modules to store parameters or status information of these modules (these will be extensively described in the respective controller manuals in context with the specific modules). Further,

there are registers which are used by the operating system of the controller.

Please be careful when using special registers

The numbers of the special registers can be taken from the respective controller manuals. Below, a general survey will be given on the registers of the PROCESS-PLC.

Registers Combining Flags:

Exemplary combining is illustrated by NANO-B numbering

The special register numbers combining flags can be taken from the respective controller manuals. Below, NANO-B is used as an example. Flags 0 to 255 are combined in registers 2600 to 2610.

Reg	2600	Flags	0	to	23
Reg	2601	Flags	24	to	47
Reg	2602	Flags	48	to	63
Reg	2603	Flags	64	to	87
.
Reg	2610	Flags	240	to	255 *)

*) Register 2610:

As all registers, register 2610 consists of 24 Bit. **In only the first 16 Bits of these, flags 240 to 255 are combined.**

Registers Combining Inputs or Outputs

Easy access to several inputs or outputs combined in registers

In various controller registers, **8, 16 or 24 inputs** have been combined in one register. The same applies to digital **outputs**..

The numbers of special registers overlapping with inputs or outputs can be taken from the respective controller manuals. Please find an illustration using DELTA below.

32 registers of 8 inputs each

These 8 inputs are written into Bits 0 to 7; all the other Bits (8 bis 23) are 0. This means there is a value range for these registers from 0 to 255. The **32 registers from 62464 to 62495** have got 8 inputs each:

Exemplary numbering:
DELTA

RegNo	Inputs		RegNo	Inputs
62464	101 - 108		62480	301 - 308
62465	109 - 116		62481	309 - 316
62466	117 - 124		62482	317 - 324
62467	125 - 132		62483	325 - 332
62468	133 - 140		62484	333 - 340
62469	141 - 148		62485	341 - 348
62470	149 - 156		62486	349 - 356
62471	157 - 164		62487	357 - 364
62472	201 - 208		62488	401 - 408
62473	209 - 216		62489	409 - 416
62474	217 - 224		62490	417 - 424
62475	225 - 232		62491	425 - 432
62476	233 - 240		62492	433 - 440
62477	241 - 248		62493	441 - 448
62478	249 - 256		62494	449 - 456
62479	257 - 264		62495	457 - 464

32 registers
of 16
inputs each

These 16 inputs are written into Bits 0 to 15, all the other bits (16 to 23) are 0. This makes a value range from 0 to 65535 for these registers. The **32 registers from 62528 to 62559** correspond to 16 inputs each:

Exemplary
numbering
DELTA

RegNo	Inputs		RegNo	Inputs
62528	101 - 116		62544	301 - 316
62529	109 - 124		62545	309 - 324
62530	117 - 132		62546	317 - 332
62531	125 - 140		62547	325 - 340
62532	133 - 148		62548	333 - 348
62533	141 - 156		62549	341 - 356
62534	149 - 164		62550	349 - 364
62535	157 - 164		62551	357 - 364
62536	201 - 216		62552	401 - 416
62537	209 - 224		62553	409 - 424
62538	217 - 232		62554	417 - 432
62539	225 - 240		62555	425 - 440
62540	233 - 248		62556	433 - 448
62541	241 - 256		62557	441 - 456
62542	249 - 264		62558	449 - 464
62543	257 - 264		62559	457 - 464

**32 registers
of 24 inputs
each**

These 24 inputs are written into Bits 0 to 15, all the other bits (16 to 23) are 0. The sign of the resulting integer value is determined by Bit 23, which always corresponds to the input of the highest number. The value range of these registers equals the value range of all integer registers, this is, from -8388608 to 8388607. The **32 registers from 62592 to 62623** correspond to 24 inputs each:

Exemplary
numbering
DELTA

RegNo	Inputs		RegNo	Inputs
62592	101 - 124		62608	301 - 324
62593	109 - 132		62609	309 - 332
62594	117 - 140		62610	317 - 340
62595	125 - 148		62611	325 - 348
62596	133 - 156		62612	333 - 356
62597	141 - 164		62613	341 - 364
62598	149 - 164		62614	349 - 364
62599	157 - 164		62615	357 - 364
62600	201 - 224		62616	401 - 424
62601	209 - 232		62617	409 - 432
62602	217 - 240		62618	417 - 440
62603	225 - 248		62619	425 - 448
62604	233 - 256		62620	433 - 456
62605	241 - 264		62621	441 - 464
62606	249 - 264		62622	449 - 464
62607	257 - 264		62623	457 - 464

Examples:

1)

```
REGISTER_LOAD [ 62528 with 255]
```

Here, value 255 is loaded into the register corresponding to inputs IN 101 to IN 116. This way, the bits of lowest value are set, while the other ones are cleared. In consequence, inputs IN 101 to IN 108 are set (this is, active), while inputs IN 109 to IN 116 are cleared.

2)

Easy masking
of inputs and
outputs

These registers can also be very useful, in connection with the register instructions `WAND`, `WOR` and `WXOR`.

```
REG 62784
=
REG 62784
WAND
b000000001010101010101010
```

This assignment causes all odd-numbered outputs of `OUT 101` to `OUT 116` (`OUT 101`, `OUT 103`, `OUT 105` etc.) to be blinded out, respectively cleared. The other outputs will be kept as they are.

4.1.2 Flags

Flags have got
value 1 or 0

Flags are actually one-bit registers; that is, values 1 or 0 can be stored in them. The flags can be used for marking certain conditions. Thus, very easily programmable timing of various tasks can be achieved. For flags, a difference will be made between special and "normal" flags. The special flags are used by the operating system to store one status each, for example pressing a key of the input keyboard, error reports, etc.



All flags can be changed by flag instructions - they can be set, cleared or just queried. A detailed description of those flag instructions can be found in *Chapter 3.6.5 Flags and Flag Instructions*.

Special Flags

Please be
careful when
dealing with
special flags!

The special flags are used by the operating system to indicate certain conditions, or for function control. The numbers of special flags can be taken from the respective controller manuals, where a general overview over registers and flags of the PROCESS-PLC has been given.

5. Realtime Clock



5.1 Overview, Function

A realtime clock has been integrated into various PROCESS-PLCs, which is battery buffered independently from the RAM store.

The realtime clock will be explained by the example of DELTA register numbers

The register numbers used as an example here refer to the DELTA.

There are two register sets of 8 registers each. Register set 1 (62920 to 62927) can be written into and read. By these register numbers, writing access is directly made into the realtime clock module (setting of the time), reading access is directly made out of the time module.

Besides that, there is register set 2 (62912 to 62919). This second register set has got the following meaning: If, by the program, a certain time is waited for, change of the operands (time, ...) during the comparison operation must be prevented. For this reason, all realtime data are copied into the registers of register set 2 at each reading access to register set 1. There, they will be available without having been changed, until another reading access to a register of set 1 is made (see exemplary program).

For setting the clock the values are written into register set 2 and then completely transferred to the realtime clock by writing into one of the registers of set 1.

5.2 Register Description

Register set 1	Register set 2	Data	Range
write/read direct	read/write buffer		
62920	62912	seconds	0-59
62921	62913	minutes	0-59
62922	62914	hours	0-23
62923	62915	12/24h format	0,128
62924	62916	day of the week	1-7
62925	62917	day (date)	1-31
62926	62918	month	1-12
62927	62919	year	0-99

The following special function has been hidden in register 62924: The content of this register is "Day of the Week" 1=Sunday, 2=Monday, 3=Tuesday, etc.

In order to display, respectively print, the time in the usual way, the value range of special register **61454** has been expanded. If this register has got value 2, the sign place is suppressed in a DISPLAY_REG instruction (see exemplary program).

5.3 Realtime Clock: An Exemplary Program

An exemplary illustration of the realtime clock will be given here by the NANO register numbers

With the help of a battery buffered register set access can be made on the realtime clock functions.

Register Description

Register Set: Realtime Clock 2911 .. 2917	
Register	Function
2911	Seconds
2912	Minutes
2913	Hours
2914	Day of the Week
2915	Day
2916	Month
2917	Year

Exemplary Program: Realtime Clock

In the following exemplary program, the data of the realtime clock will be presented on the user interface.

The following trick has been applied to get leading zeros, when minutes and seconds are displayed:

In flush left number display with the help of register 8205, it can be determined how many places are to be displayed. If less places are permitted than there are significant places in the number, **leading** places will be omitted.

This fact is made use of in the program by adding value 100 to seconds and minutes and not displaying the leading 1 afterwards.

```

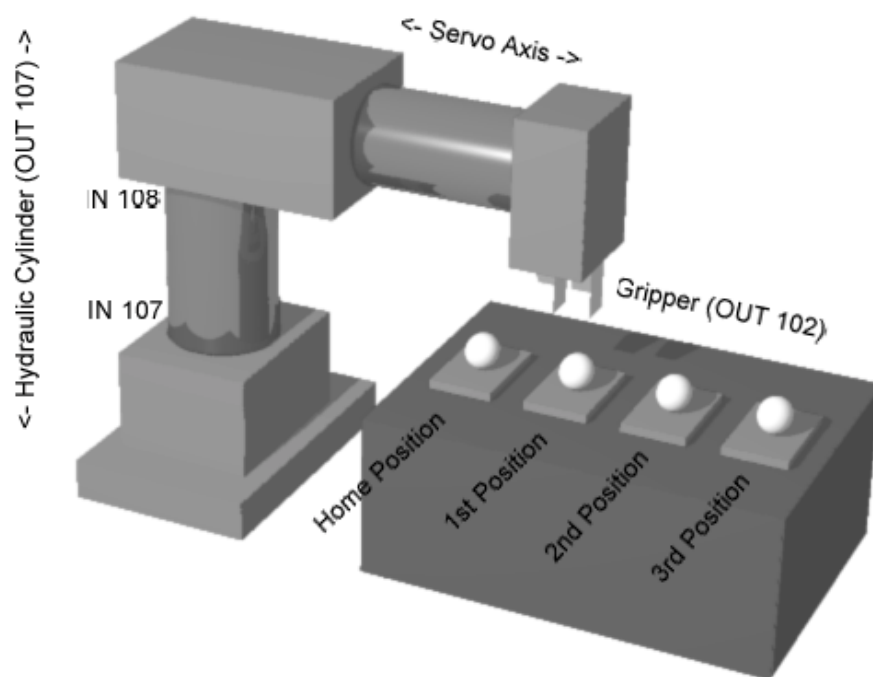
0: TASK 0 -----
1:      ;
2:      REGISTER_LOAD [2816 with 1]          ;no sign
3:      REGISTER_LOAD [2812 with 3]          ;2-place numbers
4:      DISPLAY_TEXT [#0, cp=1, "_The time is now:"]
5:      ;
6: LABEL 100
7:      SUBROUTINE 900
8:      DELAY 5
9:      GOTO 100
10:     ;
11: LABEL 900
12:     DISPLAY_TEXT [#0, cp=27, ". .19 , : :"]
13:     DISPLAY_REG [#0, cp=25, Reg=2915]     ;Day
14:     DISPLAY_REG [#0, cp=28, Reg=2916]     ;Month
15:     DISPLAY_REG [#0, cp=33, Reg=2917]     ;Year
16:     ;
17:     ;----- Time Display -----
18:     ;
19:     DISPLAY_REG [#0, cp=36, Reg=2913]     ;hour
20:     REG 900                               ;TRICK, to
21:     =                                     ;display
22:     REG 2912                               ;tens place,
23:     +                                     ;even if it has
24:     100                                    ;got value zero
25:     DISPLAY_REG [#0, cp=39, Reg=900]     ;Minute
26:     REG 900                               ;TRICK, to
27:     =                                     ;display
28:     REG 2911                               ;tens place,
29:     +                                     ;even if it has
30:     100                                    ;got value zero
31:     DISPLAY_REG [#0, cp=42, Reg=900]     ;second
32:     RETURN

```

6. Demonstrating Example: Handling-System

6.1 Problem Description

As an example, the controller program for a two axis machine has been explained below according to the following figure:



The vertical axis is moved downward by setting output 107 and upward again by resetting (hydraulic cylinder). Inputs 108 and 107 are active, when the basic position

(IN 108), respectively the working position (IN 107) of the vertical cylinder have been reached.

The horizontal cylinder is a servo-NC axis. The gripper is opened, respectively. closed, with the help of output 2.

Certain parts have to be taken, one after the other, from the basic position to three different depositing positions, which are free programmable by the user in teach-in mode; in manual mode the required position is driven to and stored by pressing a key on the display module.

Besides the automatic mode all motions are to be carried out by hand as well.

Further, the process will be supported by interactive input and output on the user interface.

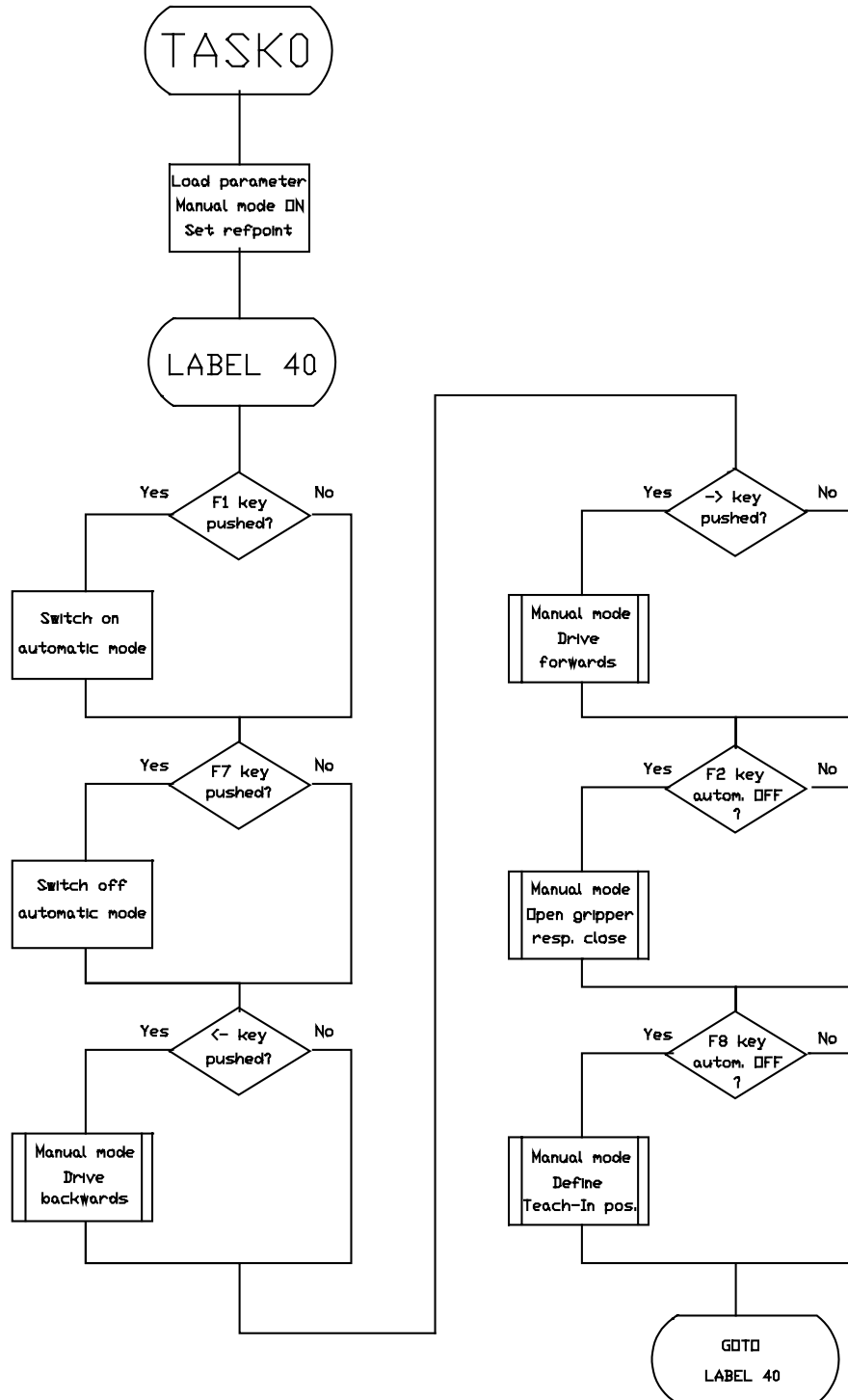
The following keys on the user interface (LCD9, LCD 10) can be used for process control:

Key:	Function:
F1	Automatic mode ON, manual mode OFF
F2	Gripper OPEN/CLOSED
F7	Manual mode ON, automatic mode OFF
F8	Teach-In; storing the basic position and the three stacker positions
<-	Manual mode "backwards"
->	Manual mode "forward"

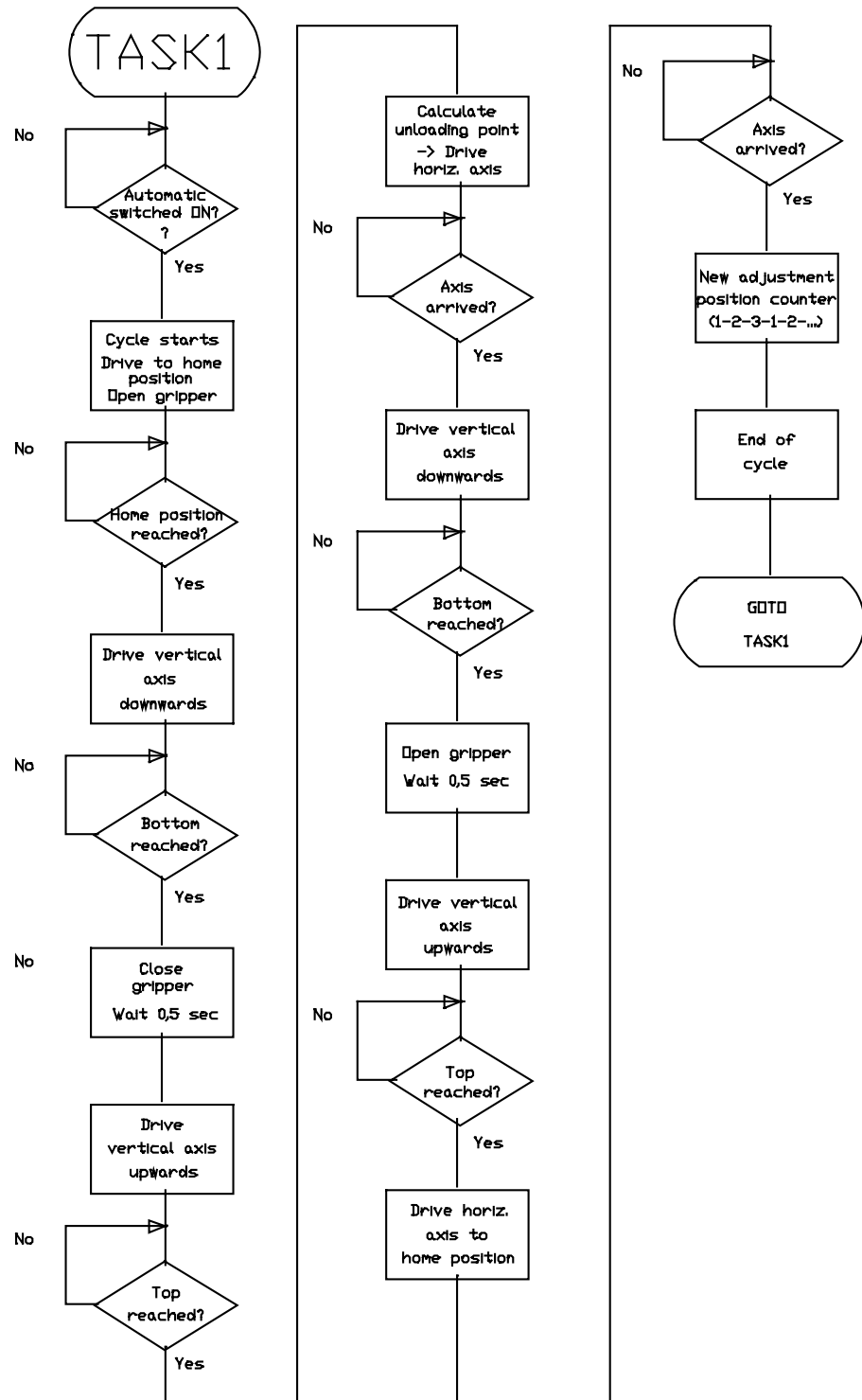
The program is divided into three main tasks. On the following pages an extensive overview over the structure of the three tasks, over program listing and symbol listing is to be given. Detailed comments are to explain the program structure.

6.2 Flow Charts of the Three Tasks

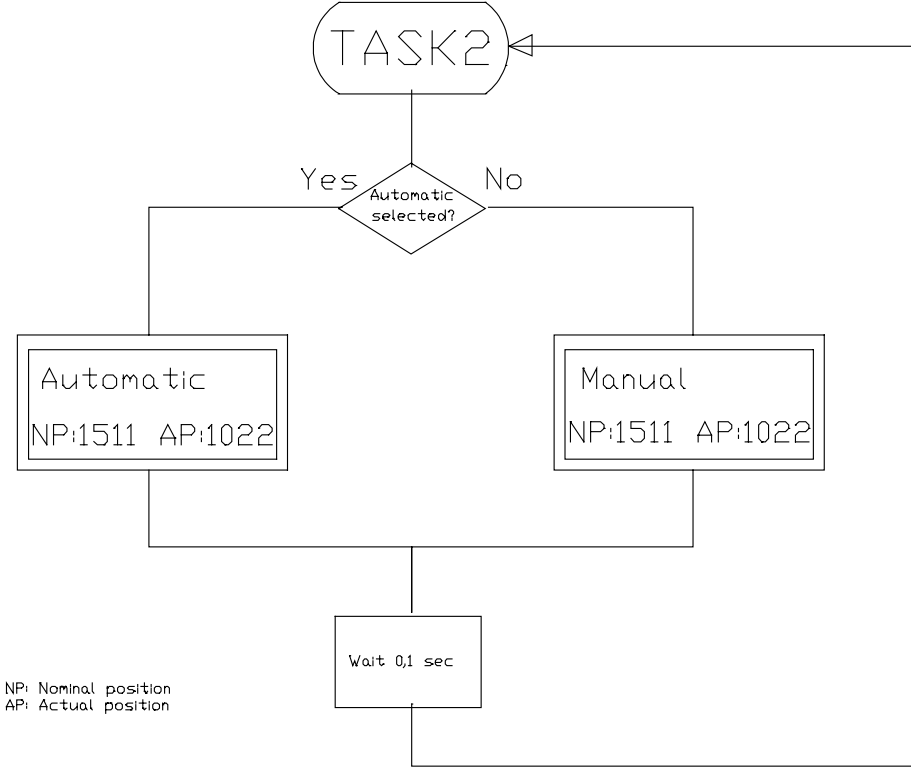
6.2.1 TASK 0 - Control Task



6.2.2 TASK 1 - Automatic Task



6.2.3 TASK 2 - Display Task



6.3 Program Listing

NANO-B - Program Listing page 1

JETTER PROCESS-PLC NANO-B

Version# : 1

```

0: TASK tInitialisation
1:   ; *****
2:   ;           TASK tInitialisation
3:   ;
4:   ; Initializes controller, carries out
5:   ; reference run and scans the function keys.
6:   ;
7:   ; *****
8:   ;
9:   THEN
10:  DELAY 2                               ;Wait 2/10 sec!
11:  COPY [n=4, from Start ramp to Start_Offset]
12:  ; -----
13:  ; Fix registers for start and stop ramp,
14:  ; destination window range and offset are
15:  ; set.
16:  ; -----
17:  -FLAG fAutomatic                       ;manual mode
18:  -FLAG fCycleisWorking                  ;no cycle is working
19:  -FLAG fAutoLED                         ;switch off auto-LED
20:  FLAG fManualLED                       ;switch on manual LED
21:  REGISTER_LOAD [rCommandreg with 3]    ;set reference point
22:  OUT oRelay                             ;switch relay on
23:  REGISTER_LOAD [rSlaveConfig with 3]  ;activate servo axis
24:  REGISTER_LOAD [rCycleCounter with 1];set cycle counter=1
25:  ;
26:  ; * * *
27:  ; After switching on the cycle counter is
28:  ; set to "1", so the machine starts in
29:  ; automatic mode with the first position
30:  ; for the workpiece put off.
31:  ; * * *
32:  ;
33:  ; -----
34:  ; -----
35:  ;           Loop function key scanning
36:  ; -----
37:  ;
38: LABEL lFctscan
39:   ;
40:   ; -----
41:   ; Scanning display keys F1 and F7.
42:   ; (Automatic/manual mode switching)
43:   ; -----
44:   ;

```

```

45: IF
46:   FLAG fKey_F1                               ;F1 key pushed?
47:   THEN
48:     FLAG fAutomatic                           ;switch on automatic
49:     FLAG fAutoLED                             ;activate auto LED
50:     -FLAG fManualLED                         ;deactivate manual LED
51: IF
52:   FLAG fKey_F7                               ;F7 key pushed?
53:   THEN
54:     -FLAG fAutomatic                         ;switch off automatic
55:     -FLAG fAutoLED                           ;deactivate auto LED
56:     FLAG fManualLED                         ;activate manual LED
57:     ;
58:     ; -----
59:     ; Scanning cursor keys ~ and @ for axis
60:     ; motion in manual mode
61:     ; -----
62:     ;
63: IF
64:   ; ** The following three conditions are      **
65:   ; ** logically AND-linked.                  **
66:   ;
67:   FLAG fKeyBackwards                         ;<- key pushed?
68:   -FLAG fAutomatic                           ;automatic switched
69:   -FLAG fCycleIsWorking                     ;off?
70:   -FLAG fCycleIsWorking                     ;automatic cycle
71:   -FLAG fCycleIsWorking                     ;ended?
72: THEN
73:   CALL ManualBackwards                       ;* manual backwards *
74: IF
75:   ; ** Following three conditions are      **
76:   ; ** logically AND linked.              **
77:   ;
78:   FLAG fKeyForward                           ;-> key pushed?
79:   -FLAG fAutomatic                           ; automatic switched
80:   -FLAG fCycleIsWorking                     ;off?
81:   -FLAG fCycleIsWorking                     ;automatic cycle
82:   -FLAG fCycleIsWorking                     ;ended?
83: THEN
84:   CALL ManualForwards                       ;* manual forwards *
85:   ;
86:   ; -----
87:   ; Scanning of F2 key (open/close gripper)
88:   ;
89:   ; -----
90:   ;
91: IF
92:   ; ** The following three conditions are      **
93:   ; ** logically AND linked                **
94:   ;
95:   FLAG fKey_F2                               ;F2 key pushed?
96:   -FLAG fAutomatic                           ;automatic switched
97:   -FLAG fCycleIsWorking                     ;off?
98:   -FLAG fCycleIsWorking                     ;automatic cycle
99:   -FLAG fCycleIsWorking                     ;ended?
100: THEN
101:   CALL Gripper                              ;* Gripper OPEN/CLOSED *
102:   ;
103:   ; -----
104:   ; Scanning F8 key (Teach-In)
105:   ; -----
106:   ;
107: IF
108:   ; ** The following three conditions are      **
109:   ; ** logically AND-linked                **
110:   ;
111:   FLAG fKey_F8                               ;F8 key pushed?
112:   -FLAG fautomatic                           ;automatic switched
113:   -FLAG fCycleIsWorking                     ;off?
114:   -FLAG fCycleIsWorking                     ;automatic cycle
115:   -FLAG fCycleIsWorking                     ;ended?
116: THEN
117:   CALL Teach_In                             ;*Teach-In put-off
118:   ;pos*

```

```

110:      ;
111:      THEN                                     ;repeat function key
                                                ;scanning
112:      GOTO Fctscan                             ;(end of loop)
113:      ;
114:      ; *** End of function key scanning! ***
115:      ; *** ( End of Loop ) ***
116:      ; -----
117:      ; -----
118:      ;
119:      ;
120: TASK tAutomaticCycle
121:      ; *****
122:      ;          TASK tAutomaticCycle
123:      ;
124:      ; Automatic cycle:
125:      ; the workpieces are put down
126:      ; sequentially
127:      ; at the put-down positions 1 to 3.
128:      ; *****
129:      ;
130:      WHEN
131:      FLAG fAutomatic                         ;Automatic mode
                                                ;switched on?
132:      ; * * *
133:      ; The switching on procedure automatic
134:      ; ON/OFF is programmed in TASK
135:      ; "Initialisation" (0).
136:      THEN
137:      ;
138:      ; -----
139:      ;          Drive to home position
140:      ; -----
141:      ;
142:      FLAG fCycleIsWorking                   ;automatic cycle
                                                ;starts
143:      ; * * *
144:      ; This flag is reset at the end of this
145:      ; task, which is the end of the automatic cycle.
146:      ;
147:      ; If the user switches off the automatic
148:      ; during and working automatic cycle,
149:      ; the cycle is operated until its end,
150:      ; before manual operations can be started.
151:      ;
152:      ; (This flag is scanned in the TASK "Ini-
153:      ; tialisation" during the scanning of the
154:      ; function keys several times!)
155:      ; * * *
156:      -OUT oDriveVertical                     ;drive vertical
                                                ;cylinder
157:      -OUT oGripperOpenClose                ;upwards and open
                                                ;gripper
158:      NOP                                     ;(home position!)
159:      NOP                                     ;
160:      WHEN
161:      IN iHomePosition                       ;vertical cylinder
                                                ;above?
162:      THEN
163:      ; * * *
164:      ; Drive axis with automatic speed to
165:      ; home position!
166:      ; * * *
167:      POS [axis=1, pos=R(Homepos), v=R(SpeedAutomatic)]
168:      WHEN                                     ;Horizontal axis
                                                ;reached
169:      AXARR axis=1                            ;home position?
170:      THEN
171:      ;
172:      ; -----
173:      ;          Pick up workpiece at home position
174:      ; -----

```

```

175:      ;
176:      ; * * *
177:      ; Drive the vertical cylinder
178:      ; downwards!
179:      ; * * *
180:      OUT oDriveVertical          ;cylinder downwards
181:  WHEN
182:      IN iWorkingPosition        ;vertical cylinder
                                      ;down?
183:  THEN
184:      OUT oGripperOpenClose      ;close gripper
185:      DELAY 5                    ;wait 0.5 seconds
186:      -OUT oDriveVertical        ;drive vertical axis
187:      NOP                        ;upwards!
188:  WHEN
189:      IN iHomePosition          ;vertical cylinder
                                      ;above?
190:  THEN
191:      ;
192:      ; -----
193:      ; Drive to put-off position, which cor-
194:      ; responds to the register "CycleCounter".
195:      ;
196:      ; -----
197:      ;
198:      ; * * *
199:      ; With the help of the put-off position number
200:      ; (content of register "CycleCounter") the
201:      ; register number is calculated, in which
202:      ; the nominal position is stored.
203:      ;
204:      ; * * *
205:      ;
206:      REG rPositionReg          ;Calculation of
                                      ;register
207:      =                        ;number which contains
208:      REG rCycleCounter        ;the nominal position
209:      +                        ;of the corres. put
                                      ;down pos.
210:      RegrOffset_1
211:      ;
212:      ; * * *
213:      ; Drive horizontal axis to put-off position
214:      ; with the number, which is stored in re-
215:      ; gister "CycleCounter".
216:      ; * * *
217:      ;
218:      POS [axis=1, pos=RR(PositionReg), v=R(SpeedAutomatic)]
219:  WHEN
220:      AXARR axis=1              ;axis reached put off
221:      NOP                      ;position?
222:  THEN
223:      ;
224:      ; -----
225:      ; Put off workpiece at the actual put-off
226:      ; position.
227:      ; -----
228:      ;
229:      ; * * *
230:      ; Drive vertical cylinder downwards!
231:      ;
232:      ; * * *
233:      OUT oDriveVertical        ;cylinder downwards
234:  WHEN
235:      IN iWorkingPosition        ;vertical cylinder
                                      ;down?
236:  THEN
237:      -OUT oGripperOpenClose    ;open gripper
238:      DELAY 5                  ;wait 0.5 seconds
239:      -OUT oDriveVertical        ;drive vertical
                                      ;cylinder

```

```

240:      NOP                                ;upwards
241:  WHEN
242:      IN iHomePosition                    ;vertical cylinder
                                           ;above?

243:      THEN
244:      ;
245:      ; -----
246:      ;      Drive back to home position
247:      ; -----
248:      ;
249:      ; *                *                *
250:      ; Drive back horizontal axis to home
251:      ; position
252:      ; *                *                *
253:      ;
254:      POS [axis=1, pos=R(HomePos), v=R(SpeedAutomatic)]
255:  WHEN
256:      AXARR axis=1                        ;axis reached home
257:      NOP                                ;position?
258:      THEN
259:      ;
260:      ; -----
261:      ;      Prepare next cycle
262:      ; -----
263:      ;
264:      ; *                *                *
265:      ; The sequence of the three put-off positions,
266:      ; which is realized during automatic mode
267:      ; is: 1-2-3-1-2-3-1-....
268:      ; The following instructions secures, that
269:      ; the value of the register "CycleCounter"
270:      ; represents this sequence of put-off pos.
271:      ; *                *                *
272:      ;
273:  IF
274:      REG rCycleCounter                    ;the third put-off
                                           ;position
275:      <                                    ;of the cycle was
                                           ;served?
276:      3                                    ;
277:      NOP                                ;
278:      THEN
279:      REGINC CycleCounter                  ;increment cycle counter
280:      NOP                                ;by one!
281:      ELSE
282:      REGISTER_LOAD [CycleCounter with 1] ;repeat from the
                                           ;beginning
283:      NOP                                ;
284:      THEN
285:      ;
286:      ; -----
287:      ;      End of cycle
288:      ; -----
289:      FLAG fCycleIsWorking
290:      ; *                *                *
291:      ; The meaning of this flag is described
292:      ; at the beginning of this task (TASK
293:      ; "AutomaticCycle")
294:      ; *                *                *
295:      GOTO AutomaticCycle                  ;repeat from the beginning
296:      ;
297:      ;
298:  TASK tDisplay
299:      ; *****
300:      ;      TASK Display
301:      ;
302:      ; Displays the operation of automatic
303:      ; or manual mode and the nominal and
304:      ; actual position additionally.

```



```

305:         ;
306:         ; *****
307:         ;
308: IF
309:     FLAG fAutomatic                ;automatic selected?
310: THEN
311:     ;
312:     ; -----
313:     ; If automatic is selected the LCD
314:     ; displays "AUTOMATIC".
315:     ;
316:     ; $ = erases up to end of line!
317:     ; -----
318:     ;
319:     DISPLAY_TEXT [#0, cp=1, "AUTOMATIC$ "]
320: ELSE
321:     ;
322:     ; -----
323:     ; If manual mode is selected the LCD
324:     ; displays "MANUAL".
325:     ;
326:     ; $ = erases rest of the first line!
327:     ; -----
328:     ;
329:     DISPLAY_TEXT [#0, cp=1, "MANUAL$ "]
330: THEN
331:     ;
332:     ; -----
333:     ; Both cases display the values
334:     ; of the nominal and actual position
335:     ; in the second line.
336:     ; -----
337:     ;
338:     DISPLAY_TEXT [#0, cp=25, "NP: "]
339:     DISPLAY_REG [#0, cp=28, reg=NominalPosition]
340:     DISPLAY_TEXT [#0, cp=37, "AP: "]
341:     DISPLAY_REG [#0, cp=41, reg=ActualPosition]
342:     ;
343:     ; -----
344:     ; Additionally there is a delay of 0.1 se-
345:     ; conds inserted. Without this delay the
346:     ; this task would consume too much CPU time,
347:     ; because it would refresh the display content
348:     ; steadily. This capacities would not be
349:     ; available for the other tasks.
350:     ;
351:     ;
352:     ; -----
353:     ;
354:     DELAY 1
355:     GOTO Display
356:     ;
357:     ;
358:     ; *****
359:     ;           S U B R O U T I N E S
360:     ; *****
361:     ;
362: LABEL lManualBackwards
363:     ; -----
364:     ;           CALL ManualBackwards
365:     ;
366:     ; In manual mode the horizontal axis is
367:     ; moved backwards, until the key <-
368:     ; is released.
369:     ; -----

```

```

370:      ;
371:      THEN
372:      ;
373:      ; * * *
374:      ; Drive backwards with manual
375:      ; speed!
376:      ; * * *
377:      ;
378:      POS [axis=1, pos=Backwards, v=R(SpeedManual)]
379:  WHEN
380:      -FLAG fKeyBackwards          ;key <- released?
381:  THEN
382:      AXARR axis=1                ;stop axis
383:      RETURN
384:      ;
385:      ;
386:  LABEL lManualForwards
387:      ; -----
388:      ;          CALL ManualForwards
389:      ;
390:      ; In manual mode the horizontal axis is
391:      ; moved forward until the -> key
392:      ; is released.
393:      ; -----
394:      ;
395:  THEN
396:      ;
397:      ; * * *
398:      ; Drive forward with manual
399:      ; speed!
400:      ; * * *
401:      ;
402:      POS [axis=1, pos=Forward, v=R(SpeedManual)]
403:  WHEN
404:      -FLAG fKeyForwards          ;key <- released?
405:  THEN
406:      AXARR axis=1                ;stop axis
407:      RETURN
408:      ;
409:      ;
410:  LABEL lGripper
411:      ; -----
412:      ;          CALL Gripper
413:      ;
414:      ; The gripper is closed respectively
415:      ; opened by the subroutine.
416:      ; -----
417:      ;
418:      ; * * *
419:      ; Because this manual routine displays
420:      ; the opening or closing of the gripper
421:      ; on the LCD display, the task "Display"
422:      ; has to be stopped, else the LCD is
423:      ; filled with the character output
424:      ; of the "Display" task.
425:      ;
426:      ; * * *
427:      ;
428:  IF
429:      OUT oGripperOpenClose          ;gripper closed?
430:  THEN
431:      TASKBREAK #Display              ;interrupt display
432:      DISPLAY_TEXT [#0, cp=1, "_ Opening gripper"]
433:      -OUT oGripperOpenClose          ;open gripper
434:  ELSE

```

```

435:         TASKBREAK #Display                ;interrupt display
436:         DISPLAY_TEXT [#0, cp=1, "_ Closing gripper"]
437:         OUT oGripperOpenClose            ;close gripper
438:     WHEN
439:         -FLAG fKey_F2                    ;key F2 released?
440:     THEN
441:         DISPLAY_TEXT [#0, cp=1, "_ "]
442:         TASKCONTINUE #Display            ;activate display
443:         RETURN
444:     ;
445:     ;
446: LABEL lTeach_In
447:     ; -----
448:     ;             CALL Teach_In
449:     ;
450:     ; With the help of this subroutine the user
451:     ; defines the three put-off positions
452:     ; and the home position.
453:     ; -----
454:     ;
455:     ; *           *           *
456:     ; The home position and the three put-off
457:     ; positions are defined by the user by
458:     ; driving to the positions manually and
459:     ; defining with the display keys
460:     ; which of the four positions is to be set.
461:     ;
462:     ; Because also in this subroutine the
463:     ; communication is managed with the LCD
464:     ; the "Display" task has to be interrupted
465:     ; for teach-in time.
466:     ;
467:     ; *           *           *
468:     ;
469:     THEN
470:         TASKBREAK #Display                ;interrupt display
471:     ;
472:     ; *           *           *
473:     ; The user is asked for position definition
474:     ; following the pattern:
475:     ;
476:     ; 1 = home position
477:     ; 2 = 1. put-off position
478:     ; 3 = 2. put-off position
479:     ; 4 = 3. put-off position
480:     ; *           *           *
481:     ;
482:     DISPLAY_TEXT [#0, cp=1, "Input position no. (1-4)"]
483:     DISPLAY_TEXT [#0, cp=25, "1=HomePos.)$ "]
484:     USER_INPUT [#0, cp=40, reg=WorkingRegister]
485:     ;
486:     ; *           *           *
487:     ; The validity of the user defined
488:     ; position numbers is checked (range
489:     ; between 1 and 4).
490:     ; *           *           *
491:     ;
492:     IF
493:         LIMITS [reg=WorkingRegister, low=1, up=4]
494:     THEN
495:     ;
496:     ; *           *           *
497:     ; Now the register number, in which the
498:     ; actual position should be stored, is
499:     ; calculated using the user defined

```

```

500:      ; position number.
501:      ; *                *
502:      ;
503:      REG rWorkingRegister          ;Calculation of the
504:      =                             ;register number
505:      REG rWorkingRegister
506:      +
507:      RegOffset_2
508:      ;
509:      ; *                *                *
510:      ; Then the actual position is stored in
511:      ; the register calculated before.
512:      ; Now the user is informed about correct
513:      ; data input with the help of the display.
514:      ; After a delay of 0.5 seconds the task
515:      ; "Display" is activated again.
516:      ;
517:      ;
518:      ; *                *                *
519:      ;
520:      REGISTER_LOAD [R(rWorkingRegister) with R(rActualPositon)]
521:      DISPLAY_TEXT [#0, cp=1, "_ok! "]
522:      DELAY 5                        ;wait 0.5 seconds
523:      TASKCONTINUE #Display          ;activate display
524:      RETURN
525:  ELSE
526:      ;
527:      ; *                *                *
528:      ;           E r r o r   m e s s a g e !
529:      ;
530:      ; The display signals unvalid data
531:      ; input for 1 second, then the user is
532:      ; asked for data input again.
533:      ; *                *                *
534:      ;
535:      DISPLAY_TEXT [#0, cp=1, "_Invalid Pos.No.,"]
536:      DISPLAY_TEXT [#0, cp=25, "Please repeat!"]
537:      DELAY 10                        ;wait 1 second
538:  THEN
539:      GOTO Teach_In                    ;begin again
540:      ;
541:      ;
542:      ; *****
543:      ;           E N D   O F   P R O G R A M
544:      ; *****
545:      ; *****

```

The following registers have to be initialized before program start in the setup screen:

- Register 100 (start ramp) with 10
- Register 101 (stop ramp) with 10
- Register 102 (destination window range) with 0
- Register 103 (digital offset) with 32
- Register 110 (speed in automatic mode) with 10000
- Register 111 (speed in manual mode) with 1000

6.4 Symbol Listing

The corresponding symbol listing is carried out as follows:

NANO-B - Symbol listing of "DEMOPROG" V1 page 1

JETTER Automation Technique, NANO-B

S Y M B O L E D I T O R

T A S K S

tInitialisation	0	The controller is initialised, the reference run is managed and the function keys are scanned.
tAutomaticCycle	1	The workpieces are picked from the home position by the automatic task and put down at the put-down positions 1 to 3.
tDisplay	2	Activation of the automatic or manual mode and the current values of nominal and actual position are displayed.

S U B R O U T I N E S

ManualBackwards	200	Drives the horizontal axis in manual mode backwards, until the <- key is released.
ManualForwards	201	Drives the horizontal axis in manual mode forwards, until the -> key is released.
Gripper	202	If the gripper is open, it will be closed by the subroutine. If the gripper is closed, it will be opened by the subroutine.
Teach_In	203	With the help of this subroutine the user defines the home position and the three put-down positions.

L A B E L S

lFctscan	40	This label is the entrance into the task "Initialisation", which is jumped to at the end of this task again; so endless function key scanning is secured.
----------	----	---

 I N P U T S

iHomePosition	10	The input is active, if the vertical axis is placed at home position (above).
iWorkingPosition	7	The input is active, if the vertical axis is placed at working position (down).

 O U T P U T S

oDriveVertical	7	After the output is set, the hydraulic cylinder moves downwards, after reset of the output the hydraulic cylinder moves upwards.
oGripperOpenClose	2	If the output is set the gripper closes, if the output is reset the gripper opens.
oRelay	1	This output switches the relay for the servo driver output in the MSP1 simulator ON.

 R E G I S T E R S

rStartRamp	100	start ramp register
rStopRamp	101	stop ramp register
rDestWindow	102	destination window range register
rDigitalOffset	103	digital offset value register
rSpeedAutomatic	110	automatic mode axis speed register
rSpeedManual	111	manual mode axis speed register
rHomepos	120	position value of home position
rWorkingPos_1	121	position value of the 1 st put-off position (Teach-In!)
rWorkingPos_2	122	position value of the 2 nd put-off position (Teach-In!)
rWorkingPos_3	123	position value of the 3 rd put-off position (Teach-In!)
rCycleCounter	130	cycle counter (put-off positions 1-2-3-1-2-3-1-2...)
rPositionReg	131	This register contains the value of the current put-down (working) position
rWorkingRegister	200	Into this register the user can input the position numbers (1 to 4, 1 = home position) in Teach-In mode.


```

-----
-----
                                F I X E D   R E G I S T E R S
-----
-----

```

rCommandReg	1101	command register (the reference point is set with value 3)
rNominalPosition	1102	contains the value of the current nominal position, which is used in the task "Display"
rStart_Offset	1105	This register is used as offset between the number of the registers, in which the slave parameters and the corresponding slave registers themselves are stored (100...)
rActualPosition	1109	Contains the value of the actual position of the axis.(The value is used in the Teach-In task)
rSlaveConfig	1200	To activate the slave process servo controller of the PASE-Mikro this register is initialized with value 3 at the beginning of the program

 F L A G S

fAutomatic	1	Set, if automatic mode is selected.
fCycleIsWorking	2	Set, during operation of the automatic cycle.

 F L A G S F O R K E Y B O A R D
 S C A N N I N G

fKey_F1	221	flag for key F1 scanning (automatic mode ON)
fKey_F2	222	flag for key F2 scanning (gripper open/close)
fKey_F7	227	flag for key F7 scanning (automatic mode OFF)
fKey_F8	228	flag for key F8 scanning (Teach-IN)
fKeyBackwards	217	flag for <- key scanning
fKeyForwards	218	flag for -> key scanning
fAutoLED	201	flag for activation of the F1 key LED
fManualLED	207	flag for activation of the F7 key LED

 N U M B E R S

Regoffset_1	120	Difference value, which is added to the value of CycleCounter (register 130) to calculate the value of the put off position value register.
Regoffset_2	119	Difference value, which is used for calculation of the register, in which the position number (1 to 4) of the user input is stored.
Backwards	-500000	Nominal position for manual mode driving backwards. See subroutine "ManualBackwards".
Forwards	500000	Nominal position for manual mode driving forwards. See subroutine "ManualForwards".

Index

- 50000er Numbers
 - Addressing the Flags 255
 - Addressing the Inputs 251
 - Addressing the Outputs 253
 - Addressing the Registers 248
- Arithmetic Comparison 143
- Arithmetic Expressions 149
- AUTOEXEC.BAT 10; 13
 - Block 47
 - SYMPAS in the Network 105
- Block 47
 - Input AUTOEXEC.BAT 47
- Menu 45
- Boolean Expressions 140
- Change Directory 41
- Change Environment 41
- Change scale 64
- Combined Flags 263
- Combined Inputs 265
- Combined Outputs 265
- Command Line Parameters 105
- Commentaries 100
- Controller Type 7
- Convert Symbol Language 37
- Copy (Ctrl K-C) 46
- Copy Tools 7
- DA-File 51; 52
 - Setup 52
- Destination Directory 7
- Dialogue Language 8; 66
- Display ref. file ... 64
- DOS surface 42
- Edit view box 62
- Editor -> File.ENB 48
- Editor -> NANO-B 51
- Elementary Conditions 140
- Erase (Ctrl K-Y) 46
- Erase ref. display 64
- Error Messages 87
 - Miscellaneous Errors 95
 - Symbol Errors 88
 - Syntax Check 89
- Example
 - CLEAR_FLAGS 191
- Field
 - Text Register Field 33
- Fields 28
- File 54
- File.DA -> Register ... 51
- File.ENB -> Editor ... 49
- File.ENB -> NANO-B ... 51
- Files
 - Program and System Files Sympas 50
- Files (in General) 97
 - Backup Program File 97
 - Backup Symbol File 97
 - Configuration File 98
 - Configuration Setup 98
 - Desk file 98
 - Object File 99
 - Print File 98
 - Program File 97
 - ReverseTable 99
 - Symbol File 97
- Find 43
- Find Text 44
- Flags 141; 269
 - Combined 263
 - Special Flags 269
- Floating point register
 - Assignment 154
- Form feed 55
- Functions 165
 - Definition 165
 - Definition of the Function Text 166
 - Example
 - Input Condition 168
 - Example Output Instruction 167
 - Function Call-Up 166
- Hardware Installation 5
- Hardware Requirements 4
- INCLUDE Files 81
 - in the Program Editor 81
 - in the Symbol Editor 84
- INCLUDE Instruction 81
 - Main File 36; 82
 - Pick List 83
- Indirect Addressing 100
- Input 141
- Input Field 29
- INSTALL.EXE 6
- Installation 6
 - Start 8
- Instructions
 - 50000er Numbers 248
 - AXARR 220
 - BIT_CLEAR 186
 - BIT_SET 186
 - Boolean Expressions 140
 - CALL (Subroutine) 160
 - COPY 178
 - Delay 20; 137
 - Destination
 - GOTO 21
 - DISPLAY_REG 201
 - DISPLAY_TEXT 197
 - FLAG 190
 - Flags 141
 - Functions 165
 - Input 142; 192
 - Instruction Set 127
 - Instructions IF..THEN..ELSE 134
 - LABEL 157
 - LIMITS 238

- N-GET REGISTER 245
- NOMINALPOS 225
- NOP 234
- N-SEND REGISTER 244
- Numbers 150
- Output 142; 194
 - Reset 20
- Output Parameter 18
- REG 183
- REGDEC 184
- REGINC 184
- Register Bit 142
- REGISTER_LOAD 175
- REGZERO 184
- SPECIALFUNCTION 180; 235
- START-TIMER 229
- SUBROUTINE 157
- Subroutine (CALL) 160
- Task 17; 157
- TASKBREAK 226
- TASKCONTINUE 227
- TASKRESTART 227
- TIMER-END? 229
- USER_INPUT 205
- WAND 240
- WHEN..THEN 130
- WHEN_MAX...THEN 132
- WOR 241
- WXOR 241
- Instructions Input 16
- Interface 66
- JETWay-H 9; 10
 - Board for the PC 10
 - Setting in SYMPAS 12
- JETWay-H Board for the PC
 - AUTOEXEC.BAT 10
 - DIL Switches 11
- Left margin 55
- Listing 46
- Load block 46
- Load Environment 42
- Main File 36; 82
- Menu 34
 - Edit 43
 - File 39
 - Listing 54
 - File ... 54
 - Form feed 55
 - Left margin 55
 - Page settings ... 55
 - Printer 54
 - Sheet length 55
 - Monitor 56
 - NANO-B continue 57
 - NANO-B start 56
 - NANO-B stop 56
 - Setup 56
 - Project 35
 - Pull-down Menus 15
 - Scope 58
 - Display ref. file ... 64
 - Edit view box ... 62
 - Erase ref. display 64
 - PCX-File 64
 - Scale Y-axis ... 63
 - Stop recording 61
 - Transfer data ... 62
 - Trigger setup 61
 - Zoom 63
- Scope
 - Module Configuration 59
 - Start Recording... 60
- Special 65
- Transfer 48
 - Compare Editor -> NANO-B 51
 - Editor -> File.ENB 48
 - File.DA -> Register ... 51
 - File.ENB -> Editor ... 49
 - File.ENB -> NANO-B ... 51
 - NANO-B -> File.ENB 48
 - Register -> File.DA ... 51
- Menu Block
 - Block on/off 45
 - Copy (Ctrl K-C) 46
 - Erase (Ctrl K-Y) 46
 - Listing 46
 - Load block ... 46
 - Move (Ctrl K-V) 45
 - Save block ... 47
- Menu Edit
 - Find 43
 - Find Text ... 44
 - Next 44
 - Program 43
 - Replace ... 44
 - Replace Text ... 44
 - Restore Line 44
 - Symbol 43
- Menu File
 - Change Directory 41
 - Change Environment 41
 - DOS surface 42
 - Load Environment 42
 - New Program 40
 - New Project 39
 - Open 39
 - Pick List 41
 - Program Editor 41
 - Save 40
 - Save all 40
 - Save as ... 40
 - Setup Screen 42
 - Symbol Editor 41
 - Sympas 42
- Menu Scope
 - Change scale 64
- Module Configuration 59
- Monitor Functions 214
 - Restriction of 214
- Move (Ctrl. K-V) 45
- NANO-B -> File.ENB 48
- NANO-B continue 57
- NANO-B stop 56
- New Program 40
- New Project 39
- Next 44
- Numbers 150
- Object File 50
- Open 39
- Output 141
- Page settings 55
- Password 103

- PCX-File 64
- Pick List 41; 83
- Printer 54
- Program 43
- Program Editor 14; 41
 - Block Operations 23
 - Functions 22
 - Keys 22
 - Miscellaneous 24
 - Program Transfer 25
 - Storage of Cursor Position 24
- Program Input 16
- Program Language 67
- Program Setup 107
- Program Structure
 - Rules 113
- Program Transfer 25
- Programming
 - Exemplary Creation 16
- Programming Language 8
 - Functions 165
- Pull-Down Menu 34
 - Functions 34
 - Keys 34
- README 4
- Realtime Clock
 - An Exemplary Program 272
- Realtime Clock
 - Overview 270
- Recording ... 60
- Register -> File.DA ... 51
- Register Bit 142
- Registers
 - Basics 259
 - Floating Point Registers 261
 - Integer Registers 260
 - Integer Registers - Assignment 152
 - Special Registers 173; 262
- Registers in General
 - Basics 171
 - Combined Flags 263
 - Combining Inputs 265
 - Combining Outputs 265
 - DA-file 99
 - Floating Point Registers 172
 - Include Table 99
 - Instructions - REGISTER_LOAD 175
 - Integer Registers 171
 - Slave Registers 173
- Replace 44
- Replace Text 44
- Requirements 4
- Restore Line 44
- Save 40
- Save all 40
- Save as ... 40
- Scale Y-axis ... 63
- Scope Function 58
- Scope Screen 59
- Screens
 - Definition 14
- Settings 68
- Setup 56
- Setup Screen 14; 42
 - Axis Field 30
 - Binreg Field 32
 - Display Field 32
 - Fields 28
 - Flag Field 30
 - Functions 27
 - Index Field 30
 - Input Field 1 29
 - Keys 27
 - Output Field 30
 - Overview 26
 - Refresh Cycle 33
 - Text Register Field 33
- Sheet length 55
- Software 4
- Software Installation 6
- Stop recording 61
- Symbol 43
- Symbol Editor 14; 41
- Symbolic Notation 122
- Symbolic Programming 74; 121
 - Example 123
 - Symbol Editor 74
 - Symbol Editor - Creating a Symbol File 78
 - Symbol Editor - Example of a Symbol File 80
 - Symbol Editor - Functions 75
 - Symbol Editor - Keys 75
 - Symbol File 78
 - Symbolic Notation 122
 - Symbolic Notation - Example 123
- SYMPAS 42
 - Programming Environment 2
- SYMPAS Programming Environment
 - AUTOEXEC.BAT 13
 - Start 13
- Syntax Check
 - (ON/OFF) 71
 - Error Messages 89
- Tasks
 - Definition 113
 - Parallel Tasks 113
 - Program Structure - Rules 113
 - Rules for Task Switching 116
 - Task Structure 113
- Transfer data ... 62
- Trigger setup 61
- User Interfaces
 - Cursor Position 198
 - Device Number 197
 - Display Text 199
- Zoom 63