MICRO DESIGN

MC-1A Command Documentation

Command Reference for Programming the MC100-System using the MC-1A Language

Copyright © 1998, 1999, 2000 MICRO DESIGN GmbH. All rights reserved.

Document version: July 2005

MC-1A Command Documentation

Fehler! Formatvorlage nicht definiert.

This document or any accompanying software or firmware may not be reproduced without prior written consent by MICRO DESIGN Industrieelektronik GmbH. Violations will result in prosecution. MICRO DESIGN retains all rights to these documents as well as to associated software, hardware and/or firmware.

Trademarks mentioned in the text are used in consideration and recognition of the individual trademark owners. Separate identification of trademarks is not made everywhere in the text. Failure to mention or identify trademarks does not imply that the particular mark is not recognized or registered.

To the extent that system and/or applications software is associated with this document, you as the legal purchaser have a right to pass on this software together with MICRO DESIGN hardware components to your end customers without a license, as long as no other separate agreement to the contrary has been reached. If software associated with this document contains sample programs or sample applications, you may not pass these items on unchanged to your end customers. They are for your own use for training purposes only.

Limited Guarantee: MICRO DESIGN accepts no liability with respect to the accuracy of the contents of this document. Since errors, despite all efforts and checks, can never be completely avoided, we always appreciate any suggestions. Any guarantee with respect to the suitability of this documentation, associated devices and/or associated software is expressly not given.

We reserve the right at any time and without prior announcement to make technical changes to the software, hardware and/or firmware associated with this document.

Copyright © 1998, 1999 MICRO DESIGN Industrieelektronik GmbH.

Waldweg 55, 88690 Uhldingen, Germany

Telephone +49-7556-9218-0, Fax +49-7556-9218-50

e-mail: technik@microdesign.de

http://www.microdesign.de

We like to move it!™

Table of Contents

Space for your notes

Chapter 1 Introduction

Congratulations on your decision to implement your project using an MC100 system and the MC-1A programming language. The MC100 system is a comprehensive family of control systems that offers you enormous flexibility combined with ease of use and a wide range of expansion options intended to cover practically any conceivable application.

The MC100 System

The MC100 is a modular control system housed in a 19" rack or a case. The MC100 can be configured with up to 8 stepper and 8 servo motor axes. It can also be expanded using PLC components such as I/O modules, analogue I/O modules, fast counters or a temperature regulator.

Here is a brief description of only a few of the more important features of the MC100 System:

- The open PLC structure of the MC system allows simultaneous or independent operation of the stepper or servo motor axes. Control of entire machines and systems can be performed simultaneously.
- Various PLC software packages are available for the MC100, for example software for palettizers or rotary indexing tables.
- The MC100 can be supplied with 2 or 5 phase stepper motor power units in various power output ranges.
- DC motors, brushless synchronous motors or AC servomotors can be used as servo drive devices.

1.1 About This Documentation

The documentation presented here is logically organized into the following chapters:

Chapter 1 - Introduction Fehler! Ungültiger Eigenverweis auf Textmarke. **(starting on page 9)**

You are reading this chapter right now. It provides you with an overview of the MC100 System and the organization of this document. You can also find important tips in this chapter on how to best use the handbook to quickly locate the information you are looking for.

Chapter 2 - Programming with MC-1A (starting on page 12)

In the second chapter, we turn our attention to the basic concepts of MC-1A programming. This chapter will be of particular interest to you if you have not previously developed programs using MC-1A. This chapter provides a great deal of information about program structure, project work, how programs are executed in the control system and much more.

Chapter 3 - Command Overview Fehler! Verweisquelle konnte nicht gefunden werden. **-** Fehler! Verweisquelle konnte nicht gefunden werden. **(starting on page** Fehler! Textmarke nicht definiert.**)**

Probably the most important chapter in this document is the command reference. This is where you will find all the PLC instructions and macros available in the MC-1A language sorted alphabetically as well as by functional group. An example is given to illustrate the use of each instruction.

Chapter 4 - Special Applications Fehler! Verweisquelle konnte nicht gefunden werden. **-** Fehler! Verweisquelle konnte nicht gefunden werden. **(starting on page** Fehler! Textmarke nicht definiert.**)**

Programming special modules and the use of special functions are described in this chapter. This is where you can find information about the programming of displays and the integration of serial expansion modules or analogue I/O modules as well as a description of the extended measurement functions for the MC100 family.

Fehler! Verweisquelle konnte nicht gefunden werden. **-** Fehler! Verweisquelle konnte nicht gefunden werden. **(starting on page** Fehler! Textmarke nicht definiert.**)**

In this chapter, you will learn all you need to know about setting parameters in the MC100 system. The major topic is, of course, axis parameters, but you will also find information about configuration of the serial expansion modules or other system data in this chapter.

Fehler! Verweisquelle konnte nicht gefunden werden. **-** Fehler! Verweisquelle konnte nicht gefunden werden. **(starting on page** Fehler! Textmarke nicht definiert.**)**

What we have described here simply as "system data" provides you with a very useful overview of all system variables and flags. This data contains information about the current status of the system and attached equipment, including for example axes installed, displays, analogue I/O modules or counters. This chapter also includes the variables and flags that you will use again and again in your PLC programs, for example result variables.

Fehler! Verweisquelle konnte nicht gefunden werden. **-** Fehler! Verweisquelle konnte nicht gefunden werden. **(starting on page** Fehler! Textmarke nicht definiert.**)**

After presenting so much theoretical material, we now get down to business. We use examples to show you how you can achieve productive results as quickly as possible using the MC-1A language.

Appendices (starting on page Fehler! Textmarke nicht definiert.**)**

The appendices contain a variety of summary information, for example:

- ⇒ A truth table for logical operations
- ⇒ A table directory
- ⇒ A list of illustrations
- ⇒ An index of PC software required
- ⇒ A number of other items.

Structure and Numbering

For improved clarity, we have not assigned a chapter number to every heading. Only the most important sections are identified by chapter number. If you are looking for specific information on a particular topic, the best thing to do is use the Table of Contents at the beginning of the document.

Formatting in This Document

To enable you to get your bearings quickly in this document, special information is identified by the use of a particular format. If you familiarize yourself with the formatting, you will find it easier to work with this document.

Important notes

Information which is particularly important, such as the basic syntax of an MC-1A instruction, is always emphasized using **bold type** and a marking at the left border, for example:

LAD_VV Variable, Variable

Examples

Sample programs, which are frequently used in this document, are identified by a `different font` and by a wavy line at the left border, for example:

```
LAD_M      M_TEST1                 // If  M_TEST1
UND_M      M_TEST2                 // and M_TEST2 are turned on,
SPRINGJ    TESTZYKLUS              // then jump to label TESTZYKLUS
```

Tables

Where a large amount of information must be presented at one time, we use tables in this document. The appendix of this handbook contains an overview of all tables.

PC Software Documentation

The complete documentation for the VMC Workbench PC software is available as online documents or as Windows Help files on the VMC Workbench CD. A printed version of the VMC Workbench documentation is not available.

Chapter 2 Programming with MC-1A

If you have already written programs with PLC languages, the MC-1A language will appear familiar to you right from the start. This is due to the fact that its basic structure is very similar to popular PLC dialects, which are programmed as command lists. These languages always have the following basic characteristics:

- Only one instruction is allowed per line
- Conditional instruction execution is allowed
- The data types flag, variable, inputs, outputs and constants are available
- Conditional results are returned to the status registers

All of this applies to the MC-1A language as well. For this reason, we do not intend to concern ourselves here with PLC programming basics as such. Instead, we will concentrate specifically on the particular characteristics of the MC-1A system.

Why not IEC1131?

We gave a lot of thought to the question of whether to develop an IEC 1131-compatible programming language for the MC100 system. We finally decided against that option to avoid undermining the many advantages, such as easy integration of axes or additional equipment, very fast program execution and more. We are convinced that an experienced PLC programmer will achieve productive results significantly faster with the MC-1A language than with IEC 1131, and beginners will find it easier to get started because of the language's inherent logical structure.

2.1 Basic Conventions

Let us first of all define MC-1A language basics. These include, for example, how instructions must be written or how comments must be declared. If you have not yet worked with the MC-1A language, it is essential that you work through this section.

Language Definition

In the MC-1A language, programming is performed as a rule in the form of a command list in the following fixed format:

PLC instruction parameter(s)

The number of parameters depends on the individual instruction. There are also PLC instructions that do not require any parameter.

The data type of the parameter(s) is also dependent on the individual instruction. The data type expected is usually obvious from the instruction itself. Here are a few examples:

```
LAD_M 1                 // LAD_M means load flag. Thus the
                        // expected parameter is always a flag,
                        //in this case flag 1.
EIN_A 12                // EIN_A means turn on output. The
                        // parameter must therefore be an output, in
                        // this case output 12.
SPRING Schleife         // SPRING performs a program jump
                        // and thus always expects a label
                        // as a parameter.
```

Passing Parameter(s)

For most instructions, it is possible to directly enter a number for the corresponding resource, as in the example above, where "12" indicates digital output 12 or "1" means flag 1. The significance of the parameter is derived from the instruction.

Symbolic Names

Normally, however, you will be working with symbolic names for resources, since the MC-1A language supports the free assignment of names to variables, flags, constants as well as inputs and outputs. To do this, you assign the names you want at the beginning of your program or in a special definition file. This makes your programs much easier to read. Let us use the example above, this time using symbolic names:

```
DEF_M 1,M_START         // Assign the name "M_START" to flag 1
DEF_A 12,A_LAMPE        // Assign the name "A_LAMPE" to output 12

LAD_M M_START           // Load flag M_START (= flag 1)
EIN_A A_LAMPE           //Set output A_LAMPE (= output 12)
```

Symbolic Name Restrictions

Please observe the following restrictions on symbolic names:

- The symbolic name may not be longer that 23 characters.
- The name may only contain the letters A-Z as well as underscores, dashes and the numbers 0-9. No other characters, such as spaces, German umlaut characters or other special characters are allowed in symbolic names.
- Case is ignored in symbolic names.

Comments

Comments or notes about program flow make it easier to read the source code. You should be generous in including pertinent explanatory information. In the examples shown so far, you have seen that we have added comments after each instruction, always beginning with the "//" string. This is one method of inserting comments into the source code.

Mark the rest of the current line as comments

If you use either the "//" string or the ";" character (semicolon) in your source code, the rest of that line is marked as comments and has no affect on program execution.

instruction parameter(s) //comment
instruction parameter(s) ; comment

This is the normal way of including comments in the source code. You place a comment after each PLC instruction to explain what is happening at that point.

Longer Comments

If you wish to insert a longer comment, for example to explain a basic function or to mark tasks that do not need to be executed yet, you can designate a multi-line block as comments. To do this, you mark the beginning of the block with the "/*" string. Everything that follows this string is treated by the compiler as comments and is ignored during program execution. To close the comments, you use the "*/" string. Subsequent source code is processed by the compiler.

/* comments */

Example of the use of comments

```
/* Our comments begin here.  We can now write
several lines of explanation relating to the program.
Everything that appears in the comments is ignored
 by the compiler. */

Start:                          // Label
          LAD_M M_START         ;  check M_START flag
          SPRINGN START         /* back to beginning */
```

In the example shown above, you can see at a glance all the possible ways of including comments in your source code. Please note that if you start a comment with "/*", you must close that comment with "*/".

Comments are ignored when your PLC program is compiled, and they are not transferred to the controller. Comments thus do not occupy any memory space in the controller.

Labels

You use a label to define a particular place in your PLC program which you want to jump to from another place in the program. You may want to do this when you:

- want to create a program loop
- define the entry point of a subroutine or
- execute different program sequences depending on the result of a conditional statement.

Labels assign a name to a particular place in the program that you can use at any time in your PLC program. The following example will clarify this point:

Programming a loop

```
Start:                          // Defines the label "Start". The name of
                                // this line in your PLC program
                                // will now simply be "Start".
          LAD_E E_SteuerungEin  // We query the input E_SteuerungEin,
                                // which is connected to the key switch
                                // used to turn on the controller.
          SPRINGN Start         // If the input is inactive, we jump
                                // back to the line in the program where the
                                // label "Start" is located.
```

Programming a subroutine

```
Haupt:                          // Defines the label “Haupt” for the current
                                // program line.
          GEHUPR Achsen         // Call the subroutine "Achsen"
          SPRING Haupt          // Back to label "Haupt"

Achsen:                         // Defines the label "Achsen" for the current
          ...                   // line. This label is used to
          ...                   // call the subroutine.
          ...                   // subroutine execution
          UPREND                // end subroutine
```

Programming a conditional jump

```
          LAD_M M_Display       // Queries a flag to determine whether
                                // the display should be updated.
          SPRINGN Weiter        // If no,  jump to label "Weiter".
          ...                   // The source code for the display  goes here.
Weiter:                         // Defines the label "Weiter"
          ...                   // continue program execution
```

Label name restrictions

Please observe the following restrictions on label names:

- The label name may not be longer that 23 characters.
- The name may only contain the letters A-Z as well as underscores, dashes and the numbers 0-9. No other characters, such as spaces, German umlaut characters or other special characters are allowed in label names.
- Case is ignored in label names.

Macros

The MC-1A language supports the use of macros. This means that you can define frequently used routines in a macro. You then only need to call the macro. This creates clarity in your programs, but you should use this method only when routines are completely identical. Here is an example of a macro:

```
TOGGLE_A   MACRO Ausgang          // Start of the macro definition
           NLAD_A Ausgang         // Load inverted status of the output
           MOD_A Ausgang          // toggle output state
           ENDM                   // End of the macro definition

// Use the macro in the program

           TOGGLE_A 14            // toggle output 14
TOGGLE_A A_Blinklicht             // toggle output A_Blinklicht
```

You will find a summary of all macro instructions as well as further information about this topic in Chapter 3.1 - Instructions by Functional Group under "Macros" (Page 34).

Structure of the Source Code Project

In principle, your MC-1A project can be made up of any number of individual files. With the aid of the VMC Workbench development environment supplied, you can add new source files to your PLC project or remove files from the project at any time.

You normally use at least four different files in an MC-1A project for the following tasks:

- ⇒ PLC source code, contains the actual program
- ⇒ Definition file, contains symbolic definitions and constants
- ⇒ Macro file, contains macro definitions
- ⇒ PLC text file, contains text for the user interface

You can of course combine all of these functions into one file. However, to maintain better order and clarity in your project, we recommend using a separate file for each task.

In practice, you usually take this concept further. For bigger projects, a large number of individual files are normally used. Each file represents a particular piece of equipment or a special process. For example, it is common to create all initialization routines in a separate file. This also applies to axis management, fault handling, etc. This gives you a better overview of your project. Strict separation also creates an environment for modular programming, which enables quick adaptation of standard machines to particular tasks.

You can find further information about management of PLC projects in the online documentation of your development environment, VMC Workbench Studio.

Modular Programming

If you develop a well thought out design for your PLC project right from the start, you will be able to create a standard project easily and simply, which you will normally only have to modify in certain places for special machine variations. This concept is based on modular, carefully planned program design and of course on the conditional compilation capability of the MC-1A language.

Let us assume that you are producing a standard machine that is available in various models. You market a basic machine that is not equipped with a display. The next larger model of your machine is equipped with a display but not with a warning light and siren. Only the largest model has all of these options.

In planning your PLC program, you first write the program for the largest machine, which has all conceivable options. When doing so, ensure that all functions which are not available on the smaller models are logically segregated from the remaining machine functions. It makes no sense if you repeatedly access the display handler during execution. It is better to segregate the corresponding code.

As soon as the program for the largest model of your machine is complete, you then begin to systematically remove the options that are not available for the smaller models. This is most easily done with conditional compilation.

Definition of Constants for the Options

Declare a constant for each option that might possibly not be available on a smaller model of your machine. These constants will later be used to decide which source code will be used in your project.

```
          DEF_W 1,OptionDisplay   // 0 = no display, 1 = display installed
          DEF_W 0,OptionSirene    // 0 = no siren, 1 = siren installed
```

Use of Constants in the Program

In the next step, you create switches for conditional compilation at places that, for example, access the display or the siren:

```
          ...                     // We are starting in the middle of program
execution
          LAD_M M_Stoerung        // Has a fault occurred?

#IF OptionSirene EQ 1             // The instruction to activate the siren
          EIN_A A_Sirene          // is compiled with the source code only
#ENDIF                            // if the "Sirene" option is set

#IF OptionDisplay EQ 1            // We load the fault number into
          LAD_VA V_Text,V_Stoer   // our text  variable and show the corresponding
          GEHUPR StoerungAnz      // fault  on the display only if
#ENDIF                            // the "Display" option is set
```

Once you have included the appropriate conditional compilation switch in your program, from then on you will only need to change the values of the constants "OptionDisplay" and "OptionSirene" to create the project for the particular machine model.

Special Machines using Modular Programming

This type of programming is particularly powerful when you have to program particular modifications for special machines. You simply use your standard program and insert the special enhancements, always enclosed by switches, for conditional compilation. In this way, you always retain a current version of your source code, which you can adapt over time to almost any situation by using the appropriate switches.

You will find an overview of the topic "Conditional Compilation" in Chapter 3.1 - Instructions by Functional Group under "Compiler Directives" (Page 28).

2.2 Data Types

There are six different data types in all in the MC-1A system, which are explained in the following sections.

Variables

PLC variables are always 24 bits wide in the MC-1A system. This leads to the following limitations:

⇒ Largest possible value: 8388607

⇒ Smallest possible value: - 8388608

Only whole numbers can be stored in the MC-1A system. These are called "integers" in technical language. Decimal places, so-called "real numbers", are not supported. If you have to manipulate numbers with decimal places, you multiply the number by the factor 10 as many times as necessary until the result is a whole number.

Variables can be universally used in MC-1A. There is no basic limitation on what must actually be contained in a variable. Variables can thus contain a normal value, represent text or act as pointers to other data. A complex set of instructions is available to you for working with variables, which extends from simple load and compare operations right up to arithmetic calculations.

Flags

PLC flags can only store one bit of information. They can have only one of two possible states: "on" or "off".

The use of flags is recommended whenever one single item of status information is stored or must be passed to another part of the program. Flags have the following advantages:

⇒ Simple programming without constants or numerical values

⇒ Fast evaluation of logical states in the PLC program

⇒ Simplest logical operations between several conditions

Flags are often used to store the basic state of a sequence or to activate other functions within a sequence. Additional information can of course also be stored in a variable.

In this way, fault handling could be programmed such that the fault monitor writes the error code to a variable and also sets a flag to signal that a fault has occurred. In the main program, only the fault flag is continually queried instead of the fault variable being checked each time. Detailed checking of the error code is only performed if the flag has been set.

Outputs

Similar to the PLC flags, an output only contains the information "on" or "off". However, this data type always represents a digital PLC output that actually exists. A change of an output always changes the particular output immediately, independently of PLC program execution.

Inputs

An "input" data type variable represents a digital input that actually exists. The value is continually updated.

Constants

A constant defines a symbolic name as a number. In the PLC program, this symbolic name can then be used as a clear text name in place of the number with instructions that take numerical values.

The use of constants is particularly convenient if you use the same value in several different places in your program, for example to indicate the velocity of a positioning movement. Instead of having to replace the numerical value in various places in case of change, you only have to change the value in definition of the constant.

Labels

A label represents a particular location in your PLC program, or in other words a program address. Labels are normally only used in connection with jump instructions. You can, however, also pass the program address of a label to a variable to create dynamic cycle programming.

2.3 Cycles, Sequences and Tasks

The MC100 system is by nature a multi-tasking machine, so that sequences can be processed independently of one another. This capability is fully supported by the MC-1A programming language. It is, however, even more important to understand that the MC100 system does not strictly speaking operate as a cycle machine. In contrast to a cycle machine, the MC100 always operates in an asynchronous mode. All operating states, for example the states of inputs and outputs, are continually updated, and program execution in not bound to a fixed cycle.

How Does a Program Execute in MC-1A?

The most important basic principle that you must grasp when programming a system using MC-1A is that processing is not performed in a fixed cycle sequence. With MC-1A, you can define the manner in which PLC program will be processed.

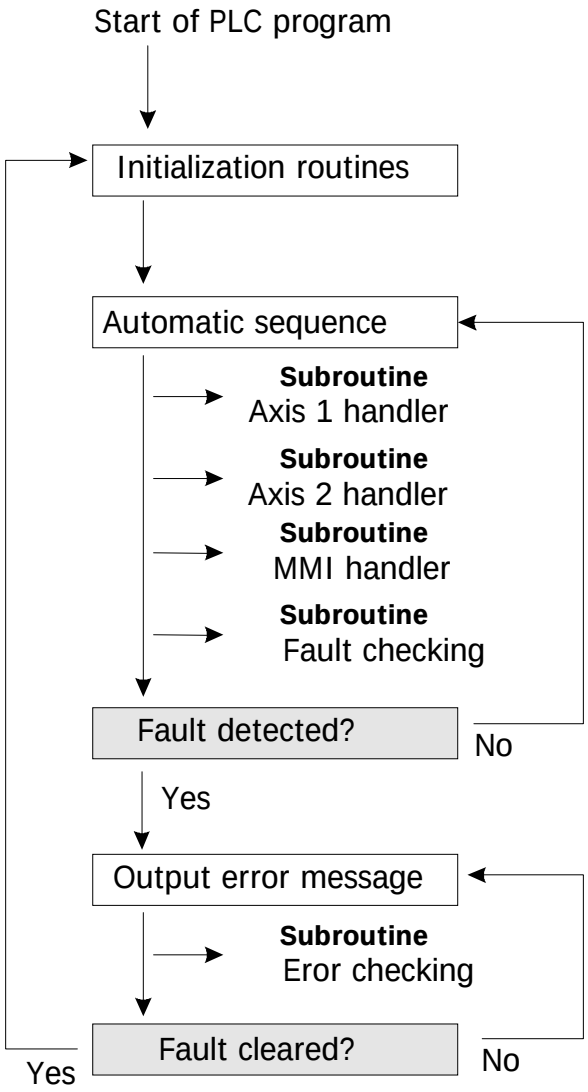

Figure 1 – Typical Flow of a PLC Program in MC-1A

As you can see in this example, MC-1A supports more than one sequence. You can nest your main program anyway you wish. You can create program loops, jump from one part of your program to another, call subroutines (which in turn can of course represent sequences and cycles) and much more.

The example above as an MC-1A Program

If we wanted to represent the example above using MC-1A, it might look as follows:

```
Start:                          //Beginning of program
          GEHUPR Willkommen     // Send power on message to display
                                // Function is called as subroutine

Init:                           // Initialization routine label
                                // Calls the subroutines that
                                // initialize the PLC program
          GEHUPR MemLoeschen    // Deletes internal data from the previous run
          GEHUPR Referenzfahrt  // Re-references all axes

Automatik:                      // Auto sequence label
          GEHUPRI Achse1Verwalt // Subroutine handler axis 1
GEHUPRI Achse2Verwalt           // Subroutine handler axis 2
          GEHUPRI StatusAnzeige // Subroutine to output status
                                // to the display
          GEHUPRI FehlerTest    // Check whether any faults have occurred
          LAD_M M_Stoerung      // Check whether the subprogram has
                                // reported a fault
          SPRINGN Automatik     // No fault, continue auto sequence

Stoerung:                       //  Fault handler label
          GEHUPRI FehlerMeldung // Send fault message to display
          GEHUPRI FehlerTest    // Check whether fault still exists
SPRINGJ Stoerung                // Yes, wait for corrective action
          SPRINGN Init          // No, fault cleared, re-initialize
```

You can see that a PLC program can well be simple and easy to understand. You may have noticed that all instructions are derived from the German language. Wherever possible, we have created instructions based on German words. The instruction GEHUPR, for example, is derived from the German phrase "GEHe UnterPRogramm", which means "go to subroutine". GEHUPRI stands for "GEHe UnterPRogramm Immer", which means "go to subroutine always", SPRINGJ stands for "SPRINGe wenn Ja" ("jump if yes"), etc.

Using Tasks

The MC-1A program flow concept becomes even more powerful when your program contains not only subroutines, but also tasks.

We wish to define a "task" as an independent, additional main program, which runs without a direct link to other "tasks" (in other words main programs). The term "parallel program" is often used to describe this concept.

The use of tasks gives you new, enhanced possibilities. If, for example, when handling axis movements you wait in your main program until the axis movement has been completed (is in position), without tasks you have two programming possibilities:

- You program as you would for a classical cycle machine using auxiliary flags and variables. The disadvantage is that you have to run through your entire cycle each time and must of course make the necessary provisions in your program.

- You program a wait loop that waits in the PLC program until the condition is met (in this case that the desired axis position has been reached). The disadvantage of course is that your program can do nothing else in the meantime. The problem is not only that there is more than one axis to handle on more complex machines. You are also unable to easily react to a fault condition, because your program is "hung up" in a wait loop.

Let a task do the work

All of this becomes easier if you move this type of critical or time-intensive function into tasks. Task 2, for example, could wait in a loop for any length of time until a condition is met. You can even move entire pieces of equipment, in other words component elements of your machine, into separate tasks. The main program is not affected and continues to execute normally.

Tasks can be easily programmed

In order to move sections of the program to tasks, you need only write a normal routine in the MC-1A language. To transform this routine into a task, you only need to insert the following instruction wherever you wish:

```
LAD_P2 Ueberwachung
```

Your program routine, which begins at the "Ueberwachung" label, would then run as a separate, independent task in your PLC program (in this example as Task 2). When programming tasks, the only thing you have to consider is that individual, independent tasks must not "get in each other's way". They must not use variables or flags at the same time that another task is using them.

Programming Cycles

Since MC-1A programs do not inherently run as a cycle machine, they have to, if desired, simulate the function of a cycle machine. This can, however, easily be done, if you simply declare a cycle variable and store the cycle counter in this variable:

```
          DEF_V 200,V_ZYKLUS      // Define variable 200 as a cycle variable

Automatik:                        // Label for cycle routine
          LAD_VA V_ZYKLUS,1       // Begin with the first cycle

Zyklus1:                          // Label  for the first  cycle
          VERGL_VA V_ZYKLUS,1     // When cycle 1 has been processed,
                                  // the cycle variable must be greater than 1
          NLAD_M M_GROESSER       // Check result of comparison (if  not greater,
                                  // the Bit Result  is set)
SPRINGN Zyklus2                   // Jump to cycle 3 if Bit  Result  off

          ...                     // Routine for cycle 1

          LAD_VA V_Zyklus,2       // Set cycle variable to next cycle

Zyklus2:                          // Label for second cycle
          VERGL_VA V_ZYKLUS,2     // When cycle 2 has been processed,
                                  // the cycle variable must be greater than 2
          NLAD_M M_GROESSER       // Check result of comparison (if not greater,
                                  // the Bit Result is set)
SPRINGN Zyklus3                   // Jump to cycle 3 if Bit Result off

          ...                     // Routine for cycle 1

          LAD_VA V_Zyklus,3       // Set cycle variable to next cycle
```

So you can see that even a "classical" cycle machine can very easily be implemented using the MC-1A language. All of this works even better if you follow a few tips which make your programming and the handling of cycle routines easier:

Tips and tricks on cycle programming

- Leave gaps in your cycle designations. When you begin your first draft of the routine, leave a good 20 free routines between each cycle. You would thus number them cycle 1, cycle 21, cycle 41, etc. This leaves scope for inserting the necessary instructions between the existing cycles easily and without complications if, as almost always happens, additional routines or equipment need to be added at a later time.
- Does not use flags when programming cycles. These 1-bit variables may seem to be suitable for this purpose. However, if you have to handle a large number of routines, this method very quickly becomes unwieldy.
- Have your cycle machine run in a subroutine or task. If you program the cycle routine to run independently of handlers for other equipment on your machine, additional monitoring or a simple man-machine interface can be integrated more easily and effectively.

Cycles using Dynamic Jumps

The MC-1A language allows you to use dynamic routine handling in addition to classic cycle programming. Instead of storing only a cycle counter in a cycle variable, you can directly store the jump destination there for the particular cycle:

```
          DEF_V 200, V_ZYKLUS     // Define variable 200 as cycle variable
          LAD_VL V_ZYKLUS,Zyklus1 // Pre-load the cycle variable with the label
                                  // of the first cycle routine

Automatik:                        // Label for auto loop
          GEHUPRI Display         // Call display handler subroutine
          GEHUPRI Achsen          // Call axis handler subroutine
          ...                     // Additional handler subroutines

// Now comes the most important step: instead of calling a general cycle
subroutine,
// we jump directly into the subroutine pointed to by the variable V_ZYKLUS


          GEHUPRV V_Zyklus        // Jump to the subroutine pointed to
                                  // by the variable V_ZYKLUS
          SPRING Automatik        // Continue with the auto loop

// The following is a typical cycle for dynamic programming:


Zyklus1:                          // Label for the first cycle
          ...                     // Program code for the flow of this cycle
...                               // No further checks are necessary at this
          ...                     // point, the subroutine is only called
          ...                     // if this cycle is actually supposed to
          ...                     // be performed
          LAD_VL V_ZYKLUS,Zyklus2 // Load the label of the next cycle
                                  // to the cycle variable
UPREND                            // End subroutine for cycle 1
```

As you can see from the example above, it takes much less effort to program a dynamic cycle machine as opposed to a "classical" version. You no longer have to check at the beginning of each routine whether that cycle is actually active. As soon as the cycle subroutine is called, you can be sure that it is actually supposed to be executed.

2.4 Our Result Buffer: the Bit Result (BES)

It is part of the basic philosophy of the MC-1A language that some instructions are only executed if the central result buffer the Bit Result, is turned on. This enables simple, effective and fast programming, because you do not necessarily need to perform a jump after testing a condition. Since execution of these instructions depends on the Bit Result, you can even perform complex testing of conditions without using a single jump instruction.

Understanding how the Bit Result works and how its affects program execution is extremely important for programming in MC-1A, so please read this section carefully.

What Affects the Bit Result?

Every test of a bit-oriented data type (flag, outputs and inputs) passes the status of the data tested to the Bit Result. If an inactive input is tested, the Bit Result is turned off as well. If the input was active, the Bit Result is on.

What Effect Does the Bit Result Have?

All instructions that depend on the Bit Result are executed only if the Bit Result is in the appropriate state. Turning on an output is only performed, for example, if at the time of execution the Bit Result is turned on. Some instructions, such as all jump instructions, are available in different variations to be able to react to the current Bit Result status.

Example of working with the Bit Result

```
// We assume for the moment that input 1 is not active

          LAD_E 1               // Load state of input 1 to the Bit Result.
                                // As the input is not active,
                                // the Bit Result is turned off.
          EIN_A 1               // This instruction is not executed, because
                                // the Bit Result is turned off.
          AUS_A 2               // This instruction is not executed either, as
                                // the Bit Result is still turned off.
          LAD_M M_EIN           // All load and compare instructions
                                // execute regardless on the Bit Result.  At this
                                // point we pass the state of the  "always on"
                                // Flag to the Bit Result.  This guarantees
                                // that the Bit Result is now
                                // turned on.
          AUS_A 1               // This instruction is now executed, because
                                // the Bit Result is turned on.
          SPRINGN Irgendwo      // Jump to label "Irgendwo",  if the
                                // Bit Result is turned off
          SPRINGJ Sonstwo       // Jump to label "Sonstwo", if the
                                // Bit Result is turned on
```

In the instruction summary, which begins in the next chapter, we indicate for each individual instruction what effect the instruction has on the Bit Result and whether execution of the instruction depends on the Bit Result.

Bit Result Shift Register

Using the Bit Result Shift Register, you have direct access to the last 16 Bit Results. Whenever an instruction directly changes the Bit Result, the contents of the Bit Result Shift Register are shifted to the left, and the new value is stored at the first position in the Bit Result Shift Register. You can use this function, for example, to nest various queries, achieving a similar effect to using levels of parentheses. You can find further information about the Bit Result Shift Register in Chapter **Fehler! Verweisquelle konnte nicht gefunden werden.** - **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

Chapter 3 Command Overview

In this chapter, we present a complete summary of all MC-1A instructions and pre-defined macros available in the development environment. For better clarity and to enable easier look-up of any particular instruction, the reference is divided into two parts:

Section 3.1 - Instructions by Functional Group (starting on page 28)

In this section, you will find a summary of all MC-1A instructions sorted by functional group. Each instruction is accompanied by the corresponding syntax and a brief description.

Section 3.2 - Alphabetical Summary of Instructions (starting on page 37)

You can find a complete alphabetical summary of all MC-1A PLC instructions here. The description of each instruction is supplemented by a program example. You can see at a glance how each instruction is used.

Designation of Parameters

The parameter(s) for each instruction is (are) also indicated using a small symbol, for example:

LAD_MI First Flag V

This information defines the data type (ref. Chapter 2.2 - Data Type starting on page 18) that the instruction expects. The following data types are allowed:

- V - Variable
- M - Flag
- E - Input
- A - Output
- K - Constant or Label

Interaction with the Bit Result Buffer (BES)

At the beginning of the description for each instruction, you will see a small box:

Group	Depends on BES	Changes BES
Definitions	**No**	**No**

The meaning of this information is as follows:

- "Group" associates the instruction with a functional group. You can find a summary of all instructions sorted by functional group in Chapter 3.1 starting on page 28.
- "Depends on BES" immediately tells you whether this instruction is only executed if the Bit Result Buffer (BES) has a particular value. A "no" in this position means that the instruction is always executed. "Yes", on the other hand, means that the instruction is only executed if the "BES" state is "on". In a few special cases, "if off" can appear here. This means that the instruction is only executed if the BES is turned off.
- "Changes BES" indicates whether the BES state is changed by this instruction.

Important Note

In the instruction summary, the assumption is made that you are already familiar with the basics of MC-1A programming. If appropriate, please refer to the explanations in Chapter 2 - Programming with MC-1A starting on page 12.

3.1 Instructions by Functional Group

Definition Instructions

Instruction	Meaning	Page
DEF_A OutputNumber K, SymbolName K	DEF_A assigns a symbolic name to an output.	42
DEF_E InputNo. K, SymbolName K	DEF_E assigns a symbolic name to an input.	42
DEF_M FlagNumber K, SymbolName K	DEF_M assigns a symbolic name to a flag.	43
DEF_V VariableNo. K, SymbolName K	DEF_V assigns a symbolic name to a variable.	43
DEF_W Value K, SymbolName K	DEF_W assigns a symbolic name to a constant.	44

Table 1 – Definition Instructions

Compiler Directives

Instruction	Meaning	Page
#ELSE	#ELSE is used for testing conditions in connection with conditional compiler runs. Prior to the instruction, a condition is tested with the EQ instruction. If the first dependent directive is not fulfilled, the second dependent directive is executed.	98
#ENDIF	#ENDIF ends conditional compilation.	98
#IF TestCondition K #IF Value1 K EQ Value2 K	#IF is used for testing conditions in connection with conditional compiler runs. Prior to the instruction, a condition is tested with the EQ instruction. If the first dependent condition is fulfilled, the first directive executed. If the first dependent directive is not fulfilled, the second dependent directive (if any) is executed.	99
#INCLUDE SourceCodeFileName K	#INCLUDE inserts an additional source code file at the current location.	100

Table 2 – Compiler Directives

Flag Instructions

Instruction	Meaning	Page
AUS_M Flag M	The instruction AUS_M clears the specified flag.	40
AUS_MI Pointer V	The instruction AUS_MI clears the flag referenced by the pointer.	40
EIN_M Flag M	EIN_M sets the specified flag.	47
EIN_MI Pointer V	EIN_MI sets the flag referenced by the specified pointer.	48
LAD_M Flag M	LAD_M loads the state of the specified flag into the Bit Result Buffer (BES)	58
LAD_MI Pointer V	LAD_MI loads the state of the flag referenced by the pointer into the Bit Result Buffer (BES).	59
MOD_M Flag M	MOD_M loads the state of the Bit Result Buffer (BES) into the specified flag.	68
MOD_MI Pointer V	MOD_MI loads the state of the Bit Result Buffer (BES) into the flag referenced by the pointer.	69
NLAD_M Flag M	NLAD_M loads the inverted state of the flag into the Bit Result Buffer (BES).	72
NODER_M Flag M	NODER_M performs a logical OR operation between the state of the Bit Result Buffer (BES) and the inverted state of the specified flag. The result of this operation is placed into the Bit Result.	74
NUND_M Flag M	NUND_A performs a logical AND operation between the state of the Bit Result Buffer (BES) and the inverted state of the specified flag. The result of this operation is placed into the Bit Result.	75
ODER_M Flag M	ODER_M performs a logical OR operation between the states of the Bit Result Buffer (BES) and the specified flag. The result of this operation is placed into the Bit Result.	77
UND_M Flag M	UND_M performs a logical AND operation between the current state of the Bit Result Buffer (BES) and the state of the specified flag. The result is placed into the Bit Result Buffer (BES).	88
XODER_M Flag M	XODER_M performs a logical exclusive-or (XOR) operation between the state of the Bit Result Buffer (BES) and the state of the specified flag. The result of this operation is put into the Bit Result Buffer (BES).	98

Table 3 – Flag Instructions

Input / Output Instructions

Instruction	Meaning	Page
AUS_A Output A	The instruction AUS_A turns off the specified output.	39
AUS_AI Pointer V	The instruction AUS_AI turns off the output specified by the pointer.	39
EIN_A Output A	EIN_A turns on the specified output.	46
EIN_AI Pointer V	EIN_AI turns on the output referenced by the specified pointer.	47
LAD_A Output A	LAD_A loads the state of the specified output into the Bit Result Buffer (BES).	52
LAD_AI Pointer V	LAD_AI loads the state of the output referenced by the pointer into the Bit Result Buffer (BES).	53
LAD_E Input E	LAD_E loads the state of the specified input into the Bit Result Buffer (BES).	56
LAD_EI Pointer V	LAD_EI loads the state of the input referenced by the pointer into the Bit Result Buffer (BES).	56
MOD_A Output A	MOD_A sends the state of the Bit Result Buffer (BES) to the specified output.	67
MOD_AI Pointer V	MOD_AI sends the state of the Bit Result Buffer (BES) to the output referenced by the pointer.	68
NLAD_E Input E	NLAD_E loads the inverted state of the input into the Bit Result Buffer (BES).	72
NLAD_A Output A	NLAD_A loads the inverted state of the output into the Bit Result Buffer (BES).	71
NUND_A Output A	NUND_A performs a logical AND operation between the state of the Bit Result Buffer (BES) and the inverted state of the specified output. The result of this operation is placed into the Bit Result.	74
NUND_E Input E	NUND_E performs a logical AND operation between the state of the Bit Result Buffer (BES) and the inverted state of the specified input. The result of this operation is placed into the Bit Result.	75
UND_E Input E	UND_E performs a logical AND operation between the current state of the Bit Result Buffer (BES) and the state of the specified input. The result of this operation is placed into the Bit Result Buffer (BES).	87
UND_A Output A	UND_A performs a logical AND operation between the current state of the Bit Result Buffer (BES) and the state of the specified output. The result of this operation is placed into the Bit Result Buffer (BES).	87
NODER_A Output A	NODER_A performs a logical OR operation between the state of the Bit Result Buffer (BES) and the inverted state of the specified output. The result of this operation is placed into the Bit Result.	73
NODER_E Input E	NODER_E performs a logical OR operation between the state of the Bit Result Buffer (BES) and the inverted state of the specified input. The result of this operation is placed into the Bit Result.	73
ODER_A Output A	ODER_A performs a logical OR operation between the states of the Bit Result Buffer (BES) and the specified output. The result of this operation is placed into the Bit Result.	76

Instruction	Meaning	Page
ODER_E Input E	ODER_E performs a logical OR operation between the states of the Bit Result Buffer (BES) and the specified input. The result of this operation is placed into the Bit Result.	76
XODER_A Output A	XODER_A performs a logical exclusive-or (XOR) operation between the state of the Bit Result Buffer (BES) and the state of the specified output. The result of this operation is put into the Bit Result Buffer (BES).	97
XODER_E Input E	XODER_E performs a logical exclusive-or (XOR) operation between the state of the Bit Result Buffer (BES) and the state of the specified input. The result of this operation is put into the Bit Result Buffer (BES).	97

Table 4 – Input / Output Instructions

Variable Instructions

Instruction	Meaning	Page
ADD_IV Pointer V, Variable V	The instruction ADD_IV adds the contents of the variable referenced by the pointer to the contents of the variable in the second parameter. The result of the addition operation is stored in VARERG.	37
ADD_VA Variable V, Value K	The instruction ADD_VA adds the specified constant value to the contents of the variable. The result of the addition operation is stored in the VARERG result variable.	37
ADD_VI Variable V, Pointer V	The instruction ADD_VI adds the contents of the variable specified to the contents of the variable referenced by the pointer. The result of the addition operation is stored in the VARERG result variable.	38
ADD_VV Variable1 V, Variable2 V	The instruction ADD_VV adds the contents of the two variables specified. The result of the addition operation is stored in the VARERG result variable.	38
DEC_V Variable V, Value K	The instruction DEC_V decrements the contents of the variable by the specified value. The VARERG result variable is not changed.	41
DIV_IV Pointer V, Variable V	DIV_IV divides the contents of the variable which the pointer references by the contents of the second variable. The result of the division operation is stored in VARERG and DIVREST.	44
DIV_VA Variable V, Value K	DIV_VA divides the contents of the variable by the value specified. The integer value result of the division operation is stored in the result variable VARERG, and the remainder is stored in DIVREST.	45
DIV_VI Variable V, Pointer V	DIV_VI divides the contents of the specified variable by the contents of the variable referenced by the pointer. The result of the division operation is stored in VARERG and DIVREST.	45
DIV_VV Variable1 V, Variable2 V	DIV_VV divides the contents of Variable1 by the contents of Variable2. The integer value result of the division operation is stored in the result variable VARERG, and the remainder is stored in DIVREST.	46
LAD_IV Pointer V, Variable V	LAD_IV loads the contents of the specified variable into the variable referenced by the pointer.	57

Instruction	Meaning	Page
LAD_VA Variable V, Value K	LAD_VA loads the specified value into the variable.	63
LAD_VI Variable V, Pointer V	LAD_VI loads the value of the variable referenced by the pointer into the specified variable.	63
LAD_VV Variable1 V, Variable2 V	LAD_VV loads the contents of variable2 into variable1.	66
MUL_IV Pointer V, Variable V	MUL_IV multiplies the contents of the variable referenced by the pointer by the contents of the specified variable. The result of the multiplication operation is placed in the result variable VARERG, and any multiplication overflow (> 32 Bit) is stored in the result variable MUL_REST.	69
MUL_VA Variable V, Value k	MUL_VA multiplies the contents of the variable by the specified value. The result of the multiplication operation is placed in the result variable VARERG, and any multiplication overflow (> 32 Bit) is stored in the result variable MUL_REST.	70
MUL_VI Variable V, Pointer V	MUL_VI multiplies the contents of the variable referenced by the pointer by the contents of the specified variable. The result of the multiplication operation is placed in the result variable VARERG, and any multiplication overflow (> 32 Bit) is stored in the result variable MUL_REST.	70
MUL_VV Variable1 V, Variable2 V	MUL_VV multiplies the contents of variable1 by variable2. The result of the multiplication operation is placed in the result variable VARERG, and any multiplication overflow (> 32 Bit) is stored in the result variable MUL_REST.	71
INC_V Variable V, Value K	INC_V increments the contents of the variable by the specified value. The result variable VARERG is not changed	52
ODER_IV Pointer V, Variable V	ODER_IV performs a binary OR operation between the contents of the variable referenced by the pointer and the specified variable. The result of this operation is stored in the result variable VARERG.	77
ODER_VI Variable V, Pointer V	ODER_VI performs a binary OR operation between the contents of the variable referenced by the pointer and the specified variable. The result of this operation is stored in the result variable VARERG.	78
ODER_VA Variable V, Value k	ODER_VA performs a binary OR operation between the contents of the variable and the specified value. The result of this operation is stored in the result variable VARERG.	78
ODER_VV Variable1 V, Variable2 V	ODER_VV performs a binary OR operation between the two specified variables. The result of this operation is stored in the result variable VARERG.	79
SRL_VA Variable V, Value K	SRL_VA shifts the contents of the variable to the right by the specified value. The result is stored in the result variable VARERG.	83
SRL_VV Variable1 V, Variable2 V	SRL_VV shifts the contents of variable1 to the right by the contents of variable2. The result is stored in the result variable VARERG.	83
SLL_VA Variable V, Value K	SLL_VA left shifts the contents of the variable specified as the first parameter by the specified value. The result is stored in the result variable VARERG.	79
SLL_VV Variable1 V, Variable2 V	SLL_VV shifts the contents of variable1 to the left by the contents of variable2. The result is stored in the result variable VARERG.	80

Instruction	Meaning	Page
SUB_VA Variable V, Value K	SUB_VA subtracts the specified value from the contents of the variable. The result of the subtraction operation is stored in VARERG.	84
SUB_VV Variable1 V, Variable2 V	SUB_VV subtracts the contents of variable2 from the contents of variable1. The result of the subtraction operation is stored in VARERG.	85
SUB_VI Variable V, Pointer V	SUB_VI subtracts the contents of the variable referenced by the pointer from the contents of the specified variable. The result of the subtraction operation is stored in VARERG.	85
SUB_IV Pointer V, Variable V	SUB_IV subtracts the contents of the specified variable from the contents of the variable referenced by the pointer. The result of the subtraction operation is stored in VARERG.	84
UND_IV Pointer V, Variable V	UND_IV performs a binary AND operation between the contents of the variable referenced by the pointer and the contents of the specified variable. The result is stored in the result variable VARERG.	88
UND_VA Variable V, Value K	UND_VA performs a binary AND operation between the contents of the variable and the specified value. The result is stored in the result variable VARERG.	89
UND_VI Variable V, Pointer V	UND_VI performs a binary AND operation between the contents of the specified variable and the variable referenced by the pointer. The result is stored in the result variable VARERG.	89
UND_VA Variable1 V, Variable2 V	UND_VV performs a binary AND operation between the contents of the two specified variables. The result is stored in the result variable VARERG.	90

Table 5 – Variable Instructions

Variable Comparison Instructions

Instruction	Meaning	Page
VERG_IV Pointer V, Variable V	VERG_IV compares the contents of the variable specified by the pointer with the contents of the specified variable. The result is stored in the variable comparison flags.	92
VERG_VA Variable V, Value K	VERG_VA compares the contents of the specified variable with the specified value. The result is stored in the variable comparison flags.	93
VERG_VI Variable V, Pointer V	VERG_VI compares the contents of the specified variable with the contents of the variable referenced by the pointer. The result is stored in the variable comparison flags.	94
VERG_VV Variable1 V, Variable2 V	VERG_VV compares the contents of the two specified variables. The result is stored in the variable comparison flags:	95

Table 6 – Variable Comparison Instructions

Data Conversion Instructions

Instruction	Meaning	Page
LAD_MV First Flag M, Variable V	LAD_MV loads the contents of the variable into 32 flags. The transfer occurs bitwise. The first bit of the variable is loaded to the first flag, the second bit to the second flag, etc.	60
LAD_VM Variable V, First Flag M	LAD_VM loads the state of 32 flags beginning with the specified first flag into a variable. Loading takes place bitwise. The first specified flag is loaded into the first bit of the variable, the second flag into the second bit, etc.	65

Table 7 – Data Conversion Instructions

Macros

Instruction	Meaning	Page
ENDM	ENDM closes a macro definition. If the instruction ENDM is not preceded by a macro directive, a compiler error occurs.	48
EXITM	EXITM terminates the macro definition depending on a conditional compiler run. Use EXITM to abort a macro depending on symbolic constants.	49
Name K MACRO [Par 1 K, [Par 2 K, [...]]]	MACRO defines the start of a macro definition. Within the macro definition block, you then assign the name of the macro, any parameters and the associated PLC program code. The square brackets in the syntax indicate that you can decide for yourself how many parameters you want to pass to a macro instruction.	66

Table 8 – Macro Instructions

Program Flow Control Instructions

Instruction	Meaning	Page
GEHUPRI Label K	GEHUPRI calls a subroutine starting at the specified label.	49
GEHUPRJ Label K	GEHUPRJ calls a subroutine starting at the specified label. The call is only performed if the Bit Result Buffer (BES) is turned on at the time the instruction is executed.	50
GEHUPRN Label K	GEHUPRJ calls a subroutine starting at the specified label. The call is only performed if the Bit Result Buffer (BES) is turned off at the time the instruction is executed.	50
GEHUPRV DestinationAddress V	GEHUPRV calls a subroutine at the start address contained in the specified variable. Use this instruction for dynamic cycle programming.	51
LAD_P1 Label K	LAD_P1 loads the specified label into the program counter of the first parallel program. The parallel program is started immediately at the specified label.	61
LAD_P2 Label K	LAD_P2 loads the specified label into the program counter of the second parallel program. The parallel program is started immediately at the specified label.	61
LAD_P3 Label K	LAD_P3 loads the specified label into the program counter of the third parallel program. The parallel program is started immediately at the specified label.	62
LAD_P4 Label K	LAD_P4 loads the specified label into the program counter of the fourth parallel program. The parallel program is started immediately at the specified label.	62
LAD_VL Variable V, Label K	LAD_VL loads the program address of the label into the specified variable. The instruction LAD_VL is required to jump to a subroutine using GEHUPRV.	64
SPRING Label K	SPRING continues program flow at the specified label.	81
SPRINGJ Label K	SPRINGJ continues program flow at the specified label if the Bit Result Buffer (BES) is currently turned on.	82
SPRINGN Label K	SPRINGN continues program flow at the specified label if the Bit Result Buffer (BES) is currently turned off.	82
UPREND	UPREND ends a subroutine. The program continues from the point where the subroutine was called.	90
UPRENDJ	UPRENDJ ends a subroutine. The program continues from the point where the subroutine was called.	91
UPRENDN	UPRENDN ends a subroutine. The program continues from the point where the subroutine was called.	91
WART_A...WART_E	WART_A can be used to program a simple loop. The loop continues to run until the Bit Result Buffer (BES) is turned on.	96

Table 9 – Program Flow Control Instructions

Space for Your Notes

3.2 Alphabetical Summary of Instructions

ADD_IV

Group	Depends on BES	Changes BES
Variable Instructions	✸ **Yes**	■ **No**

ADD_IV Pointer V, Variable V

The instruction ADD_IV adds the contents of the variable referenced by the pointer to the contents of the variable in the second parameter. The result of the addition operation is stored in VARERG.

Operation

(VARERG) ← (Pointer → Variable) + (Variable)

Example

```
ADD_IV     V_ZEIGER,V_TEST         // The contents of the variable referenced by the
                                   // pointer V_ZEIGER, is added to  variable
V_TEST.
                                   // The result  is stored in VARERG.
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

ADD_VA

ADD_VA Variable V, Value K

Group	Depends on BES	Changes BES
Variable Instructions	✸ **Yes**	■ **Nein**

The instruction ADD_VA adds the specified constant value to the contents of the variable. The result of the addition operation is stored in the VARERG result variable.

Operation

(VARERG) ← (Variable) + Value

Example

```
DEF_W      10,ZAHL                 // Defines ZAHL as the value 100
ADD_VA     V_TEST,ZAHL             // Adds the variable V_TEST to the value ZAHL.
                                   // The result is stored in VARERG
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

ADD_VI

ADD_VI Variable V, Pointer V

Group	Depends on BES	Changes BES
Variable Instructions	✸ **Yes**	■ **No**

The instruction ADD_VI adds the contents of the variable specified to the contents of the variable referenced by the pointer. The result of the addition operation is stored in the VARERG result variable.

Operation

(VARERG) ← (Variable) + (Pointer → Variable)

Example

```
ADD_VI     V_TEST,V_ZEIGER        // The contents of the variable referenced by the
                                  // pointer V_ZEIGER, is added to variable
V_TEST.
                                  // The result is stored in VARERG
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

ADD_VV

Group[	Depends on	Changes BES
Variable Instructions	✸ **Yes**	■ **No**

ADD_VV Variable1 V, Variable2 V

The instruction ADD_VV adds the contents of the two variables specified. The result of the addition operation is stored in the VARERG result variable.

Operation

(VARERG) ← (Variable1) + (Variable2)

Example

```
LAD_VA     V_ZAHL1,10             // The value 10 is loaded into
                                  // the variable V_ZAHL1.
LAD_VA     V_ZAHL2,20             // The value 20 is loaded into
                                  // the variable V_ZAHL2.
ADD_VV     V_ZAHL1,V_ZAHL2        // The variables V_ZAHL1 and V_ZAHL2
                                  // are
                                  // added and the result is  stored
                                  // in VARERG.
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

AUS_A

AUS_A Output A

Group	Depends on BES	Changes BES
I/O instructions	✸ **Yes**	■ **No**

The instruction AUS_A turns off the specified output.

Operation

(Output) ← OFF

Example

```
AUS_A      A_TEST                  // Turns the output A_TEST off
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

I/O Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

AUS_AI

AUS_AI Pointer V

Group	Depends on BES	Changes BES
I/O Instructions	✸ **Yes**	■ **No**

The instruction AUS_AI turns off the output specified by the pointer.

Operation

(Pointer → Output) ← OFF

Example

```
LAD_VA     V_TEST,320              // Loads the value 320 into variable V_TEST
AUS_AI     V_TEST                  // Turns off  output 320
INC_VA     V_TEST,1                // Increments the value of the variable by 1
AUS_AI     V_TEST                  // Turns off output 321
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

I/O Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

AUS_M

AUS_M Flag M

Group	Depends on BES	Changes BES
Flag Instructions	✸ **Yes**	■ **No**

The instruction AUS_M clears the specified flag.

Operation

(Flag) ← OFF

Example

```
AUS_M     M_TEST                  // The flag M_TEST is cleared
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Flag Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

AUS_MI

AUS_MI Pointer V

Group	Depends on BES	Changes BES
Flag Instructions	✸ **Yes**	■ **No**

The instruction AUS_MI clears the flag referenced by the pointer.

Operation

(Pointer → flag) ← OFF

Example

```
LAD_VA    V_TEST,320              // Loads the value 320 into the variable V_TEST
AUS_MI    V_TEST                  // Clears the flag 320
```

Note

This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Flag Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

DEC_V

Group	Depends on BES	Changes BES

Variable	✸ Yes	■ No

DEC_V Variable V, Value K

The instruction DEC_V decrements the contents of the variable by the specified value. The VARERG result variable is not changed.

After variable1 is decremented, the system automatically performs a comparison with "0" and sets the variable comparison flag accordingly:

- M_GLEICH is set if variable1 contains the value "0" after the operation
- M_GROESSER is set if variable1 is greater than "0" after the operation
- M_KLEINER is set if variable1 is smaller than "0" after the operation

Operation

```
(Variable1)  ← (Variable) - Value
(M_GLEICH)   ← (Variable = 0)
(M_GROESSER) ← (Variable > 0)
(M_KLEINER)  ← (Variable < 0)
```

Example

```
//In this example, we have a variable field, which is written, for example, by the
PC,
// and we set the lower 16 bits. We use the instruction DEC_V to initialize the
pointer
// variable to the next entry in the table during each pass through
// the loop.

DEF_W      4000,TABELLE_ENDE       // Beginning of variable field
LAD_VA     V_ZEIGER,TABELLE_ENDE   // Initialize pointer variable
LAD_VA     V_MASKE,65535           // equals FFFFh

LOOP:
ODER_VI    V_MASKE,V_ZEIGER        // perform logical operation
LAD_IV     V_ZEIGER,VARERG         // place result of operation back into table
DEC_V      V_ZEIGER,1              // table pointer to preceding value
SPRING     LOOP                    // back to loop
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

DEF_A

DEF_A OutputNumber K, SymbolName K

Group	Depends on BES	Changes BES
Definitions	**No**	**No**

DEF_A assigns a symbolic name to an output.

Example

```
DEF_A      10,A_TEST              // Assigns the name A_TEST to output 10.
```

The symbolic name of the output can subsequently be used in the program.

Notes

- This instruction does not occupy any additional memory space in the controller. The size of the PLC program does not depend on whether you work using symbolic names or input??? output? Eingangsnummern numbers.
- You can place definitions both in a global definition file and within the MC-1A source code. Please note, however, that definitions within the source code are only valid for that particular source code file, not for the entire project.
- **See also**

Definition Instructions (Page 28)

DEF_E

DEF_E InputNo. K, SymbolName K

Group	Depends on BES	Changes BES
Definitions	**No**	**No**

DEF_E assigns a symbolic name to an input.

Example

```
DEF_E      10,E_TEST              // Assigns the name E_TEST to input 10
```

The symbolic name of the input can subsequently be used in the program.

Note

- This instruction does not occupy any additional memory space in the controller. The size of the PLC program does not depend on whether you work using symbolic names or input numbers.
- You can place definitions both in a global definition file and within the MC-1A source code. Please note, however, that definitions within the source code are only valid for that particular source code file, not for the entire project.

See also

Definition Instructions (Page 28)

DEF_M

Group	Depends on BES	Changes BES
Flag Instructions	**No**	**No**

DEF_M FlagNumber K, SymbolName K

DEF_M assigns a symbolic name to a flag.

Example

```
DEF_M      10,M_TEST              // Assigns the name M_TEST to flag 10
```

The symbolic name of the flag can subsequently be used in the program.

Note

- This instruction does not occupy any additional memory space in the controller. The size of the PLC program does not depend on whether you work using symbolic names or input??? flag? Eingangsnummern numbers.
- You can place definitions both in a global definition file and within the MC-1A source code. Please note, however, that definitions within the source code are only valid for that particular source code file, not for the entire project.

See also

Definition Instructions (Page 28)

DEF_V

Group	Depends on BES	Changes BES
Definitions	**No**	**No**

DEF_V VariableNo. K, SymbolName K

DEF_V assigns a symbolic name to a variable.

Example

```
DEF_V      10,V_TEST              // Assigns the name V_TEST to variable 10
```

The symbolic name of the variable can subsequently be used in the program.

Note

- This instruction does not occupy any additional memory space in the controller. The size of the PLC program does not depend on whether you work using symbolic names or input??? variable? Eingangsnummern numbers.
- You can place definitions both in a global definition file and within the MC-1A source code. Please note, however, that definitions within the source code are only valid for that particular source code file, not for the entire project.

See also

Definition Instructions (Page 28)

DEF_W

DEF_W Value K, SymbolName K

Group	Depends on BES	Changes BES
Definitions	**No**	**No**

DEF_W assigns a symbolic name to a constant.

Example

```
DEF_W     10,TEST                 // Assigns the name TEST to constant 10
```

The symbolic name of the constant can subsequently be used in the program.

Note

- This instruction does not occupy any additional memory space in the controller. The size of the PLC program does not depend on whether you work using symbolic names or input??? constant? **Eingang**snummern numbers.
- You can place definitions both in a global definition file and within the MC-1A source code. Please note, however, that definitions within the source code are only valid for that particular source code file, not for the entire project.

See also

Definition Instructions (Page 28)

DIV_IV

Group	Depends on BES	Changes BES
Variable Instructions	**Yes**	**No**

DIV_IV Pointer V, Variable V

DIV_IV divides the contents of the variable which the pointer references by the contents of the second variable. The result of the division operation is stored in VARERG and DIVREST.

Operation

```
(VARERG)  ← (Pointer → Variable) / (Variable)
(DIVREST) ← (Pointer → Variable) % (Variable)
```

Example

```
DIV_IV    V_ZEIGER,V_TEST         // The contents of the variable, which V_ZEIGER
                                  // points to, is divided by the contents of the
variable
                                  // V_TEST. The result  is stored in VARERG
                                  // and DIVREST
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

DIV_VA

Group	Depends on	Changes BES
Variable Instructions	✸ **Yes**	■ **No**

DIV_VA Variable V, Value K

DIV_VA divides the contents of the variable by the value specified. The integer value result of the division operation is stored in the result variable VARERG, and the remainder is stored in DIVREST.

Operation

```
(VARERG)  ← (Variable) / Value
(DIVREST) ← (Variable) % Value
```

Example

```
DEF_W     10,ZAHL              // loads the value 10 into ZAHL
DIV_VA    V_TEST,ZAHL          // divides the variable by the value ZAHL
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

DIV_VI

Group	Depends on BES	Changes BES
Variable Instructions	✸ **Yes**	■ **No**

DIV_VI Variable V, Pointer V

DIV_VI divides the contents of the specified variable by the contents of the variable referenced by the pointer. The result of the division operation is stored in VARERG and DIVREST.

Operation

```
(VARERG)  ← (Variable) / (Pointer → Variable)
(DIVREST) ← (Variable) % (Pointer→ Variable)
```

Example

```
DIV_VI    V_TEST,V_ZEIGER      //The contents of the variable V_TEST are divided
                               // by the contents of the variable referenced by
V_ZEIGER.
                               // The result is stored in VARERG
                               // and DIVREST.
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

DIV_VV

		Group	Depends on	Changes BES
Variable Instructions	✸ **Yes**	■ **No**		

DIV_VV Variable1 V, Variable2 V

DIV_VV divides the contents of Variable1 by the contents of Variable2. The integer value result of the division operation is stored in the result variable VARERG, and the remainder is stored in DIVREST.

Operation

```
(VARERG)  ← (Variable1) / (Variable2)
(DIVREST) ← (Variable1) % (Variable2)
```

Example

```
LAD_VA     V_ZAHL1,24             // The value 24 is loaded into V_ZAHL1
LAD_VA     V_ZAHL2,10             // The value 10 is loaded into V_ZAHL2
DIV_VV     V_ZAHL1,V_ZAHL2        // Divides V_ZAHL1 by V_ZAHL2
                                  // The result (2) is stored in VARERG, the
remainder (4)
                                  // is stored in DIVREST
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

EIN_A

Group	Depends on BES	Changes BES
I/O Instructions	✸ **Yes**	■ **No**

EIN_A Output A

EIN_A turns on the specified output.

Operation

```
(Output) ← On
```

Example

```
EIN_A      A_TEST                 // Turns on the output A_TEST
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

I/O Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

EIN_AI

EIN_AI Pointer V

Group	Depends on BES	Changes BES
I/O Instructions	✸ **Yes**	■ **No**

EIN_AI turns on the output referenced by the specified pointer.

Operation

```
(Pointer → Output) ← ON
```

Example

```
LAD_VA     V_TEST,320            // Loads the value320 into the variable V_TEST
EIN_AI     V_TEST                // turns on output 320
INC_V      V_TEST,1              // increments the value of the variable by 1
EIN_AI     V_TEST                // turns on output 321
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.
- You use EIN_AI to set an output when the output's number has not yet been defined during creation of the program.

See also

I/O Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

EIN_M

EIN_M Flag M

Group	Depends on BES	Changes BES
Flag Instructions	✸ **Yes**	■ **No**

EIN_M sets the specified flag.

Operation

```
(Flag) ← ON
```

Example

```
EIN_M      M_TEST                // Sets the flag M_TEST
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Flag Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

EIN_MI

EIN_MI Pointer V

Group	Depends on BES	Changes BES
Flag Instructions	✸ **Yes**	■ **No**

EIN_MI sets the flag referenced by the specified pointer.

Operation

(Pointer → Flag) ← ON

Example

```
LAD_VA     V_TEST,320            // Loads the value 320 into the variable V_TEST
EIN_MI     V_TEST                // sets the flag 320
INC_V      V_TEST,1              // increments the value of the variable by 1
EIN_MI     V_TEST                // sets the flag 321
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.
- You use EIN_MI to set a flag when the flag's number has not yet been defined during creation of the program.

See also

Flag Instruction **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

ENDM

ENDM

Group	Depends on BES	Changes BES
Macros	⊠ **No**	■ **No**

ENDM closes a macro definition. If the instruction ENDM is not preceded by a macro directive, a compiler error occurs.

Example

```
// BES_EIN turns the Bit Result Buffer (BES) on. No parameters are required.

BES_EIN    MACRO                 // Define macro BES_EIN
LAD_M      M_EIN                 // Macro source code
ENDM                             // End macro
```

See also

Macros (Page 34)

EXITM

	Group	Depends on BES	Changes BES
Macros	☒ **No**	■ **No**	

EXITM

EXITM terminates the macro definition depending on a conditional compiler run. Use EXITM to abort a macro depending on symbolic constants.

Example

```
// The following macro is intended to test the inputs / outputs on an interface.
// Depending on the configuration, we exit from the macro early.

DEF_W     2,Interface            // We assume that we are to test 2 inputs here

INTERF    MACRO                  // Define macro INTERF
EIN_A     10                     // Turn on output 10
LAD_E     11                     // Load input 11
UND_E     12                     // Logical operation with input 12
#IF Interface EQ 2               // if 2 inputs are to be tested
MOD_M     INTER_OK               // load status to flag
EXITM                            // and end macro
#ENDIF
UND_E     13                     // otherwise logical operation with input 13
MOD_M     INTER_OK               // and load status to flag
ENDM                             // end of macro definition
```

See also

Macros (Page 34)

GEHUPRI

GEHUPRI Label K

Group	Depends on BES	Changes BES
Program Flow	☒ **No**	**See Text**

GEHUPRI calls a subroutine starting at the specified label.

Operation

```
(Stack pointer)  ← Program address
Stack pointer ← Stack pointer - 2
Program address ← Label address
(Bit Result) ← On
```

Example

```
GEHUPRI    INIT                  // Call subroutine INIT. After returning
                                 // from the subroutine, the Bit Result Buffer
(BES)
                                 // is always turned on
```

Note

- The instruction is not dependent on the status of the Bit Result Buffer (BES). It is always executed.
- After execution of this instruction, the Bit Result Buffer (BES) is always turned on.
- A maximum of 8 subroutines can be nested.

See also

Program Flow Control Instructions (Page 35)

GEHUPRJ

GEHUPRJ Label K

Group	Depends on BES	Changes BES
Program Flow	✸ **Yes**	**See Text**

GEHUPRJ calls a subroutine starting at the specified label. The call is only performed if the Bit Result Buffer (BES) is turned on at the time the instruction is executed.

Operation

```
(Stack pointer)  ← Program address
Stack pointer ← Stack pointer - 2
Program address ← Label address
(Bit Result) ← ON
```

Example

```
GEHUPRJ   INIT                  // If the Bit Result Buffer (BES) is turned on,
                                // call subroutine INIT. Upon return,
                                // the Bit Result Buffer (BES) is always ON
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.
- After execution of this instruction, the Bit Result Buffer (BES) is always turned on.
- A maximum of 8 subroutines can be nested.

See also

Program Flow Control Instructions (Page 35)

GEHUPRN

Group	Depends on BES	Changes BES
Program Flow	✸ **Yes**	**See Text**

GEHUPRN Label K

GEHUPRJ calls a subroutine starting at the specified label. The call is only performed if the Bit Result Buffer (BES) is turned off at the time the instruction is executed.

Operation

```
(Stack pointer)  ← Program address
Stack pointer ← Stack pointer - 2
Program address ← Label address
(Bit Result) ← ON
```

Example

```
GEHUPRN   INIT                  // If the Bit Result Buffer (BES) is turned off,
                                // call subroutine INIT. Upon return,
                                // the Bit Result Buffer (BES)  is always ON
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned off.
- After execution of this instruction, the Bit Result Buffer (BES) is always turned on.
- A maximum of 8 subroutines can be nested.

See also

Program Flow Control Instructions (Page 35)

GEHUPRV

	Group	Depends on BES	Changes BES
Program Flow	✸ **Yes**	**See Text**	

GEHUPRV DestinationAddress V

GEHUPRV calls a subroutine at the start address contained in the specified variable. Use this instruction for dynamic cycle programming.

Operation

```
(Stack pointer)  ← Program address
Stack pointer ← Stack pointer - 2
Program address ←  (Destination address)
(Bit Result) ← ON
```

Example

```
LAD_VL     V_ZYKLUS,PROG1          // Loads label PROG1 into V_ZYKLUS
                                   // (Initialization value)

LOOP:                              // Main program loop
GEHUPRV    V_ZYKLUS                // Jump to the start address that is contained
                                   // as a parameter in the variable V_ZYKLUS
SPRING     LOOP                    // Continue main program loop

PROG1:                             // Subroutine 1
...                                // Program code
LAD_VL     V_ZYKLUS,PROG2          // The starting address of the next cycle
subroutine
                                   // is loaded into the variable V_ZYKLUS,
                                   // PROG2 in this example
UPREND                             // End of the cycle subroutine

PROG2:                             // Subroutine 2
                                   // Program code
LAD_VL     V_ZYKLUS,PROG1          // The starting address of the next cycle
subroutine
                                   // is loaded into the variable V_ZYKLUS,
                                   // PROG1 in this example
UPREND                             // End of the cycle subroutine
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned off.
- After execution of this instruction, the Bit Result Buffer (BES) is always turned on.
- A maximum of 8 subroutines can be nested.

See also

Program Flow Control Instructions (Page 35)

INC_V

		Group	Depends on	Changes BES
Variable Instructions	✸ **Yes**	■ **No**		

INC_V Variable V, Value K

INC_V increments the contents of the variable by the specified value. The result variable VARERG is not changed.

Operation

(Variable) ← (Variable) + Value

Example

```
// In this example, we are manipulating a variable field and set the lower 16 bits
// in each variable. We use INC_V to set the pointer to the next table entry.
LAD_VA     V_ZEIGER,TABELLE_START  // Initialize pointer variable
LAD_VA     V_MASKE,0xFFFF          // hexadecimal FFFF = lower 16 bits
LOOP:
ODER_VI    V_MASKE,V_ZEIGER        // Perform logical operation
LAD_IV     V_ZEIGER,VARERG         // Store result of operation back into table
INC_V      V_ZEIGER,1              // Table pointer to next entry
SPRING     LOOP                    // Back to loop
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

LAD_A

LAD_A Output A

Group	Depends on BES	Changes BES
I/O Instructions	☒ **No**	☑ **Yes**

LAD_A loads the state of the specified output into the Bit Result Buffer (BES).

Operation

(Bit Result) ← (Output)

Example

```
LAD_A      A_TEST                  // Loads the physical state of the output
                                   // into the Bit Result Buffer (BES)
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.

See also

I/O Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

LAD_AI

LAD_AI Pointer V

Group	Depends on BES	Changes BES
I/O Instructions	☒ **No**	☑ **Yes**

LAD_AI loads the state of the output referenced by the pointer into the Bit Result Buffer (BES).

Operation

(Bit Result) ← (Pointer → Output)

Example

```
LAD_AI     V_ZEIGER              // Loads the state of the output referenced by
V_ZEIGER
                                 // into the Bit Result Buffer (BES)
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.

See also

I/O Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

LAD_DT

Group	Depends on BES	Changes BES
Display/Keyboard	✸ **Yes**	■ **No**

LAD_DT FormatMask V, Line K, Column K, Length K

LAD_DT defines the format of subsequent text display using the arguments line, starting position within the line and length of text to be displayed. The coded format description is loaded into the specified variable.

Valid format mask values

Name	Number	Meaning
V_ANZMSK1	91	Show output on the display attached
V_ANZMSK2	95	Show output on the second display attached
V_ANZMSK3	100	Text output via the internal serial port
V_SERMSK	100	Text output via a serial expansion module

Table 10 –Valid Format Mask Values for LAD_DT

Example

```
// A text with a length of 10 is to be inserted into line 7 at position 7

LAD_DT     V_ANZMSK1,ZEILE7,7,10   // Define format for  the display
LAD_VA     V_ANZNR1,1              // Select text string number 1
SETDSP     1,1                     // Output text to display No. 1
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Fehler! Verweisquelle konnte nicht gefunden werden. (Page **Fehler! Textmarke nicht definiert.**)
LAD_DV (Page 54)

LAD_DV

		Group	Depends on BES	Changes BES
Display/Keyboard	✸ **Yes**	■ **No**		

LAD_DV FormatMask V, Line K,
Column K, Length K, Dec. K, Sign K

LAD_DV defines the output of subsequent variable display or the subsequent editor instruction using the arguments line, starting position within the line and field length, number of decimal places and sign enable. The coded format description is loaded into the variable.

Valid format mask values

Name	Number	Meaning
V_EING_NR	238	Input variable, number of variables
V_ANZ_NR	239	Display variable, text number
V_EING_MSK	240	Input description
V_ANZ_MSK	241	Display description
M_VARKEY	561	Flag activates display
M_ANZTRN	564	Flag sends text to display
M_CLRANZ	565	Flag clears display
M_TRNVAR	566	Flag transfers the value entered into the variable

Table 11 – Valid Format Mask Values for LAD_DV

Example

```
// In this example, the variable 214 is to be output at line 1, column 4
// as a 5- digit number, two of which are decimal places, including
// sign character

DEF_V      214,V_TEST              // Variable to be displayed

LAD_VA     V_ANZ_NR,V_TEST         // Load variable to be displayed
LAD_DV     V_ANZ_MSK,0,4,5,2,1     // Define output format
                                   // First  line = 0
EIN_M      M_ANZTRN                // Send message


// In the next example, a variable value is to be edited
// on the display.

DEF_V      214,V_TEST              // Define variable

EIN_M      M_VARKEY                // Activate input
.                                  // input
.
LAD_VA     V_EING_NR,V_TEST        // variable to be edited
LAD_DV     V_EING_MSK,0,4,5,2,1    // Define editing format
                                   // first line = 0
EIN_M      M_VARKEY                // Transfer value from display to variable


// In this example, the text 10 will be output on line 3 of the display.

LAD_VA     V_ANZ_NR,10             // Load text 10 into the display variable
LAD_DT     V_ANZ_MSK,2,1,20        // Define display format
                                   // third line = 2; 1 = column
                                   // 20 = text length
EIN_M      M_ANZTRN
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Fehler! Verweisquelle konnte nicht gefunden werden. (Page **Fehler! Textmarke nicht definiert.**)
LAD_DT (Page 53)

LAD_E

LAD_E Input E

Group	Depends on BES	Changes BES
I/O Instructions	☒ **No**	☑ **Yes**

LAD_E loads the state of the specified input into the Bit Result Buffer (BES).

Operation

(Bit Result) ← (Input)

Example

```
LAD_E     E_TEST                // Load the state of input
                                // E_TEST into the Bit Result Buffer (BES)
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.

See also

I/O Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

LAD_EI

LAD_EI Pointer V

Group	Depends on BES	Changes BES
I/O Instructions	☒ **No**	☑ **Yes**

LAD_EI loads the state of the input referenced by the pointer into the Bit Result Buffer (BES).

Operation

(Bit Result) ← (Pointer → Input)

Example

```
LAD_EI    V_ZEIGER              // Loads the physical state of the input
                                // into the Bit Result Buffer (BES). The variable
                                // contains the number of the input.
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.

See also

I/O Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

LAD_IV

		Group	Depends on	Changes BES
Variable Instructions	✸ **Yes**	■ **No**		

LAD_IV Pointer V, Variable V

LAD_IV loads the contents of the specified variable into the variable referenced by the pointer.

Operation

(Pointer → Variable) ← (Variable)

Example

```
// In this example, measurement results are written to a table.

LAD_VA     V_ZEIGER,1800          // Table begins a variable 1800
LAD_IV     V_ZEIGER,V_WERT        // Loads contents of variable V_Wert  into
variable 1800,
                                  // which is referenced by pointer V_ZEIGER.
INC_V      V_ZEIGER,1             // Increments V_ZEIGER to next table entry
```

Note

This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

LAD_M

LAD_M Flag M

Group	Depends on BES	Changes BES
Flag Instructions	☒ **No**	☑ **Yes**

LAD_M loads the state of the specified flag into the Bit Result Buffer (BES).

Operation

(Bit Result) ← (Flag)

Example

```
LAD_M      M_TEST                 // Loads the state of M_TEST into the
                                  // Bit Result Buffer (BES).
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.

See also

Flag Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

LAD_MI

LAD_MI Pointer V

Group	Depends on BES	Changes BES
Flag Instructions	☒ **No**	☑ **Yes**

LAD_MI loads the state of the flag referenced by the pointer into the Bit Result Buffer (BES).

Operation

`(Bit Result) ← (Pointer → Flag)`

Example

```
LAD_MI     V_TEST                 // The state of the flag referenced by V_TEST
                                  // is loaded into the Bit Result Buffer (BES).
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.

See also

Flag Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

LAD_MV

		Group	Depends on BES	Changes BES
Data Conversion	✱**Yes**	■ **No**		

LAD_MV First Flag M, Variable V

LAD_MV loads the contents of the variable into 32 flags. The transfer occurs bitwise. The first bit of the variable is loaded to the first flag, the second bit to the second flag, etc.

Bit Number	Decimal Value	Hexadecimal Value	Flag Number
First Bit (Bit 0)	1	00000001	First Flag
Second Bit (Bit 1)	2	00000002	Second Flag
Third Bit (Bit 2)	4	00000004	Third Flag
Fourth Bit (Bit 3)	8	00000008	Fourth Flag
Fifth Bit (Bit 4)	16	00000010	Fifth Flag
Sixth Bit (Bit 5)	32	00000020	Sixth Flag
Seventh Bit (Bit 6)	64	00000040	Seventh Flag
Eighth Bit (Bit 7)	128	00000080	Eighth Flag
…	…	…	…
32nd Bit (Bit 31)	2147483648	80000000	32nd Flag

■ Table 12 – LAD_MV Summary

Operation

```
(First Flag + 0)  ← (Variable & 0x00000001)
(First Flag + 1)  ← (Variable & 0x00000002)
(First Flag + 2)  ← (Variable & 0x00000004)
...
(First Flag + 31) ← (Variable & 0x80000000)
```

Example

```
// In this example, we use the instruction LAD_MV to transfer a variable
// into 32 flags. Only the first 16 flags are of interest to us, however, so
// we mask the others out using UND_VV .

LAD_VA     V_MASKE,65535          // hex FFFF to mask out the upper bits
UND_VV     V_DATEN,V_MASKE        // mask out the upper 16 bits (= flags)
LAD_VV     V_DATEN,VARERG         // Store the result back in the variable
LAD_MV     ERSTER_MERKER,V_DATEN  // Transfers the variable to 32 flags
```

Note

This instruction is only executed if the Bit Result Buffer (BES) is turned on.

Please note that the flag starting address - 1 must be a value divisible by 8, for example 1,9,17 or 33.

See also

Data Conversion Instructions (Page 34)
LAD_VM (Page 65)

LAD_P1

Group	Depends on BES	Changes BES
Program Flow	✸ **Yes**	**See Text**

Task start: LAD_P1 Label K
Task stop: LAD_P1 TASK_STOP K

LAD_P1 loads the specified label into the program counter of the first parallel program. The parallel program is started immediately at the specified label.

Example

```
LAD_P1      LABEL                    // Starts task1
```

Bit Result Buffer (BES)

The Bit Result Buffer of the calling task remains unchanged. The Bit Result Buffer in the new task is always turned on.

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.
- You can also change the execution position within the task by using the LAD_P1 instruction.
- The first parallel program is the program that is automatically started at system start-up, which is essentially the main program. In a program with only one task, LAD_P1 has the same meaning for a PLC program as the SPRING instruction.

See also

Program Flow Control Instructions (Page 35)
LAD_P2 (Page 61)
LAD_P3 (Page 62)
LAD_P4 (Page 62)

LAD_P2

Group	Depends on BES	Changes BES
Program Flow	✸ **Yes**	**See Text**

Task start: LAD_P2 Label K
Task stop: LAD_P2 TASK_STOP K

LAD_P2 loads the specified label into the program counter of the second parallel program. The parallel program is started immediately at the specified label.

Example

```
LAD_P2      LABEL                    // Starts task2
```

Bit Result Buffer (BES)

The Bit Result Buffer of the calling task remains unchanged. The Bit Result Buffer in the new task is always turned on.

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.
- You can also change the execution position within the task by using the LAD_P2 instruction.

See also

Program Flow Control Instructions (Page 35)
LAD_P1 (Page 61)
LAD_P3 (Page 62)
LAD_P4 (Page 62)

LAD_P3

		Group	Depends on BES	Changes BES
Program Flow	✸ **Yes**	**See Text**		

Task start: LAD_P3 Label K
Task stop: LAD_P3 TASK_STOP K

LAD_P3 loads the specified label into the program counter of the third parallel program. The parallel program is started immediately at the specified label.

Example

```
LAD_P3     LABEL                    // Starts task3
```

Bit Result Buffer (BES)

The Bit Result Buffer of the calling task remains unchanged. The Bit Result Buffer in the new task is always turned on.

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.
- You can also change the execution position within the task by using the LAD_P3 instruction.

See also

Program Flow Control Instructions (Page 35)
LAD_P1 (Page 61)
LAD_P2 (Page 61)
LAD_P4 (Page 62)

LAD_P4

		Group	Depends on BES	Changes BES
Program Flow	✸ **Yes**	**See Text**		

Task start: LAD_P4 Label K
Task stop: LAD_P4 TASK_STOP K

LAD_P4 loads the specified label into the program counter of the fourth parallel program. The parallel program is started immediately at the specified label.

Example

```
LAD_P4     LABEL                    // Starts task4
```

Bit Result Buffer (BES)

The Bit Result Buffer of the calling task remains unchanged. The Bit Result Buffer in the new task is always turned on.

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.
- You can also change the execution position within the task by using the LAD_P4 instruction.

See also

Program Flow Control Instructions (Page 35)
LAD_P1 (Page 61)
LAD_P2 (Page 61)
LAD_P3 (Page 62)

LAD_VA

	Group	Depends on	Changes BES
Variable Instructions	✸ Yes	■ No	

LAD_VA Variable V, Value K

LAD_VA loads the specified value into the variable.

Operation

(Variable) ← Value

Example

```
DEF_W      100,ZAHL                // ZAHL is declared with the value100

LAD_VA     V_TEST,ZAHL             // loads W_ZAHL into die variable V_TEST

// or:

LAD_VA     V_TEST,100              // loads the value 100 into die variable V_TEST
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

LAD_VI

	Group	Depends	Changes BES
Variable Instructions	✸ Yes	■ No	

LAD_VI Variable V, Pointer V

LAD_VI loads the value of the variable referenced by the pointer into the specified variable.

Operation

(Variable) ← (Pointer → Variable)

Example

```
// In this example, values are to be read from a table.

LAD_VA     V_ZEIGER,1800           // Table starts at variable1800
LAD_VI     V_WERT,V_ZEIGER         // loads the contents of variable 1800, which the
                                   // pointer V_ZEIGER references, into the variable
V_WERT.
INC_V      V_ZEIGER,1              // Increments V_ZEIGER to the next entry
                                   // in the table
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

LAD_VL

		Group	Depends on BES	Changes BES
Program Flow	✸ **Yes**	■ **No**		

LAD_VL Variable V, Label K

LAD_VL loads the program address of the label into the specified variable. The instruction LAD_VL is required to jump to a subroutine using GEHUPRV.

Operation

(Variable) ← (Label address)

Example

```
// In this example, we are programming a sequence of steps using the instructions
// LAD_VL and GEHUPRV. This example illustrates the extent to which program code
// can be reduced by using dynamic cycle programming

LAD_VL     V_ZYKLUS,Schritt1      // Assign label "Schritt1" to cycle variable

Haupt:                            // Main program loop
GEHUPRV    V_ZYKLUS               // Call current step in the cycle
GEHURPI    Control                // Always call monitoring functions
SPRING     Haupt                  // loop

Schritt1:                         // First cycle in the step sequence
LAD_E      E_START                // Start button depressed?
NUND_E     E_NOTAUS               // and emergency off not depressed?
STHOME     1,NORMAL               // then begin start referencing
LAD_VL     V_ZYKLUS,Schritt2      // and activate next step
UPREND                            // End subroutine Schritt1

Schritt2:                         // Second step in the step sequence
LAD_M      M_INPOS_A1             // Axis 1 referencing movement complete?
UPRENDN                           // If not, remain in second step,
STPABS     1,V_ZIELPOS_1          // otherwise start axis 1 towards position
LAD_VL     V_ZYKLUS,Schritt3      // and activate next step
UPREND                            // End subroutine Schritt2

Schritt3:                         // Third step in the step sequence
LAD_M      M_INPOS_A1             // Axis 1 in position?
UPRENDN                           // No, keep waiting
EIN_A      A_ZYLINDER             // Extend cylinder
LAD_VL     V_ZYKLUS,Schritt1      // restart sequence
UPREND                            // subroutine end
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Program Flow Control Instructions (Page 35)
GEHUPRV (Page 51)

LAD_VM

	Group	Depends on BES	Changes BES
Data Conversion	✸ Yes	■ No	

LAD_VM Variable V, First Flag M

LAD_VM loads the state of 32 flags beginning with the specified first flag into a variable. Loading takes place bitwise. The first specified flag is loaded into the first bit of the variable, the second flag into the second bit, etc.

Flag Number	Bit Number	Decimal value	Hexadecimal value
First Flag	First Bit (Bit 0)	1	00000001
Second Flag	Second Bit (Bit 1)	2	00000002
Third Flag	Third Bit (Bit 2)	4	00000004
Fourth Flag	Fourth Bit (Bit 3)	8	00000008
Fifth Flag	Fifth Bit (Bit 4)	16	00000010
Sixth Flag	Sixth Bit (Bit 5)	32	00000020
Seventh Flag	Seventh Bit (Bit 6)	64	00000040
Eighth Flag	Eighth Bit (Bit 7)	128	00000080
...	...	...	...
32nd Flag	32nd Bit (Bit 31)	2147483648	80000000

■ Table 13 – LAD_VM Summary

Operation

```
(Bit 0 of the Variable)  ← (First Flag + 0)
(Bit 1 of the Variable)  ← (First Flag + 1)
(Bit 2 of the Variable)  ← (First Flag + 2)
...
(Bit 31 of the Variable) ← (First Flag + 31)
```

Example

```
// In this example, we use the instruction LAD_VM to transfer 32 flags into
// a variable. We are only interested in the first 16 flags, so
// we mask out the others with UND_VV.

LAD_VM     V_DATEN,ERSTER_MERKER   // Load 32 flags into the variable
UND_VA     V_DATEN,0xFFFF          //mask out upper 16 bits (= flags)
LAD_VV     V_DATEN,VARERG          // store result back into variable
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.
- Please note that the flag starting address - 1 must be a value divisible by 8, for example 1,9,17 or 33. Any other flag value causes a compiler error.

See also

Data Conversion Instructions (Page 34)
LAD_MV (Page 60)

LAD_VV

		Group	Depends on BES	Changes BES
Variable Instructions	✸ **Yes**	■ **No**		

LAD_VV Variable1 V, Variable2 V

LAD_VV loads the contents of variable2 into variable1.

Operation

(Variable1) ← (Variable2)

Example

```
DEF_V     10,V_TEST              // Assigns the name V_TEST to variable 10
DEF_V     11,V_LAENGE            // Assigns the name V_LAENGE to variable 11
LAD_VV    V_TEST,V_LAENGE        // loads the contents of the variable V_LAENGE
into
                                 // the variable V_TEST
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

MACRO

		Group	Depends on BES	Changes BES
Macros	⊠ **No**	■ **No**		

Name K MACRO [Par 1 K, [Par 2 K, [...]]]

MACRO defines the start of a macro definition. Within the macro definition block, you then assign the name of the macro, any parameters and the associated PLC program code. The square brackets in the syntax indicate that you can decide for yourself how many parameters you want to pass to a macro instruction.

Example without parameters

```
// BES_EIN turns on the Bit Result Buffer (BES). No parameters are required

BES_EIN   MACRO                  // Define macro BES_EIN
LAD_M     M_EIN                  // macro source code
ENDM                             // Macro end
```

Example with parameters

```
// The following macro starts a system timer with
// a specified value. The macro contains the timer number
// and the start time for this timer as parameters.

TSTART    MACRO TIM,KONST        // Define macro TSTART with the
                                 // parameters TIM and KONST.
LAD_VA    V_&TIM,KONST           // The parameter TIM is expanded to
                                 // V_&TIM, where &TIM stands for the
                                 // character string passed. The parameter
                                 // KONST is transferred unchanged.
EIN_M     M_&TIM                 // The parameter TIM is expanded a
                                 // second time, this time to M_&TIM
ENDM                             // End of macro definition
```

```
// To call the macro from the PLC program, we write:

TSTART    TIM_15,2000

//which has the following meaning: start timer 15 with the value 2000. The
// compiler translates the macro call into the following program code:

LAD_VA    V_TIM_15,2000
EIN_M     M_TIM_15

// The parameters passed are expanded and converted to new values.
```

Note

- Macros are not functions in the strict sense of the term. A macro does not call a subroutine. Instead, the complete source code of the macro is compiled instead of the macro call. A macro twenty lines long occupies exactly these 20 lines of PLC program memory for each call.
- Please note that macro names must always contain only the letters A-Z and the numbers 0-9. The macro name may not begin with a number. Spaces and other special characters are not allowed in macro names. The same applies to parameter names.

See also

Macros (Page 34)
ENDM (Page 48)
EXITM (Page 49)

MOD_A

MOD_A Output A

Group	Depends on BES	Changes BES
I/O Instructions	**No**	**No**

MOD_A sends the state of the Bit Result Buffer (BES) to the specified output.

Operation

(Output) ← (Bit Result Buffer (BES))

Example

```
BES_EIN                          // Turns the Bit Result Buffer (BES) on
MOD_A     A_TEST                 // sends the contents of the Bit Result Buffer
(BES)
                                 // to the output
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed. The effects of the instruction, however, depend on the state of the Bit Result Buffer (BES).

See also

I/O Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

MOD_AI

MOD_AI Pointer V

Group	Depends on BES	Changes BES
I/O Instructions	**No**	**No**

MOD_AI sends the state of the Bit Result Buffer (BES) to the output referenced by the pointer.

Operation

```
(Pointer → Output) ← (Bit Result Buffer (BES))
```

Example

```
LAD_VA     V_AUSGANG,W_TAB        // Loads constant  W_TAB into the variable
V_AUSGANG
MOD_AI     V_AUSGANG              // sends the contents of the Bit Result Buffer
(BES)
                                  // to the output identified by  number contained
                                  // in the variable
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed. The effects of the instruction, however, depend on the state of the Bit Result Buffer (BES).
- You use MOD_AI to send the contents of the Bit Result Buffer (BES) to an output, when the output's number has not yet been defined during creation of the program.

See also

I/O Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

MOD_M

MOD_M Flag M

Group	Depends on BES	Changes BES
Flag Instructions	**No**	**No**

MOD_M loads the state of the Bit Result Buffer (BES) into the specified flag.

Operation

```
(Flag) ← (Bit Result Buffer (BES))
```

Example

```
MOD_M      M_TEST                 // Loads the Bit Result Buffer into M_TEST
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed. The effects of the instruction, however, depend on the state of the Bit Result Buffer (BES).

See also

Flag Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

MOD_MI

MOD_MI Pointer V

Group	Depends on BES	Changes BES
Flag Instructions	**No**	**No**

MOD_MI loads the state of the Bit Result Buffer (BES) into the flag referenced by the pointer.

Operation

```
(Pointer → Flag) ← (Bit Result Buffer (BES))
```

Example

```
MOD_MI     V_MERKER                // the contents of the Bit Result Buffer (BES)
are loaded
                                   // into the flag pointed to by  V_MERKER
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed. The effects of the instruction, however, depend on the state of the Bit Result Buffer (BES).
- You use MOD_MI to load the contents of the Bit Result Buffer (BES) to a flag when the flag's number has not yet been defined during creation of the program.

See also

Flag Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

MUL_IV

Group	Depends on	Changes BES
Variable Instructions	**Yes**	**No**

MUL_IV Pointer V, Variable V

MUL_IV multiplies the contents of the variable referenced by the pointer by the contents of the specified variable. The result of the multiplication operation is placed in the result variable VARERG, and any multiplication overflow (> 32 Bit) is stored in the result variable MUL_REST.

Operation

```
(VARERG)   ←  (Pointer → Variable) * (Variable)
(MUL_REST) ← ((Pointer→ Variable) * (Variable)) >> 32
```

Example

```
MUL_IV     V_ZEIGER,V_TEST         // The contents of the variable pointed to by
                                   // V_ZEIGER are multiplied by V_TEST. The result
                                   // is stored in VARERG
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

MUL_VA

	Group	Depends on BES	Changes BES
Variable Instructions	✸ **Yes**	■ **No**	

MUL_VA Variable V, Value K

MUL_VA multiplies the contents of the variable by the specified value. The result of the multiplication operation is placed in the result variable VARERG, and any multiplication overflow (> 32 Bit) is stored in the result variable MUL_REST.

Operation

```
(VARERG)   ← (Variable) * Value
(MUL_REST) ← ((Variable) * Value) >> 32
```

Example

```
DEF_W     10,ZAHL                // Declares ZAHL with the value 10
MUL_VA    V_TEST,ZAHL            // Multiplies the variable by the value ZAHL.
                                 // The result is stored in VARERG
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

MUL_VI

	Group	Depends on BES	Changes BES
Variable Instructions	✸ **Yes**	■ **No**	

MUL_VI Variable V, Pointer V

MUL_VI multiplies the contents of the variable referenced by the pointer by the contents of the specified variable. The result of the multiplication operation is placed in the result variable VARERG, and any multiplication overflow (> 32 Bit) is stored in the result variable MUL_REST.

Operation

```
(VARERG)   ← (Variable) * (Pointer → Variable)
(MUL_REST) ← ((Variable) * (Pointer → Variable)) >> 32
```

Example

```
MUL_VI    V_TEST,V_ZEIGER        // V_TEST is multiplied by the contents of
                                 // variable referenced by the pointer V_ZEIGER.
                                 // The result is stored in VARERG.
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

MUL_VV

Group	Depends on BES	Changes BES
Variable Instructions	✸ **Yes**	■ **No**

MUL_VV Variable1 V, Variable2 V

MUL_VV multiplies the contents of variable1 by variable2. The result of the multiplication operation is placed in the result variable VARERG, and any multiplication overflow (> 32 Bit) is stored in the result variable MUL_REST.

Operation

```
(VARERG)    ← (Variable1) * (Variable2)
(MUL_REST) ← ((Variable1) * (Variable2)) >> 32
```

Example

```
LAD_VA     V_ZAHL1,10             // The value 10 is loaded into
                                  // the variable V_ZAHL1.
LAD_VA     V_ZAHL2,20             // The value 20 is loaded into
                                  // the variable V_ZAHL2.
MUL_VV     V_ZAHL1,V_ZAHL2        // Multiplies the variable V_ZAHL1 by the
                                  // variable V_ZAHL2, and the result is
                                  //stored in VARERG.
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

NLAD_A

NLAD_A Output A

Group	Depends on BES	Changes BES
I/O Instructions	☒ **No**	☑ **Yes**

NLAD_A loads the inverted state of the output into the Bit Result Buffer (BES).

Operation

```
(Bit Result Buffer) ← ~(Output)
```

Example

```
NLAD_A     A_TEST                 // Loads the inverted state of A_TEST
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.

See also

I/O Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

NLAD_E

NLAD_E Input E

Group	Depends on BES	Changes BES
I/O Instructions	☒ **No**	☑ **Yes**

NLAD_E loads the inverted state of the input into the Bit Result Buffer (BES).

Operation

```
(Bit Result Buffer (BES)) ← ~(Input)
```

Example

```
NLAD_E     E_TEST                  // Loads the inverted state of E_TEST
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.

See also

I/O Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

NLAD_M

NLAD_M Flag M

Group	Depends on BES	Changes BES
Flag Instructions	☒ **No**	☑ **Yes**

NLAD_M loads the inverted state of the flag into the Bit Result Buffer (BES).

Operation

```
(Bit Result Buffer (BES)) ← ~(Flag)
```

Example

```
NLAD_M     M_TEST                  // Loads the inverted state of M_TEST
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.

See also

I/O Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

NODER_A

NODER_A Output A

Group	Depends on BES	Changes BES
I/O Instructions	☒ **No**	☑ **Yes**

NODER_A performs a logical OR operation between the state of the Bit Result Buffer (BES) and the inverted state of the specified output. The result of this operation is placed into the Bit Result.

Operation

```
(Bit Result) ← (Bit Result) | ~(Output)
```

Example

```
LAD_A      A_TEST1                // If A_TEST1
NODER_A    A_TEST2                // and A_TEST2 are turned off,
SPRINGJ    TESTZYKLUS             // then jump to label TESTZYKLUS
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.

See also

I/O Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

NODER_E

NODER_E Input E

Group	Depends on BES	Changes BES
I/O Instructions	☒ **No**	☑ **Yes**

NODER_E performs a logical OR operation between the state of the Bit Result Buffer (BES) and the inverted state of the specified input. The result of this operation is placed into the Bit Result.

Operation

```
(Bit Result) ← (Bit Result) | ~(Input)
```

Example

```
LAD_E      E_TEST1                // If E_TEST1
NODER_E    E_TEST2                // and E_TEST2 are turned off,
SPRINGJ    TESTZYKLUS             // then jump to label TESTZYKLUS
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.

See also

I/O Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

NODER_M

NODER_M Flag M

Group	Depends on BES	Changes BES
Flag Instructions	☒ **No**	☑ **Yes**

NODER_M performs a logical OR operation between the state of the Bit Result Buffer (BES) and the inverted state of the specified flag. The result of this operation is placed into the Bit Result.

Operation

```
(Bit Result) ← (Bit Result) | ~(Flag)
```

Example

```
LAD_M     M_TEST1              // If M_TEST1 and
NODER_M   M_TEST2              // M_TEST2 are turned off
SPRINGJ   TESTZYKLUS           // then jump to label TESTZYKLUS
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.

See also

Flag Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

NUND_A

NUND_A Output A

Group	Depends on BES	Changes BES
I/O Instructions	☒ **No**	☑ **Yes**

NUND_A performs a logical AND operation between the state of the Bit Result Buffer (BES) and the inverted state of the specified output. The result of this operation is placed into the Bit Result.

Operation

```
(Bit Result) ← (Bit Result) & ~(Output)
```

Example

```
LAD_A     A_TEST1              // If A_TEST1 or A_TEST2 is turned on
NUND_A    A_TEST2              // or both outputs are turned off,
SPRINGJ   TESTZYKLUS           // then jump to label TESTZYKLUS
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.

See also

I/O Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

NUND_E

NUND_E Input E

Group	Depends on BES	Changes BES
I/O Instructions	☒ **No**	☑ **Yes**

NUND_E performs a logical AND operation between the state of the Bit Result Buffer (BES) and the inverted state of the specified input. The result of this operation is placed into the Bit Result.

Operation

```
(Bit Result) ← (Bit Result) & ~(Input)
```

Example

```
LAD_E      E_TEST1                // If E_TEST1 or E_TEST2 is turned on
NUND_E     E_TEST2                // or both inputs are turned off,
SPRINGJ    TESTZYKLUS             // then jump to label TESTZYKLUS
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.

See also

I/O Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

NUND_M

NUND_M Flag M

Group	Depends on BES	Changes BES
Flag Instructions	☒ **No**	☑ **Yes**

NUND_A performs a logical AND operation between the state of the Bit Result Buffer (BES) and the inverted state of the specified flag. The result of this operation is placed into the Bit Result.

Operation

```
(Bit Result)) ← (Bit Result)) & ~(Flag)
```

Example

```
LAD_M      M_TEST1                // If M_TEST1 or M_TEST2 is turned on
NUND_M     M_TEST2                // or both flags are turned off,
SPRINGJ    TESTZYKLUS             // then jump to label TESTZYKLUS
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.

See also

Flag Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

ODER_A

ODER_A Output A

Group	Depends on BES	Changes BES
I/O Instructions	☒ **No**	☑ **Yes**

ODER_A performs a logical OR operation between the states of the Bit Result Buffer (BES) and the specified output. The result of this operation is placed into the Bit Result.

Operation

```
(Bit Result) ← (Bit Result) | (Output)
```

Example

```
LAD_A     A_TEST1              // If A_TEST1 or A_TEST2 is turned on
ODER_A    A_TEST2              // or both outputs are turned on,
SPRINGJ   TESTZYKLUS           // then jump to label TESTZYKLUS
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.

See also

I/O Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

ODER_E

ODER_E Input E

Group	Depends on BES	Changes BES
I/O Instructions	☒ **No**	☑ **Yes**

ODER_E performs a logical OR operation between the states of the Bit Result Buffer (BES) and the specified input. The result of this operation is placed into the Bit Result.

Operation

```
(Bit Result) ← (Bit Result) | (Input)
```

Example

```
LAD_E     E_TEST1              // If E_TEST1 or E_TEST2 is turned on
ODER_E    E_TEST2              // or both inputs are turned on,
SPRINGJ   TESTZYKLUS           // then jump to label TESTZYKLUS
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.

See also

I/O Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

ODER_M

ODER_M Flag M

Group	Depends on BES	Changes BES
Flag Instructions	☒ **No**	☑ **Yes**

ODER_M performs a logical OR operation between the states of the Bit Result Buffer (BES) and the specified flag. The result of this operation is placed into the Bit Result.

Operation

```
(Bit Result) ← (Bit Result) | (Flag)
```

Example

```
LAD_M      M_TEST1                // IF M_TEST1
ODER_M     M_TEST2                // or M_TEST2 is turned on,
SPRINGJ    TESTZYKLUS             // then jump to label TESTZYKLUS
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.

See also

Flag Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

ODER_IV

Group	Depends on	Changes BES
Variable Instructions	✸ **Yes**	■ **No**

ODER_IV Pointer V, Variable V

ODER_IV performs a binary OR operation between the contents of the variable referenced by the pointer and the specified variable. The result of this operation is stored in the result variable VARERG.

Operation

```
(VARERG) ← (Pointer → Variable) | (Variable)
```

Example

```
// In this example, we have a variable field, which for example is written by the
PC;
// in this variable field, all the lower 16 bits are set. The result of the
// logical operation is stored back in the table.

LAD_VA     V_ZEIGER,2000          // Initialize pointer variable
LAD_VA     V_MASKE,0xFFFF         // Mask with hex FFFF

LOOP:
ODER_IV    V_ZEIGER,V_MASKE       // Perform logical operation
LAD_IV     V_ZEIGER,VARERG        // Result of operation back to table
INC_V      V_ZEIGER,1             // Table pointer to next entry
SPRING     LOOP                   // Back to loop
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

ODER_VA

		Group	Depends on BES	Changes BES
Variable Instructions	✸ Yes	■ No		

ODER_VA Variable V, Value K

ODER_VA performs a binary OR operation between the contents of the variable and the specified value. The result of this operation is stored in the result variable VARERG.

Operation

(VARERG) ← (Variable) | Value

Example

```
// In this example, we use the instruction LAD_VM to transfer 32 flags into
// a variable. The first 16 flags are to be set, so we
// we perform a logical operation on the flags using ODER_VA.

LAD_VM     V_DATEN,ERSTER_MERKER   // Transfer of 32flags into the variable
ODER_VA    V_DATEN,0xFFFF          // Set the lower 16 bits with hex FFFF
LAD_VV     V_DATEN,VARERG          // Store the result in the variable
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

ODER_VI

		Group	Depends on	Changes BES
Variable Instructions	✸ Yes	■ No		

ODER_VI Variable V, Pointer V

ODER_VI performs a binary OR operation between the contents of the variable referenced by the pointer and the specified variable. The result of this operation is stored in the result variable VARERG.

Operation

(VARERG) ← (Variable) | (Pointer → Variable)

Example

```
// In this example, the lower 16 bits are set in a variable field .
// The result is stored back in the table.

LAD_VA     V_ZEIGER,2000           // Initialize pointer variable
LAD_VA     V_MASKE,0xFFFF          // mask lower 16 bits

LOOP:
ODER_VI    V_MASKE,V_ZEIGER        // Perform logical operation
LAD_IV     V_ZEIGER,VARERG         // Result of operation back into table
INC_V      V_ZEIGER,1              // Table pointer to next entry
SPRING     LOOP                    // Back to loop
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

ODER_VV

	Group	Depends on BES	Changes BES
Variable Instructions	✸ **Yes**	■ **No**	

ODER_VV Variable1 V, Variable2 V

ODER_VV performs a binary OR operation between the two specified variables. The result of this operation is stored in the result variable VARERG.

Operation

```
(VARERG) ← (Variable1) | (Variable2)
```

Example

```
// In this example, we use the instruction LAD_VM to transfer 32 flags into
// a variable. The first 16 flags are to be set.

LAD_VA     V_MASKE,0xFFFF          // hex FFFF to the lower 16 Bits
LAD_VM     V_DATEN,ERSTER_MERKER   // transfer 32 flags to the variable
ODER_VV    V_DATEN,V_MASKE         // the lower 16 bits (= flags) are set
                                   // with V_MASKE
LAD_VV     V_DATEN,VARERG          // Store result back in variable
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

SLL_VA

	Group	Depends on	Changes BES
Variable Instructions	✸ **Yes**	■ **No**	

SLL_VA Variable V, Value K

SLL_VA left shifts the contents of the variable specified as the first parameter by the specified value. The result is stored in the result variable VARERG.

Operation

```
(VARERG) = (Variable) << Value
```

Example

```
// We shift the value of a variable three bits to the left using SLL_V.

LAD_VA     V_ZAHL,1                // Load the value 1 to V_ZAHL
SLL_VA     V_ZAHL,3                // Shift the contents of V_ZAHL (=0001) three
bits
                                   // to the left. There result is equal to 8
(=1000).
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

SLL_VV

	Group	Depends on BES	Changes BES
Variable Instructions	✸ **Yes**	■ **No**	

SLL_VV Variable1 V, Variable2 V

SLL_VV shifts the contents of variable1 to the left by the contents of variable2. The result is stored in the result variable VARERG.

Operation

```
(VARERG) = (Variable1) << (Variable2)
```

Example

```
// We shift the value of a variable three bits to the left using SLL_VV.

LAD_VA     V_ZAHL,1                // Load the value 1 to V_ZAHL
LAD_VA     V_ANZAHL,3              // Load the value 3 to V_ANZAHL
SLL_VV     V_ZAHL,V_ANZAHL         // Shift the contents of V_ZAHL (=0001) three
bits
                                   // to the left. There result is equal to 8
(=1000).
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

SPRING

SPRING Label K

Group	Depends on BES	Changes BES
Program Flow	**No**	**No**

SPRING continues program flow at the specified label.

Operation

```
Program address ← Label address
(Bit Result) ← ON
```

Example

```
SPRING     INIT_TIMER              // Jumps to the label in the program

INIT_TIMER:                             // Program INIT_TIMER
...                                // Program code
```

See also

Program Flow Control Instructions (Page 35)

SPRINGJ

SPRINGJ Label K

Group	Depends on BES	Changes BES
Program Flow	✸ **Yes**	■ **No**

SPRINGJ continues program flow at the specified label if the Bit Result Buffer (BES) is currently turned on.

Operation

```
Program address ← Label address
(Bit Result) ← ON
```

Example

```
SPRINGJ   INIT_TIMER            // Jumps to the label INIT_TIMER in the program
                                // if the Bit Result Buffer (BES) is turned on.
INIT_TIMER:                     // Program INIT_TIMER
...                             // Program code
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Program Flow Control Instructions (Page 35)

SPRINGN

Group	Depends on BES	Changes BES
Program Flow	✸ **Yes**	■ **No**

SPRINGN Label K

SPRINGN continues program flow at the specified label if the Bit Result Buffer (BES) is currently turned off.

Operation

```
Program address ← Label address
(Bit Result) ← ON
```

Example

```
SPRINGJ   INIT_TIMER            // Jumps to the label INIT_TIMER in the program
                                // if the Bit Result Buffer (BES) is turned off.
INIT_TIMER:                     // Program INIT_TIMER
...                             // Program code
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned off.

See also

Program Flow Control Instructions (Page 35)

SRL_VA

Group	Depends on BES	Changes BES

Variable Instructions	✸ Yes	■ No

SRL_VA Variable V, Value K

SRL_VA shifts the contents of the variable to the right by the specified value. The result is stored in the result variable VARERG.

Operation

(VARERG) ← (Variable) >> Value

Example

```
// We shift the value of a variable three bits to the right using SRL_V.

LAD_VA     V_ZAHL,8              // Load the value 8 to V_ZAHL
SRL_VA     V_ZAHL,3              // Shift contents of V_ZAHL (=1000)  three bits
                                 // to the right; result  is equal to 1 (=0001)
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

SRL_VV

Group	Depends on BES	Changes BES

Variable Instructions	✸ Yes	■ No

SRL_VV Variable1 V, Variable2 V

SRL_VV shifts the contents of variable1 to the right by the contents of variable2. The result is stored in the result variable VARERG.

Operation

(VARERG) ← (Variable1) >> (Variable2)

Example

```
// We shift the value of a variable three bits to the right using SRL_VV

LAD_VA     V_ZAHL,8              // Load the value 8 to V_ZAHL
LAD_VA     V_ANZAHL,3            //Load the value 3 to V_ANZAHL
SRL_VV     V_ZAHL,V_ANZAHL       // Shift contents of V_ZAHL (=1000)  three bits
                                 // to the right; result  is equal to 1 (=0001)
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

SUB_IV

	Group	Depends on BES	Changes BES
Variable Instructions	✸ **Yes**	■ **No**	

SUB_IV Pointer V, Variable V

SUB_IV subtracts the contents of the specified variable from the contents of the variable referenced by the pointer. The result of the subtraction operation is stored in VARERG.

Operation

(VARERG) ← (Pointer → Variable) – (Variable)

Example

```
SUB_IV     V_ZEIGER,V_TEST         // V_TEST is subtracted from the contents of the
                                   // variable pointed to by V_ZEIGER. The result
                                   // is stored in VARERG
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

SUB_VA

	Group	Depends on BES	Changes BES
Variable Instructions	✸ **Yes**	■ **No**	

SUB_VA Variable V, Value K

SUB_VA subtracts the specified value from the contents of the variable. The result of the subtraction operation is stored in VARERG.

Operation

(VARERG) ← (Variable) – Value

Example

```
SUB_VA     V_TEST,10               // Subtracts 10 from V_TEST and stores
                                   // the result in VARERG
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

SUB_VI

Group	Depends on BES	Changes BES
Variable Instructions	✸ Yes	■ No

SUB_VI Variable V, Pointer V

SUB_VI subtracts the contents of the variable referenced by the pointer from the contents of the specified variable. The result of the subtraction operation is stored in VARERG.

Operation

(VARERG) ← (Variable) – (Pointer → Variable)

Example

```
SUB_VI     V_TEST,V_ZEIGER         // Subtracts the contents of the variable pointed
to
                                   // by V_ZEIGER from the contents of the variable
V_TEST
                                   // The result is stored in VARERG
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

SUB_VV

Group	Depends on BES	Changes BES
Variable Instructions	✸ Yes	■ No

SUB_VV Variable1 V, Variable2 V

SUB_VV subtracts the contents of variable2 from the contents of variable1. The result of the subtraction operation is stored in VARERG.

Operation

(VARERG) ← (Variable1) – (Variable2)

Example

```
LAD_VA     V_ZAHL1,20              // Load the value 20 to variable V_ZAHL1
LAD_VA     V_ZAHL2,10              // Load the value 10 to variable V_ZAHL2
SUB_VV     V_ZAHL1,V_ZAHL2         // Subtracts variable V_ZAHL2 from
                                   // variable V_ZAHL1 and stores the result
                                   // in VARERG
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

TEXT

TEXT

Group	Depends on BES	Changes BES
Compiler	☒ **No**	■ **No**

TEXT declares all subsequent information in the current file up to the end of the file as text definitions.

Example

```
TEXT

// Text definitions begin here. Only text may appear from
// here to end of file.

TEXT001    "********************" ;
TEXT002    " MICRO DESIGN GmbH  " ;
TEXT003    "       MC-1A        " ;
TEXT004    "********************" ;
TEXT005    "Version 1.0 30.10.98" ;
TEXT006    "TEXT006             " ;
TEXT007    "" ;
TEXT008    "TEXT008             " ;
```

Special characters in the text

You can use special control characters or non-printable graphics characters within the text. The syntax for this is modeled on the C++ language:

- Carriage return/line feed (CRLF): string "\n", for example "evaluation\n"
- Line feed (LF): string "\r", for example "new line\r"
- All other character codes: character "\" followed by the numerical ASCII code of the character you want, for example "\025" for the character with ASCII code 25, "\128" for the character with ASCII code 128, etc. You can also specify the ASCII code in hexadecimal notation. To do this, simply place the combination "\x" instead of "\" in front of the code, for example "\x3A" for the character with the ASCII code 3A hex.

Note

- You can also use more that one file for the definition of text.
- Please note that no other PLC instructions may follow the "TEXT" instruction. Subsequent symbolic definitions are also completely ignored up to end of file.
- You should ensure that the compiler assigns consecutive text numbers. As shown in the example above, text numbers are valid only if they start at 1 and are consecutive.
- You can store the text in a separate file or together with your source code in the same file. If you choose the second option, please note that no further PLC instructions may follow the "TEXT" instruction.

See also

Fehler! Verweisquelle konnte nicht gefunden werden. (Page **Fehler! Textmarke nicht definiert.**)
Fehler! Verweisquelle konnte nicht gefunden werden. (Page **Fehler! Textmarke nicht definiert.**)

UND_A

UND_A Output A

Group	Depends on BES	Changes BES
I/O Instructions	☒ **No**	☑ **Yes**

UND_A performs a logical AND operation between the current state of the Bit Result Buffer (BES) and the state of the specified output. The result of this operation is placed into the Bit Result Buffer (BES).

Operation

```
(Bit Result) ← (Bit Result) & (Output)
```

Example

```
LAD_A      A_TEST1              // IF A_TEST1 is turned on
UND_A      A_TEST2              // and A_TEST2 is turned on
SPRINGJ    TESTZYKLUS           // then jump to label TESTZYKLUS
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.

See also

I/O Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

UND_E

UND_E Input E

Group	Depends on BES	Changes BES
I/O Instructions	☒ **No**	☑ **Yes**

UND_E performs a logical AND operation between the current state of the Bit Result Buffer (BES) and the state of the specified input. The result of this operation is placed into the Bit Result Buffer (BES).

Operation

```
(Bit Result) ← (Bit Result) & (Input)
```

Example

```
LAD_E      E_TEST1              // If E_TEST1 is turned on
UND_E      E_TEST2              // and E_TEST2 is turned on
SPRINGJ    TESTZYKLUS           // then jump to label TESTZYKLUS
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.

See also

I/O Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

UND_IV

		Group	Depends on BES	Changes BES
Variable Instructions	✸ **Yes**	■ **No**		

UND_IV Pointer V, Variable V

UND_IV performs a binary AND operation between the contents of the variable referenced by the pointer and the contents of the specified variable. The result is stored in the result variable VARERG.

Operation

(VARERG) ← (Pointer → Variable) & (Variable)

Example

```
// In this example, we have a variable field, which is written for example by the
// PC, and we mask out the upper 16 bits in the variable field (they contain other
// information).

DEF_W     2000,TABELLE_START     // Start of the variable field
LAD_VA    V_ZEIGER,TABELLE_START // Initialize pointer variable
LAD_VA    V_MASKE,65535          // corresponds to hexadecimal FFFF

LOOP:
UND_IV    V_ZEIGER,V_MASKE       // Perform logical operation
LAD_IV    V_ZEIGER,VARERG        // Result of operation back to table
INC_V     V_ZEIGER,1             // Table pointer to next entry
SPRING    LOOP                   // Back to loop
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

UND_M

UND_M Flag M

Group	Depends on BES	Changes BES
Flag Instructions	☒ **No**	☑ **Yes**

UND_M performs a logical AND operation between the current state of the Bit Result Buffer (BES) and the state of the specified flag. The result is placed into the Bit Result Buffer (BES).

Operation

(Bit Result) ← (Bit Result) & (Flag)

Example

```
LAD_M     M_TEST1                // If M_TEST1
UND_M     M_TEST2                // and M_TEST2 are turned on,
SPRINGJ   TESTZYKLUS             // then jump to label TESTZYKLUS
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.

See also

Flag Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

UND_VA

	Group	Depends on	Changes BES
Variable Instructions	✸ Yes	■ No	

UND_VA Variable V, Value K

UND_VA performs a binary AND operation between the contents of the variable and the specified value. The result is stored in the result variable VARERG.

Operation

(VARERG) ← (Variable) & Value

Example

```
// In this example, we use the instruction LAD_VM to transfer
// 32 flags to a variable. However, we are only interested in the first
// 16 flags, so we mask out the others using UND_VA.

LAD_VM     V_DATEN,ERSTER_MERKER   // Transfer of 32 flags into the variable
UND_VA     V_DATEN,0xFFFF          // Use hex FFFF to mask out the upper 16 bits
LAD_VV     V_DATEN,VARERG          // Store result back into variable
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

UND_VI

	Group	Depends on BES	Changes BES
Variable Instructions	✸ Yes	■ No	

UND_VI Variable V, Pointer V

UND_VI performs a binary AND operation between the contents of the specified variable and the variable referenced by the pointer. The result is stored in the result variable VARERG.

Operation

(VARERG) ← (Variable) & (Pointer → Variable)

Example

```
// In this example, we have a variable field, which is written for example by the
// PC, and we mask out the upper 16 bits in the variable field (they contain other
// information).

LAD_VA     V_ZEIGER,2000           // Initialize pointer variable
LAD_VA     V_MASKE,0xFFFF          // Mask for the lower 16 bits

LOOP:
UND_VI     V_MASKE,V_ZEIGER        // Perform logical operation
LAD_IV     V_ZEIGER,VARERG         // Result of operation back to table
INC_V      V_ZEIGER,1              // Table pointer to next entry
SPRING     LOOP                    // Back to loop
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

UND_VV

	Group	Depends on BES	Changes BES
Variable Instructions	✸ **Yes**	■ **No**	

UND_VA Variable1 V, Variable2 V

UND_VV performs a binary AND operation between the contents of the two specified variables. The result is stored in the result variable VARERG.

Operation

(VARERG) ← (Variable1) & (Variable2)

Example

```
// In this example we use the instruction LAD_VM to transfer 32 flags
// to a variable . However, we are only interested in the first
// 16 flags, so we mask out the others using UND_VV.

LAD_VA     V_MASKE,65535          // Hex FFFF to mask out the upper bits
LAD_VM     V_DATEN,ERSTER_MERKER  // Transfer of 32 flags to the variable
UND_VV     V_DATEN,V_MASKE        // Mask out the upper 16 bits  (= flags)
LAD_VV     V_DATEN,VARERG         // Store the result back in the variable
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Variable Instructions (Page 31)

UPREND

UPREND

Group	Depends on BES	Changes BES
Program Flow	☒ **No**	☑ **Yes**

UPREND ends a subroutine. The program continues from the point where the subroutine was called.

Operation

Program address ← (Stack pointer)
Stack pointer ← Stack pointer + 2
(Bit Result) ← ON

Example

```
GEHUPR     UPROG                  // Calls subroutine UPROG
SPRING     LOOP

UPROG:                            // Subroutine UPROG
...                               // Program code
UPREND                            // Ends the subroutine
                                  // Afterwards, the Bit Result is always ON
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.
- After the return from the subroutine, the Bit Result Buffer (BES) is always turned on.

See also

Program Flow Control Instructions (Page 35)

UPRENDJ

UPRENDJ

Group	Depends on BES	Changes BES
Program Flow	✸ **Yes**	☑ **Yes**

UPRENDJ ends a subroutine. The program continues from the point where the subroutine was called.

Operation

```
Program address ← (Stack pointer)
Stack pointer ← Stack pointer + 2
(Bit Result) ← ON
```

Example

```
GEHUPR     UPROG                  // Calls subroutine UPROG
SPRING     LOOP

UPROG:                            // Subroutine
...                               // Program code
UPRENDJ                           // Terminates the subroutine if Bit Result on
                                  // Afterwards, the Bit Result is always ON
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned on.

See also

Program Flow Control Instructions (Page 35)

UPRENDN

Group	Depends on BES	Changes BES
Program Flow	✸ **Yes**	☑ **Yes**

UPRENDN

UPRENDN ends a subroutine. The program continues from the point where the subroutine was called.

Operation

```
Program address ← (Stack pointer)
Stack pointer ← Stack pointer + 2
(Bit Result) ← ON
```

Example

```
GEHUPR     UPROG                  // Calls subroutine UPROG
SPRING     LOOP

UPROG:                            // Subroutine
...
UPRENDN                           // Terminates the subroutine if Bit Result off
                                  // Afterwards, the Bit Result is always ON
```

Note

- This instruction is only executed if the Bit Result Buffer (BES) is turned off.
- After the return from the subroutine, the Bit Result Buffer (BES) is always turned on.

See also

Program Flow Control Instructions (Page 35)

VERG_IV

	Group	Depends on BES	Changes BES
Variable Comparison	**No**	**No**	

VERG_IV Pointer V, Variable V

VERG_IV compares the contents of the variable specified by the pointer with the contents of the specified variable. The result is stored in the variable comparison flags.

Flag	Is set if ...
M_GLEICH	... the contents of the variable specified by the pointer and the specified variable are equal
M_KLEINER	... the contents of the variable specified by the pointer are less than the contents of the specified variable
M_GROESSER	... the contents of the variable specified by the pointer are greater than the contents of the specified variable

Table 14 – Variable Comparison Flags for VERG_IV

Operation

```
(M_GLEICH)   ← (Pointer → Variable) = (Variable)
(M_KLEINER) ← (Pointer → Variable) < (Variable)
(M_GROESSER) ← (Pointer → Variable) > (Variable)
```

Example

```
// In this example, we have a variable which we compare with a
// variable field that is, for example, written by the PC.

DEF_W     2000,TABELLE1          // Start of the first variable field
DEF_W     3000,TABELLE2          // Start of the second variable field
LAD_VA    V_VERGLEICH,1234       // The contents of the variable V_VERGLEICH are
1234
LAD_VA    V_ZEIGER1,TABELLE1     // Initialize pointer variable1
LAD_VA    V_ZEIGER2,TABELLE2     // initialize pointer variable2

LOOP:
VERG_VI   V_ZEIGER1,V_VERGLEICH  // Perform comparison
LAD_M     M_GLEICH               // Found entry that is equal?
SPRINGJ   FERTIG                 // If yes, jump to label "Fertig"
INC_V     V_ZEIGER1,1            // Table pointer 1 to next entry
INC_V     V_ZEIGER2,1            // Table pointer 2 to next entry
SPRING    LOOP                   // Back to loop
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.

See also

Variable Comparison Instructions (Page 34)

VERG_VA

VERG_VA Variable V, Value K

Group	Depends on BES	Changes BES
Variable Comparison	**No**	**No**

VERG_VA compares the contents of the specified variable with the specified value. The result is stored in the variable comparison flags.

Flag	Is set if ...
M_GLEICH	... the contents of the specified variable are equal to the value
M_KLEINER	... the contents of the specified variable are less than the value
M_GROESSER	... the contents of the specified variable are greater than the value

Table 15 – Variable Comparison Flags for VERG_VA

Operation

```
(M_GLEICH)   ← (Variable) = (Value)
(M_KLEINER) ← (Variable) < (Value)
(M_GROESSER) ← (Variable) > (Value)
```

Example

```
LAD_VA     V_ZEIGER,K_ANFANG_TAB   // Initialize pointer
VERG_VA    V_ZEIGER,2000           // Comparison
LAD_M      M_GLEICH                // If M_GLEICH is on, V_ZEIGER
                                   // has the value 2000
SPRINGJ    FERTIG                  // Jump to label FERTIG
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.

See also

Variable Comparison Instructions (Page 34)

VERG_VI

	Group	Depends on BES	Changes BES
Variable Comparison	**No**	**No**	

VERG_VI Variable V, Pointer V

VERG_VI compares the contents of the specified variable with the contents of the variable referenced by the pointer. The result is stored in the variable comparison flags.

Flag	Is set if ...
M_GLEICH	... the contents of the specified variable and the variable referenced by the pointer are equal
M_KLEINER	... the contents of the specified variable are less than the contents of the variable referenced by the pointer
M_GROESSER	... the contents of the specified variable are greater than the contents of the variable referenced by the pointer

Table 16 – Variable Comparison Flags for VERG_VI

Operation

```
(M_GLEICH)   ← (Variable) = (Pointer → Variable)
(M_KLEINER) ← (Variable) < (Pointer → Variable)
(M_GROESSER) ← (Variable) > (Pointer → Variable)
```

Example

```
// In this example, we have a variable field that is written by
// the PC.  We compare this variable field with a variable.

LAD_VA     V_ZEIGER1,TABELLE1      // Initialize pointer variable 1
LAD_VA     V_ZEIGER2,TABELLE2      // Initialize pointer variable 2
LAD_VA     V_VERGLEICH,1234        // the contents of variable V_VERGLEICH are 1234
LOOP:
VERG_VI    V_VERGLEICH,V_ZEIGER1   // Perform comparison
LAD_M      M_GLEICH                // equality found?
SPRINGJ    FERTIG                  // If yes, jump to label "Fertig"
INC_V      V_ZEIGER1,1             // Set Table pointer 1 to next entry
INC_V      V_ZEIGER2,1             // Set Table pointer 2 to next entry
SPRING     LOOP                    // Back to loop
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.

See also

Variable Comparison Instructions (Page 34)

VERG_VV

		Group	Depends on BES	Changes BES
Variable Comparison	**No**	**No**		

VERG_VV Variable1 V, Variable2 V

VERG_VV compares the contents of the two specified variables. The result is stored in the variable comparison flags:

Flag	Is set if ...
M_GLEICH	... the contents of both variables are equal.
M_KLEINER	... the contents of variable1 are less than the contents of variable2
M_GROESSER	... the contents of variable1 are greater than the contents of variable2

Table 17 – Variable Comparison Flags for VERG_VV

Operation

```
(M_GLEICH)   ← (Variable1) = (Variable2)
(M_KLEINER) ← (Variable1) < (Variable2)
(M_GROESSER) ← (Variable1) > (Variable2)
```

Example

```
VERG_VV    V_TEST1,V_TEST2        // Compares the variables V_TEST1 and  V_TEST2
LAD_M      M_KLEINER              // Test the result of the comparison: is V_TEST1
                                  // less than V_TEST2?
SPRINGJ    Fehler                 // Not allowed! Report error
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.

See also

Variable Comparison Instructions (Page 34)

WART_A...WART_E

		Group	Depends on BES	Changes BES
Program Flow	✸ **Yes**	■ **No**		

WART_A...WART_E

WART_A can be used to program a simple loop. The loop continues to run until the Bit Result Buffer (BES) is turned on.

Operation (during execution of WART_E)

(Program address) ← (Address of the previous WARTE_A instruction)

Example

```
// Programming a loop using WART_A...WART_E

WART_A                          // Start of loop
...                             // Program code
WART_E                          // If the Bit Result Buffer (BES) is turned on,
                                // then continue at next line
                                // If the Bit Result Buffer (BES) is turned off,
                                // then go back to the line containing WART_A

// The example of a loop using WART_A...WARTE_E shown above is the exact equivalent
// of a "normal" loop shown in the following example:

LOOP:                           // Define label
...                             // Program code
SPRINGN LOOP                         // If Bit Result Buffer (BES) on, then
continue
                                // at next line
                                // If Bit Result Buffer (BES) off, then back to
                                // label LOOP
```

Note

- The program code between WART_A and WART_E is executed until the Bit Result Buffer (BES) is on during the execution of the WART_E instruction.

See also

Program Flow Control Instructions (Page 35)

XODER_A

XODER_A Output A

Group	Depends on BES	Changes BES
I/O Instructions	☒ **No**	☑ **Yes**

XODER_A performs a logical exclusive-or (XOR) operation between the state of the Bit Result Buffer (BES) and the state of the specified output. The result of this operation is put into the Bit Result Buffer (BES).

Operation

```
(Bit Result) ← (Bit Result) ^ (Output)
```

Example

```
LAD_A      A_TEST1              // If A_TEST1 is turned on
XODER_A    A_TEST2              // or A_TEST2 is turned on,
SPRINGJ    TESTZYKLUS           // then jump to label TESTZYKLUS
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.

See also

I/O Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

XODER_E

XODER_E Input E

Group	Depends on BES	Changes BES
I/O Instructions	☒ **No**	☑ **Yes**

XODER_E performs a logical exclusive-or (XOR) operation between the state of the Bit Result Buffer (BES) and the state of the specified input. The result of this operation is put into the Bit Result Buffer (BES).

Operation

```
(Bit Result) ← (Bit Result) ^ (Input)
```

Example

```
LAD_E      E_TEST1              // If E_TEST1 is turned on
XODER_E    E_TEST2              // or E_TEST2 is turned on,
SPRINGJ    TESTZYKLUS           // then jump to label TESTZYKLUS
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.

See also

I/O Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)

XODER_M

XODER_M Flag M

Group	Depends on BES	Changes BES
Flag Instructions	☒ **No**	☑ **Yes**

XODER_M performs a logical exclusive-or (XOR) operation between the state of the Bit Result Buffer (BES) and the state of the specified flag. The result of this operation is put into the Bit Result Buffer (BES).

Operation

(Bit Result) ← (Bit Result) ^ (Flag)

Example

```
LAD_M      M_TEST1                // IF M_TEST1
XODER_M    M_TEST2                // or M_TEST2 is turned on,
SPRINGJ    TESTZYKLUS             // then jump to label TESTZYKLUS
```

Note

- This instruction does not depend on the Bit Result Buffer (BES). It is always executed.

See also

Flag Instructions **Fehler! Verweisquelle konnte nicht gefunden werden.** (Page **Fehler! Textmarke nicht definiert.**)
Fehler! Verweisquelle konnte nicht gefunden werden. (Page **Fehler! Textmarke nicht definiert.**)

#ELSE

	Group	Depends on BES	Changes BES
Compiler		☒ **No**	■ **No**

#ELSE

#ELSE is used for testing conditions in connection with conditional compiler runs. Prior to the instruction, a condition is tested with the EQ instruction. If the first dependent directive is not fulfilled, the second dependent directive is executed.

Example

```
DEF_W      3,Achsenanzahl         // Number of axes is equal to 3

#IF Achsenanzahl EQ 2             // Achsenanzahl in this example is equal to 2, so
...                               // the source code after this #IF up to the
...                               // next  #ELSE is not used.
#ELSE                             // Because the condition at #IF was not met,
...                               // the source code under #ELSE is executed
...
#ENDIF                            // #ENDIF marks the end of this conditional test.
From
...                               // this point on, the source code is used again
...                               // regardless of the conditional test.
```

See also

Compiler Directives (Page 28)

#ENDIF

Group	Depends on BES	Changes BES

Compiler	☒ No	■ No

#ENDIF

#ENDIF ends conditional compilation.

Example

```
#IF Achsenanzahl EQ 2              // If Achsenanzahl is 2, then use code
...
...
#ENDIF                             // End of conditional compilation
```

See also

Compiler Directives (Page 28)

IF

Group	Depends on BES	Changes BES
Compiler	☒ **No**	■ **No**

 # IF TestCondition K
 # IF Value1 K EQ Value2 K

#IF is used for testing conditions in connection with conditional compiler runs. Prior to the instruction, a condition is tested with the EQ instruction. If the first dependent condition is fulfilled, the first directive executed. If the first dependent directive is not fulfilled, the second dependent directive (if any) is executed.

Available Logical Operators

The following logical operators are available to you for use with definition instructions:

Operator	Meaning
EQ	"Equal": If the two parameters in the test condition are equal, conditional compilation is carried out.
NEQ	"Not Equal": If the two parameters in the test condition are not equal, conditional compilation is carried out.
GT	"Greater than": If the first parameter is greater than the second parameter, conditional compilation is carried out.
LT	"Less than": If the first parameter is less than the second parameter, conditional compilation is carried out.

■ Table 18 – Logical Operators for Conditional Compilation

Example

```
DEF_W     2,Achsenanzahl          // Achsenanzahl equals 2

#IF Achsenanzahl EQ 2             // Achsenanzahl in this example is 2, so
...                               // the source code following this if #IF up to
...                               // the next #ENDIF or #ELSE is used.
#ELSE                             // Because the condition at  #IF was met, the
...                               // source code under #ELSE is not executed
...
#ENDIF                            // #ENDIF marks the end of this conditional test.
From
...                               // this point on, the source code is used again
...                               // regardless of conditional testing.

// #IF...#ELSE...#ENDIF conditional tests can be nested as follows:

#IF Achsenanzahl EQ 2             // As described above, Achsenanzahl is 2, so
...                               // the source text is used
#IF Greifer EQ 1                       // If a system with a claw is used,
...                               // then use this source code
...
#ELSE                             // If the previous test for the constant Greifer
...                               // failed, then the following
...                               // source code is used
#ENDIF                            // End of the Greifer conditional test
#ENDIF                            // End of the Achsenanzahl conditional test
...                               // The source code is now used again
...                               // regardless of conditional testing.
```

See also

Compiler Directives (Page 28)

INCLUDE

	Group	Depends on BES	Changes BES
Compiler	**☒ No**	**■ No**	

 # INCLUDE SourceCodeFileName K

#INCLUDE inserts an additional source code file at the current location.

Example

```
#INCLUDE Achsen.mc                // The file Achsen.mc is included
```

See also

Compiler Directives (Page 28)

Space for your notes

Space for your notes

Chapter 4 Axis programming

In the MC100 system, axis control is performed entirely via system variables and system flags. This means that:

- For each axis, the range of functions is defined by particular system variables and system flags.
- A function is prepared by describing these variables and flags.
- After all the data required for these functions has been written to the variables and flags, the function can be started by setting the confirmation flag (M_SEND_xxx).
- The state of the axes is then reflected back in the MC100 system via system variables and system flags.

In a PLC program, this is how all of this might look:

```
LAD_VA     V_RAMPE_SMA1,80         // Program the ramp: 80 ms/kHz
LAD_VA     V_FMIN_SMA1,500         // Program the start-stop frequency: 500 Hz
LAD_VA     V_FMAX_SMA1,5000        // Program the travel frequency: 5000 Hz
LAD_VA     V_FAKTOR_SMA1,256       // Do not program a unit  of measure

LAD_VA     V_INDEX_SMA1,10000      // Load travel path with 10000 steps
EIN_M      M_START_SMA1            // Start flag: we want to start axis 1
EIN_M      M_SEND_SMA1             // Trigger programmed function: start
                                   // positioning of axis 1 by 10000 steps
                                   // using velocities specified above

// Before further actions related to one of the axes can be started
// or the status of the axis can be queried, the PLC program must
// ensure that the instructions to move were transferred to the axis.
//That is why we wait here until the flag M_SEND_SMA1 is reset,
// which then means: all data has been transmitted, the axis is starting

LOOP1:
LAD_M      M_SEND_SMA1             // Send flag off = all data sent?
SPRINGJ    LOOP1                   // No, so we wait a bit longer

// Axis 1 is now running, and we simply wait until the in-position flag is
received, or
// in other words until the axis has reached is destination position.

LOOP2:
LAD_M      M_FERTIG_SMA1           // Flag off means: the axis is still in motion
SPRINGN    LOOP2                   // and we wait a little while longer
```

System Variables and System Flags Stepper Motor Axis 1

Flag	Symbol	Function
641	M_SEND_SMA1	Send message is sent to axis 1
865	M_ENDSPL_SMA1	Status end switch +
866	M_ENDSMI_SMA1	Status end switch -
867	M_UEBERW_SMA1	Status input for stepper monitoring
868	M_MOPSW_SMA1	Status for optional reference switch
869	M_RGLBER_SMA1	Status for readiness state of axis control circuit
870	M_FEHLER_SMA1	Flag turns on if an error has occurred during stepper monitoring.
871	M_POSOK_SMA1	Flag on => axis in position Flag off => axis not in position
872	M_FERTIG_SMA1	Flag on => axis stationary Flag off => axis in motion
2493	M_START_SMA1	Start flag for axis 1, is reset by system
2494	M_HALT_SMA1	Axis stops, using programmed break ramp
2495	M_STOP_SMA1	Axis stops immediately without break ramp
2496	M_GETP_SMA1	Read the current position of axis 1
2500	M_INIT_SMA1	Axis 1 initialization flag
2501	M_LAUFSCH_SMA1	Flag causes the axis to move to the reference switch (end switch +)
2502	M_ENDSCH_SMA1P	End switch – becomes reference switch
2503	M_OEFFNER_SMA1	End switches are break contacts
2506	M_LAUFFMIN_SMA1	Axis travels at start-stop frequency
2507	M_HALTSCH_SMA1	Stop axis when end switch + is reached
2508	M_MINUS_SMA1	Reverse axis direction of rotation for a movement
171	V_ABPOS_SMA1	Absolute axis position (= actual position in units of measure)
172	V_INDEX_SMA1	Axis travel in units of measure Positive value = direction of rotation + Negative value = direction of rotation -
173	V_FMAX_SMA1	Axis travel frequency (Hz) 100 – 40000 Hz
316	V_ABSCHR_SMA1	Actual axis position in steps
363	V_FAKTOR_SMA1	Axis conversion factor
364	V_RAMPE_SMA1	Axis acceleration and break ramp (ms/kHz) 4 – 256 ms/kHz
365	V_FMIN_SMA1	Axis start-stop frequency 10 – 1000 Hz
366	V_FEHLSCHR_SMA1	Monitoring steps from pulse to pulse of the monitoring system, for example axis encoder 0 – 256 steps 0 = step monitoring is turned off

System Variables and System Flags Stepper Motor Axis 2

Flag	Symbol	Function
642	M_SEND_SMA2	Send message is sent to axis 2
873	M_ENDSPL_SMA2	Status end switch +
874	M_ENDSMI_SMA2	Status end switch -
875	M_UEBERW_SMA2	Status input for stepper monitoring
876	M_MOPSW_SMA2	Status for optional reference switch
877	M_RGLBER_SMA2	Status for readiness state of axis control circuit
878	M_FEHLER_SMA2	Flag turns on if an error has occurred during stepper monitoring.
879	M_POSOK_SMA2	Flag on => axis in position Flag off => axis not in position
880	M_FERTIG_SMA2	Flag on => axis stationary Flag off => axis in motion
2509	M_START_SMA2	Start flag for axis 2, is reset by system
2510	M_HALT_SMA2	Axis stops, using programmed break ramp
2511	M_STOP_SMA2	Axis stops immediately without break ramp
2512	M_GETP_SMA2	Read the current position of axis 2
2516	M_INIT_SMA2	Axis 2 initialization flag
2517	M_LAUFSCH_SMA2	Flag causes the axis to move to the reference switch (end switch +)
2518	M_ENDSCH_SMA2P	End switch – becomes reference switch
2519	M_OEFFNER_SMA2	End switches are break contacts
2522	M_LAUFFMIN_SMA2	Axis travels at start-stop frequency
2523	M_HALTSCH_SMA2	Stop axis when end switch + is reached
2524	M_MINUS_SMA2	Reverse axis direction of rotation for a movement
174	V_ABPOS_SMA2	Absolute axis position (= actual position in unit of measure)
175	V_INDEX_SMA2	Axis travel in units of measure Positive value = direction of rotation + Negative value = direction of rotation -
176	V_FMAX_SMA2	Axis travel frequency (Hz) 100 – 40000 Hz
317	V_ABSCHR_SMA2	Actual axis position in steps
367	V_FAKTOR_SMA2	Axis conversion factor
368	V_RAMPE_SMA2	Axis acceleration and break ramp (ms/kHz) 4 – 256 ms/kHz
369	V_FMIN_SMA2	Axis start-stop frequency 10 – 1000 Hz
370	V_FEHLSCHR_SMA2	Monitoring steps from pulse to pulse of the monitoring system, for example axis encoder 0 – 256 steps 0 = step monitoring is turned off

System Variables and System Flags Stepper Motor Axis 3

Flag	Symbol	Function
644	M_SEND_SMA3	Send message is sent to axis 3
881	M_ENDSPL_SMA3	Status end switch +
882	M_ENDSMI_SMA3	Status end switch -
883	M_UEBERW_SMA3	Status input for stepper monitoring
884	M_MOPSW_SMA3	Status for optional reference switch
885	M_RGLBER_SMA3	Status for readiness state of axis control circuit
886	M_FEHLER_SMA3	Flag turns on if an error has occurred during stepper monitoring.
887	M_POSOK_SMA3	Flag on => axis in position Flag off => axis not in position
888	M_FERTIG_SMA3	Flag on => axis stationary Flag off => axis in motion
2525	M_START_SMA3	Start flag for axis 3, is reset by system
2526	M_HALT_SMA3	Axis stops, using programmed break ramp
2527	M_STOP_SMA3	Axis stops immediately without break ramp
2528	M_GETP_SMA3	Read the current position of axis 3
2532	M_INIT_SMA3	Axis 3 initialization flag
2533	M_LAUFSCH_SMA3	Flag causes the axis to move to the reference switch (end switch +)
2534	M_ENDSCH_SMA3P	End switch – becomes reference switch
2535	M_OEFFNER_SMA3	End switches are break contacts
2538	M_LAUFFMIN_SMA3	Axis travels at start-stop frequency
2539	M_HALTSCH_SMA3	Stop axis when end switch + is reached
2540	M_MINUS_SMA3	Reverse axis direction of rotation for a movement
177	V_ABPOS_SMA3	Absolute axis position (= actual position in units of measure)
178	V_INDEX_SMA3	Axis travel in units of measure Positive value = direction of rotation + Negative value = direction of rotation -
179	V_FMAX_SMA3	Axis travel frequency (Hz) 100 – 40000 Hz
318	V_ABSCHR_SMA3	Actual axis position in steps
371	V_FAKTOR_SMA3	Axis conversion factor
372	V_RAMPE_SMA3	Axis acceleration and break ramp (ms/kHz) 4 – 256 ms/kHz
373	V_FMIN_SMA3	Axis start-stop frequency 10 – 1000 Hz
374	V_FEHLSCHR_SMA3	Monitoring steps from pulse to pulse of the monitoring system, for example axis encoder 0 – 256 steps 0 = step monitoring is turned off

System Variables and System Flags Stepper Motor Axis 4

Flag	Symbol	Function
644	M_SEND_SMA4	Send message is sent to axis 4
889	M_ENDSPL_SMA4	Status end switch +
890	M_ENDSMI_SMA4	Status end switch -
891	M_UEBERW_SMA4	Status input for stepper monitoring
892	M_MOPSW_SMA4	Status for optional reference switch
893	M_RGLBER_SMA4	Status for readiness state of axis control circuit
894	M_FEHLER_SMA4	Flag turns on if an error has occurred during stepper monitoring.
895	M_POSOK_SMA4	Flag on => axis in position Flag off => axis not in position
896	M_FERTIG_SMA4	Flag on => axis stationary Flag off => axis in motion
2541	M_START_SMA4	Start flag for axis 4, is reset by system
2542	M_HALT_SMA4	Axis stops, using programmed break ramp
2543	M_STOP_SMA4	Axis stops immediately without break ramp
2544	M_GETP_SMA4	Read the current position of axis 4
2548	M_INIT_SMA4	Axis 4 initialization flag
2549	M_LAUFSCH_SMA4	Flag causes the axis to move to the reference switch (end switch +)
2550	M_ENDSCH_SMA4P	End switch – becomes reference switch
2551	M_OEFFNER_SMA4	End switches are break contacts
2554	M_LAUFFMIN_SMA4	Axis travels at start-stop frequency
2555	M_HALTSCH_SMA4	Stop axis when end switch + is reached
2556	M_MINUS_SMA4	Reverse axis direction of rotation for a movement
180	V_ABPOS_SMA4	Absolute axis position (= actual position in units of measure)
181	V_INDEX_SMA4	Axis travel in units of measure Positive value = direction of rotation + Negative value = direction of rotation -
182	V_FMAX_SMA4	Axis travel frequency (Hz) 100 – 40000 Hz
319	V_ABSCHR_SMA4	Actual axis position in steps
375	V_FAKTOR_SMA4	Axis conversion factor
376	V_RAMPE_SMA4	Axis acceleration and break ramp (ms/kHz) 4 – 256 ms/kHz
377	V_FMIN_SMA4	Axis start-stop frequency 10 – 1000 Hz
378	V_FEHLSCHR_SMA4	Monitoring steps from pulse to pulse of the monitoring system, for example axis encoder 0 – 256 steps 0 = step monitoring is turned off

4.1 Program Examples

Relative Positioning of a Stepper Motor Axis

```
LAD_VA     V_RAMPE_SMA1,80         // Ramp = 80 ms/kHz
LAD_VA     V_FMIN_SMA1,500         // Start-stop frequency = 500 Hz
LAD_VA     V_FMAX_SMA1,5000        // Running frequency = 5000 Hz
LAD_VA     V_FAKTOR_SMA1,256       // No unit of measure

LAD_VA     V_INDEX_SMA1,10000      // Load travel path with 10000 steps
EIN_M      M_START_SMA1            // Set start flag axis 1
EIN_M      M_SEND_SMA1             // Send transmit message to axis 1 , axis 1 start
LOOP1:
LAD_M      M_SEND_SMA1             // Send flag off?
SPRINGJ    LOOP1
LOOP2:
LAD_M      M_FERTIG_SMA1           // Flag off = axis in motion!
SPRINGN    LOOP2
```

Absolute Positioning of a Stepper Motor Axis

```
LAD_VA     V_RAMPE_SMA1,50         ;Ramp = 50 ms/kHz
LAD_VA     V_FMIN_SMA1,500         ;Start-stop frequency = 500 Hz
LAD_VA  V_FMAX_SMA1,5000           ;Travel frequency = 5000 Hz
LAD_VA     V_FAKTOR_SMA1,5120      ;Actual position and travel path
                                   ;in units of measure
                                   ;Spindle incline = 5mm
                                   ;Steps per revolution of motor = 1000
                                   ;Resolution = 1/10 mm
SUB_VV     V_SOLLPOS_A1,V_ABPOS_SMA1    ;target minus current position gives offset
LAD_VV     V_INDEX_SMA1,VARERG     ;Load result to travel path
EIN_M      M_START_SMA1            ;Set start flag Axis 1
EIN_M      M_SEND_SMA1             ;Send transmit message to axis 1, axis 1 starts
LOOP1:
LAD_M      M_SEND_SMA1             ;Send flag off?
SPRINGJ    LOOP1
LOOP2:
LAD_M      M_FERTIG_SMA1           ;Flag off = axis in motion!
SPRINGN    LOOP2
```

Home run for a stepper motor axis

```
LAD_VA     V_RAMPE_SMA1,50         ;Ramp = 50 ms/kHz
LAD_VA     V_FMIN_SMA1,500         ;Start-stop frequency = 500 Hz
LAD_VA     V_FMAX_SMA1,5000        ;Running frequency = 5000 Hz
LAD_VA     V_FAKTOR_SMA1,5120      ;Actual position and travel path
                                   ;in units of measure
                                   ;Spindle incline = 5mm
                                   ;Steps per revolution of motor = 1000
                                   ;Resolution = 1/10 mm

;Movement to limit switch +
LAD_VA     V_INDEX_SMA1,10         ;Overrun distance = 1 mm
EIN_M      M_LAUFSCH_SMA1          ;Reference movement to end switch +
AUS_M      M_OEFFNER_SMA1          ;end switch = make contacts
EIN_M      M_START_SMA1            ;Set start flag axis 1
EIN_M      M_SEND_SMA1             ;Send transmit message to axis 1, axis 1 starts
```

```
REF1:
LAD_M      M_SEND_SMA1             ;Send flag off?
SPRINGJ    REF1
REF2:
LAD_M      M_FERTIG_SMA1           ;Flag off = axis in motion!
SPRINGN    REF2

;End switch + free movement
LAD_VA     V_INDEX_SMA1,10         ;Overrun distance = 1 mm
EIN_M      M_LAUFSCH_SMA1          ;Reference movement to end switch +
EIN_M      M_OEFFNER_SMA1          ;end switch = break contacts
EIN_M      M_MINUS_SMA1            ;Change direction of rotation for movement
EIN_M      M_START_SMA1            ;Set start flag axis 1
EIN_M      M_SEND_SMA1             ;Send transmit message to axis 1, axis 1 starts

REF3:
LAD_M      M_SEND_SMA1             ;Send flag off?
SPRINGJ    REF3

REF4:
LAD_M      M_FERTIG_SMA1           ;Flag off = axis in motion!
SPRINGN    REF4
```

Conversion factor calculation

$$\text{conversion factor} = \frac{\text{steps per revolution of the motor}}{\text{way per motor revolution} * \text{resolution}} * 256$$

Example

- Spindle incline: 5 mm
- Steps / revolution: 400 steps
- Resolution = 1/10 mm => 10
  = 1/100 mm => 100

$$\text{conversion factor} = \frac{400 \text{ steps}}{5 \text{ mm} * 10} * 256 = 2048$$

Chapter 5 Memory Allocation

5.1 Flags

The controller has 3548 flags. These flags are available to the programmer and are power fail safe. The flags reserved for system use may only be used for functions defined by the system.

Flag Range	*Contents*
1-499	Available for any use, battery buffered
500	Bit Result Buffer (BES)
501 – 507	Bit Result Buffer Shift Register
508	Variable comparison result equal
509	Variable comparison result less than
510	Flag always on
511 – 542	Timer 1 to 32
543	Reserved for system expansion
550	Controller reset
551 - 560	Reserved for system expansion
561	Variable input active
562	Lamp test
563	Machine key pressed
564	Display variables and text
565	Clear display when entry is made
566	Accept variable from display
567 – 569	Reserved for system expansion
570	Stop task switching
571	Communications error
572 - 579	Reserved for system expansion
580	Release electrical hand wheel
581 - 589	Reserved for system expansion
590	Two NC program simultaneously
591	Error NC program 1
592	Error NC program 2
593	End NC program 1
594	End NC program 2
595	Suppress flag function of the NC program
596 – 600	Reserved for system expansion
601 – 608	Send flags servo axes 1 – 8
609 – 616	Initialization conversion servo axes 1 – 8
617 – 624	Initialization PID regulator servo axes 1 – 8
625 – 632	Send flags temperature regulator MC100-TR (module 1 – 8)
633 – 640	Initialization temperature regulator MC100-TR (module 1 – 8)
641 – 648	Send flags stepper motor axes 1 – 8
649 – 656	Send flags of the analogue I/O cards (module 1)
657 – 664	Send flags of the analogue I/O cards (module 2)
665 – 800	Reserved for system expansion
801 – 808	Status flags of servo axis 1
809 – 816	Status flags of servo axis 2
817 – 824	Status flags of servo axis 3
225 – 832	Status flags of servo axis 4
833 – 840	Status flags of servo axis 5
841 – 848	Status flags of servo axis 6
849 – 856	Status flags of servo axis 7
857 – 864	Status flags of servo axis 8
865 – 872	Status flags of stepper motor axis 1
873 – 880	Status flags of stepper motor axis 2
881 – 888	Status flags of stepper motor axis 3
889 – 896	Status flags of stepper motor axis 4
897 – 904	Status flags of stepper motor axis 5

905 – 912	Status flags of stepper motor axis 6
913 – 920	Status flags of stepper motor axis 7
921 – 928	Status flags of stepper motor axis 8
929 – 936	Error flags MC100-CA (module 1)
937 – 944	Error flags MC100-CA (module 2)
953 – 1999	Reserved for system expansion
2000 – 2099	G functions for NC program
2100 – 2199	M functions for NC program
2200 – 2225	Function flags A-Z for NC program
2226 – 2300	Reserved for system expansion
2301 – 2324	Flags for servo axis 1
2325 – 2348	Flags for servo axis 2
2349 – 2372	Flags for servo axis 3
2373 – 2396	Flags for servo axis 4
2397 – 2420	Flags for servo axis 5
2421 – 2444	Flags for servo axis 6
2445 – 2468	Flags for servo axis 7
2469 – 2492	Flags for servo axis 8
2493 – 2508	Flags for stepper motor axis 1
2509 – 2524	Flags for stepper motor axis 2
2525 – 2540	Flags for stepper motor axis 3
2541 – 2556	Flags for stepper motor axis 4
2557 – 2572	Flags for stepper motor axis 5
2573 – 2588	Flags for stepper motor axis 6
2589 – 2604	Flags for stepper motor axis 7
2605 – 2620	Flags for stepper motor axis 8
2621 – 2628	Flags for MC100-CA (module 1)
2629 – 2636	Flags for MC100-CA (module 2)
2637 – 2652	Reserved for system expansion
2653 – 2700	Flags for keyboard 6 x 8
2701 – 2732	Flags for keyboard 4 x 8 (MC100-PF 4 x 6 only)
2733 – 2780	Flags for keyboard 6 x 8 MC100-BED (module 1)
2781 – 2828	Flags for keyboard 6 x 8 MC100-BED (module 2)
2829 – 2844	Status flags for temperature regulator MC100-TR (module 1)
2845 – 2860	Status flags for temperature regulator MC100-TR (module 2)
2861 – 2876	Status flags for temperature regulator MC100-TR (module 3)
2877 – 2892	Status flags for temperature regulator MC100-TR (module 4)
2893 – 2908	Status flags for temperature regulator MC100-TR (module 5)
2909 – 2924	Status flags for temperature regulator MC100-TR (module 6)
2925 – 2940	Send flags for temperature regulator MC100-TR (module 1)
2941 – 2956	Send flags for temperature regulator MC100-TR (module 2)
2957 – 2972	Send flags for temperature regulator MC100-TR (module 3)
2973 – 2988	Send flags for temperature regulator MC100-TR (module 4)
2989 – 3004	Send flags for temperature regulator MC100-TR (module 5)
3005 – 3020	Send flags for temperature regulator MC100-TR (module 6)
3021 – 3036	Send flags for temperature regulator MC100-TR (module 7)
3037 – 3052	Send flags for temperature regulator MC100-TR (module 8)
3053 – 3068	Status flags for temperature regulator MC100-TR (module 7)
3067 – 3084	Status flags for temperature regulator MC100-TR (module 8)
3085 – 3132	Flags for LEDs
3132 – 3199	Reserved for system expansion
3200 – 3548	Available

5.2 Variables

The controller has up to 16,000 buffered power fail safe variables. The variables reserved for system use may only be used for functions defined by the system.

Variable Range	*Contents*
1	Variable comparison storage for arithmetic and logical variable instructions
2 – 135	Available
136 - 146	Buffer for initialization of the servo axes
147	Actual position of servo axis 1
148	Desired position of servo axis 1
149	Desired velocity of servo axis 1
150	Actual position of servo axis 2
151	Desired position of servo axis 2
152	Desired velocity of servo axis 2
153	Actual position of servo axis 3
154	Desired position of servo axis 3
155	Desired velocity of servo axis 3
156	Actual position of servo axis 4
157	Desired position of servo axis 4
158	Desired velocity of servo axis 4
159	Actual position of servo axis 5
160	Desired position of servo axis 5
161	Desired velocity of servo axis 5
162	Actual position of servo axis 6
163	Desired position of servo axis 6
164	Desired velocity of servo axis 6
165	Actual position of servo axis 7
166	Desired position of servo axis 7
167	Desired velocity of servo axis 7
168	Actual position of servo axis 8
169	Desired position of servo axis 8
170	Desired velocity of servo axis 8
171	Actual position of stepper motor axis 1
172	Index path of stepper motor axis 1
173	Running frequency of stepper motor axis 1
174	Actual position of stepper motor axis 2
175	Index path of stepper motor axis 2
176	Running frequency of stepper motor axis 2
177	Actual position of stepper motor axis 3
178	Index path of stepper motor axis 3
179	Running frequency of stepper motor axis 3
180	Actual position of stepper motor axis 4
181	Index path of stepper motor axis 4
182	Running frequency of stepper motor axis 4
183	Actual position of stepper motor axis 5
184	Index path of stepper motor axis 5
185	Running frequency of stepper motor axis 5
186	Actual position of stepper motor axis 6
187	Index path of stepper motor axis 6
188	Running frequency of stepper motor axis 6
189	Actual position of stepper motor axis 7
190	Index path of stepper motor axis 7
191	Running frequency of stepper motor axis 7
192	Actual position of stepper motor axis 8
193	Index path of stepper motor axis 8
194	Running frequency of stepper motor axis 8

195	Actual position of fast counter 1 MC100-CA (module 1)
196	Actual position of fast counter 2 MC100-CA (module 1)
197	Actual position of fast counter 3 MC100-CA (module 1)
198	Actual position of fast counter 4 MC100-CA (module 2)
199	Actual position of fast counter 5 MC100-CA (module 2)
200	Actual position of fast counter 6 MC100-CA (module 2)
201 – 232	Timer 1 - 32
233	Division remainder
234	Multiplication overflow
235	Counter electrical handwheel
236	Counter electrical handwheel MC100-BED (module 1)
237	Counter electrical handwheel MC100-BED (module 2)
238	Variable entry: variable number
239	Display: number text / variable
240	Variable entry: Format descriptor
241	Display: Format descriptor
242 – 254	Reserved for system expansion
255	Counter for data transfer errors
256	Reserved for system expansion
257	Module 0 (PLC-CPU) type
258 – 289	Module 1 – 32 type
290 – 299	Reserved for system expansion
300	Actual velocity servo axis 1
301	Actual lag servo axis 1
302	Actual velocity servo axis 2
303	Actual lag servo axis 2
304	Actual velocity servo axis 3
305	Actual lag servo axis 3
306	Actual velocity servo axis 4
307	Actual lag servo axis 4
308	Actual velocity servo axis 5
309	Actual lag servo axis 5
310	Actual velocity servo axis 6
311	Actual lag servo axis 6
312	Actual velocity servo axis 7
313	Actual lag servo axis 7
314	Actual velocity servo axis 8
315	Actual lag servo axis 8
316	Actual position of stepper motor axis 1 in steps
317	Actual position of stepper motor axis 2 in steps
318	Actual position of stepper motor axis 3 in steps
319	Actual position of stepper motor axis 4 in steps
320	Actual position of stepper motor axis 5 in steps
321	Actual position of stepper motor axis 6 in steps
322	Actual position of stepper motor axis 7 in steps
323	Actual position of stepper motor axis 8 in steps
324	Actual value A/D channel 1 MC100-AC (module 1)
325	Actual value A/D channel 2 MC100-AC (module 1)
326	Actual value A/D channel 3 MC100-AC (module 1)
327	Actual value A/D channel 4 MC100-AC (module 1)
328	Actual value A/D channel 5 MC100-AC (module 2)
329	Actual value A/D channel 6 MC100-AC (module 2)
330	Actual value A/D channel 7 MC100-AC (module 2)
331	Actual value A/D channel 8 MC100-AC (module 2)
332 – 346	Reserved for system expansion
347	Acceleration of servo axis 1
348	Maximum lag of servo axis 1
349	Acceleration of servo axis 2

350	Maximum lag of servo axis 2
351	Acceleration of servo axis 3
352	Maximum lag of servo axis 3
353	Acceleration of servo axis 4
354	Maximum lag of servo axis 4
355	Acceleration of servo axis 5
356	Maximum lag of servo axis 5
357	Acceleration of servo axis 6
358	Maximum lag of servo axis 6
359	Acceleration of servo axis 7
360	Maximum lag of servo axis 7
361	Acceleration of servo axis 8
362	Maximum lag of servo axis 8
363	Conversion factor of stepper motor axis 1
364	Ramp of stepper motor axis 1
365	Start-stop frequency of stepper motor axis 1
366	Stepper monitor of stepper motor axis 1
367	Conversion factor of stepper motor axis 2
368	Ramp of stepper motor axis 2
369	Start-stop frequency of stepper motor axis 2
370	Stepper monitor of stepper motor axis 2
371	Conversion factor of stepper motor axis 3
372	Ramp of stepper motor axis 3
373	Start-stop frequency of stepper motor axis 3
374	Stepper monitor of stepper motor axis 3
375	Conversion factor of stepper motor axis 4
376	Ramp of stepper motor axis 4
377	Start-stop frequency of stepper motor axis 4
378	Stepper monitor of stepper motor axis 4
379	Conversion factor of stepper motor axis 5
380	Ramp of stepper motor axis 5
381	Start-stop frequency of stepper motor axis 5
382	Stepper monitor of stepper motor axis 5
383	Conversion factor of stepper motor axis 6
384	Ramp of stepper motor axis 6
385	Start-stop frequency of stepper motor axis 6
386	Stepper monitor of stepper motor axis 6
387	Conversion factor of stepper motor axis 7
388	Ramp of stepper motor axis 7
389	Start-stop frequency of stepper motor axis 7
390	Stepper monitor of stepper motor axis 7
391	Conversion factor of stepper motor axis 8
392	Ramp of stepper motor axis 8
393	Start-stop frequency of stepper motor axis 8
394	Stepper monitor of stepper motor axis 8
395	Desired value D/A channel 1 MC100-CA (module 1)
396	Reference value D/A channel 1 MC100-CA (module 1)
397	Desired value D/A channel 2 MC100-CA (module 1)
398	Reference value D/A channel 2 MC100-CA (module 1)
399	Desired value D/A channel 3 MC100-CA (module 1)
400	Reference value D/A channel 3 MC100-CA (module 1)
401	Desired value D/A channel 4 MC100-CA (module 1)
402	Reference value D/A channel 4 MC100-CA (module 1)
403	Desired value D/A channel 1 MC100-CA (module 2)
404	Reference value D/A channel 1 MC100-CA (module 2)
405	Desired value D/A channel 2 MC100-CA (module 2)
406	Reference value D/A channel 2 MC100-CA (module 2)
407	Desired value D/A channel 3 MC100-CA (module 2)

408	Reference value D/A channel 3 MC100-CA (module 2)
409	Desired value D/A channel 4 MC100-CA (module 2)
410	Reference value D/A channel 4 MC100-CA (module 2)
411 – 430	Reserved for system expansion
431	Program number NC program 1
432	Line number NC program 1
433	Line jump NC program 1
434	Error number NC program 1
435	Program number NC program 2
436	Line number NC program 2
437	Line jump NC program 2
438	Error number NC program 2
439	Reserved for system expansion
440 – 465	Format variables NC functions A – Z
466 – 469	Reserved for system expansion
470 – 495	Data NC functions A – Z
496 – 543	Reserved for system expansion
544	Status variable MC100-TR module 1
545	Status variable MC100-TR module 2
546	Status variable MC100-TR module 3
547	Status variable MC100-TR module 4
548	Status variable MC100-TR module 5
549	Status variable MC100-TR module 6
550	Short circuit output 1 – 16
551	Excess temperature output 1 – 16
552	Line break output 1 – 16 (error)
553	Line break output 1 – 16 (Status)
554	Short circuit output 17 – 32
555	Excess temperature output 17 – 32
556	Line break output 17 – 32 (error)
557	Line break output 17– 32 (Status)
558	Short circuit output 33 – 48
559	Excess temperature output 33 – 48
560	Line break output 33 – 48 (error)
561	Line break output 33 – 48 (status)
562	Short circuit output 49 – 64
563	Excess temperature output 49 – 64
564	Line break output 49 – 64 (error)
565	Line break output 49 – 64 (status)
566	Short circuit output 65 – 80
567	Excess temperature output 65 –80
568	Line break output 65 – 80 (error)
569	Line break output 65 – 80 (status)
570	Short circuit output 81 – 96
571	Excess temperature output 81 – 96
572	Line break output 81 – 96 (error)
573	Line break output 81 – 96 (status)
574	Short circuit output 97 – 112
575	Excess temperature output 97 – 112
576	Line break output 97 – 112 (error)
577	Line break output 97 – 112 (status)
578	Short circuit output 113 – 128
579	Excess temperature output 113 – 128
580	Line break output 113 – 128 (error)
581	Line break output 113 – 128 (status)
582 – 587	Actual value temperature channel 1 – 6 MC100-TR (module 1)
588 – 593	Actual value temperature channel 1 – 6 MC100-TR (module 2)
594 – 599	Actual value temperature channel 1 – 6 MC100-TR (module 3)

600 – 605	Actual value temperature channel 1 – 6 MC100-TR (module 4)
606 – 611	Actual value temperature channel 1 – 6 MC100-TR (module 5)
612 – 617	Actual value temperature channel 1 – 6 MC100-TR (module 6)
618 – 623	Desired value temperature channel 1 – 6 MC100-TR (module 1)
624 – 629	Desired value temperature channel 1 – 6 MC100-TR (module 2)
630 – 635	Desired value temperature channel 1 – 6 MC100-TR (module 3)
636 – 641	Desired value temperature channel 1 – 6 MC100-TR (module 4)
642 – 647	Desired value temperature channel 1 – 6 MC100-TR (module 5)
648 – 653	Desired value temperature channel 1 – 6 MC100-TR (module 6)
654 – 659	Actual value temperature channel 1 – 6 MC100-TR (module 7)
660 – 665	Actual value temperature channel 1 – 6 MC100-TR (module 8)
666 – 671	Desired value temperature channel 1 – 6 MC100-TR (module 7)
672 – 677	Desired value temperature channel 1 – 6 MC100-TR (module 8)
678	Status variable MC100-TR (module 7)
679	Status variable MC100-TR (module 8)
680 – 699	Reserved for system expansion
700 – 16000	Available