Table of Contents

1. Introduction to B-CONW for Windows.

B-CONW for Windows is a programming language, designed to allow engineers to create control sequences and configuration software for the System 2000 and RTU8/RTU-COM/RTU-870 range of products from Brodersen Control Systems A/S.

B-CONW can be used by both the engineer, who may not be formally educated in programming, and the experienced programmer alike. B-CONW unlocks the power of the System 2000 and RTU modules, enabling them to perform complex control functions, alarm monitoring and in the case of the RTU8/RTU-COM compact outstation, the logging of data.

The IEC1131 standard for programming software defines a range of software with the basic idea of achieving some common standards being different hardware/software vendors. B-CONW uses the IEC1131-3 Instruction List standard.

The B-CONW software development package is run from a PC, where the programme is created, compiled into code, capable of running in the System 2000 and RTU modules, tested for correct operation, and finally downloaded into the module.

In the development of B-CONW great importance has been attached to achieving an efficient programming cycle, editing, compiling, and testing.

The creation of a programme consists of the following steps.

- The programme is created using the Instruction List format in the editor, to create the source code.
- The source code is then translated into a binary code by the compiler.
- The code can then be tested for correct operation using the Debug facilities.
- The resulting code is then downloaded, via the COM port or via the field bus using the IOTOOL32. (Please see manual for information about IOTOOL32.)

Licence conditions.

Which types of Brodersen modules can be programmed?

The B-CONW can be used to programme Brodersen modules which contain the B-CON facility.

Brodersen modules with B-CON facility:

BITBUS slave modules with I/O with revision numbers 3.00 or higher.
BITBUS slave modules with I/O containing /DL
RTU modules with type numbers containing UCR
REMOTE master modules with type numbers containing /RM

1.1 Installation/Get started.

The B-CONW Programming Tool is a bundled part of the IOTOOL32 Pro software package.
To install the B-CONW you must install the full IOTOOL32 Pro package using the installation programme supplied on the CD ROM.

Please refer to the IOTOOL32 Pro Programme Installation described in the IOTOOL32 Pro User's Guide supplied with the IOTOOL32 Pro Package.

Start up of B-CONW:

B-CONW is started from the I/O Explorer by clicking on the icon B-CON.

2. Programming.

2.1 Data types and formats

A few basic points have to be explained before starting the programming:

- **Specification of data types and formats**
- **Numbering of input and output**
- **Use of symbols**
- **Set up of data which should be available on the serial port of the module for communication with external devices (not RM and BM modules)**

Specification of data types and formats.

B-CONW operates with **3 different data formats**:

-	1 Bit	Basic logic unit	Value range: 0 or 1
B	1 Byte = 8 Bits		Value range: -128 to 127
W	1 Word =2 Bytes=16 Bits		Value range:-32768 to 32767

B-CONW operates with **5 different data types:**

I	Input variable
O or Q	Output variable
M	Memory marker (internal relay/register)
C	Constant

Constants can be entered as **different value types**:

Bit constants can be entered as **1 or 0 without extension**.
Byte/Word constants can be entered as **decimal value without extension or hexadecimal value with extension H.**

Examples:

I	= Bit input
BI	= Byte input
WO	= Word output
C1	= Bit constant value1
BC100	= Byte contant decimal value 100
WC4376	= Word constant decimal value 4376
WC0fffH	= Word constant hexadecimal value 0fff (Decimal 4095)

2.2 Numbering of inputs and outputs

The inputs and outputs of the Brodersen modules with B-CON facility are **separated in 8 types:**

DI	**DO**	Digital inputs/outputs
AI	**AO**	Analogue inputs/outputs
ZI	**ZO**	Aux. inputs/outputs
YI	**YO**	Aux. inputs/outputs (communication registers)

Each **input/output type** has its **own numbering range** for addressing.

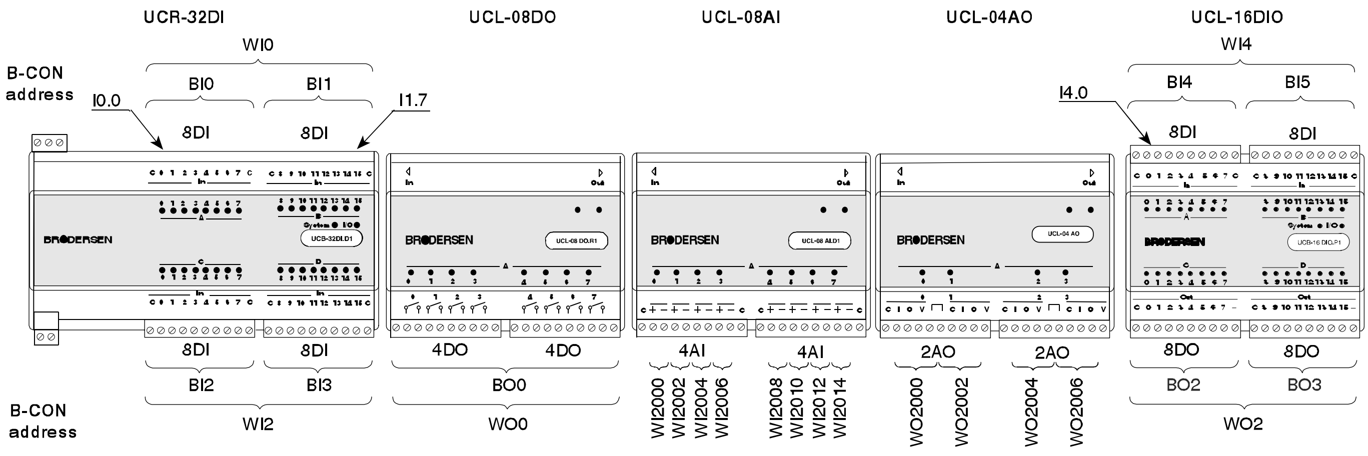
Numbering range for:

DI/DO	0-1999	Bytewise
AI/AO	2000-3999	Bytewise
ZI/ZO	4000-5999	Bytewise
YI/YO	6000-7999	Bytewise

Examples of syntax for addressing:

I0.2	Digital input bit 2 on input byte 0
BO0	Digital output byte 0
WI0	Digital input word 0. **Covers both input byte 0 AND byte 1**
WI2	Digital input word 2. **Covers both input byte 2 AND byte 3**
WI2000	Analogue input 0

Example of an island



2.3 Use of symbols

To ease the overview of a source code, it can be an advantage to use symbols instead of the basic names and numbers described earlier.

A symbol can be a synonym for an input, output address, or a memory marker.
To define symbols, the following syntax has to be used:

```
#define    <symbol> <I/O address or memory marker>
```

Example:

```
#define    pump     i0.3
#define    temp     wi2000
#define    scale    wm22
```

"pump", "temp" and "scale" can now be used in the B-CON programme instead of the standard I/O addresses and memory markers and ease both the overview of writing the B-CON source code and the readability afterwards.

The define statements should be entered **before** the "main:" label. (See 2.5 for “How to build a B-CON programme”)

2.4 Set up of data

Set up of data which should be available on the serial port of the module for communication with external devices (not RM and BM modules).

Communication with the Master (typically a PC).

To communicate with the Master it is necessary to define which data should be accessable from the Master to and from the B-CON slave, Remote slave, or RTU8 via the communication port on the module (Bitbus or Modbus).

The syntax for definition of the accessable data is:

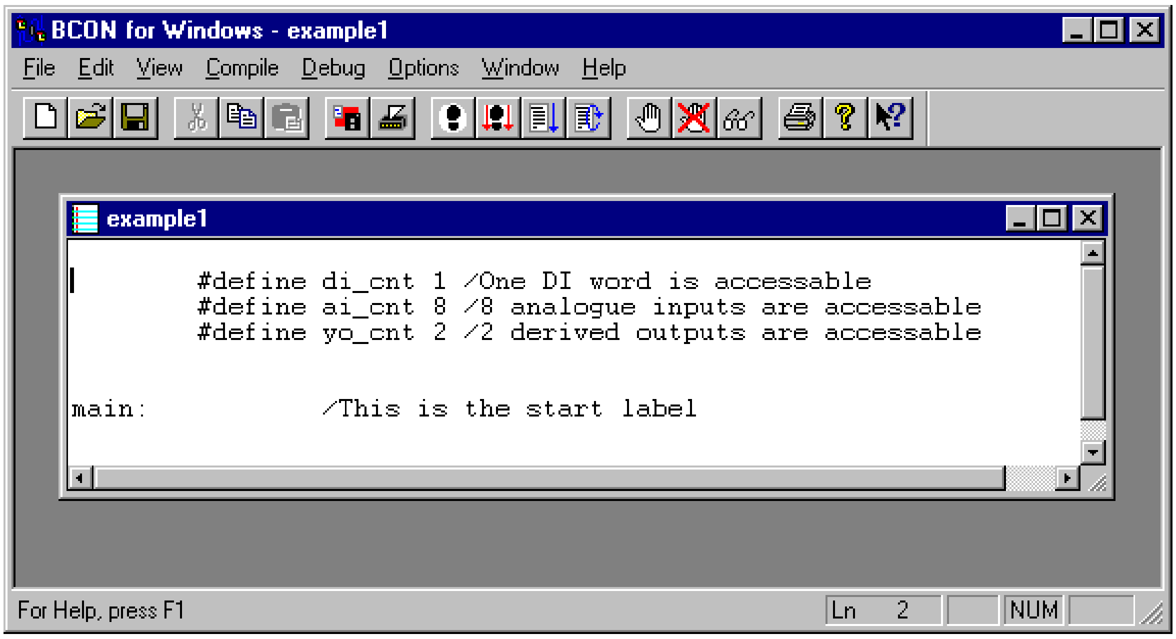
```
#define  <type of data>_cnt <number of words>
```

Example:

```
#define di_cnt 1
#define ai_cnt 8
#define yo_cnt 2
```

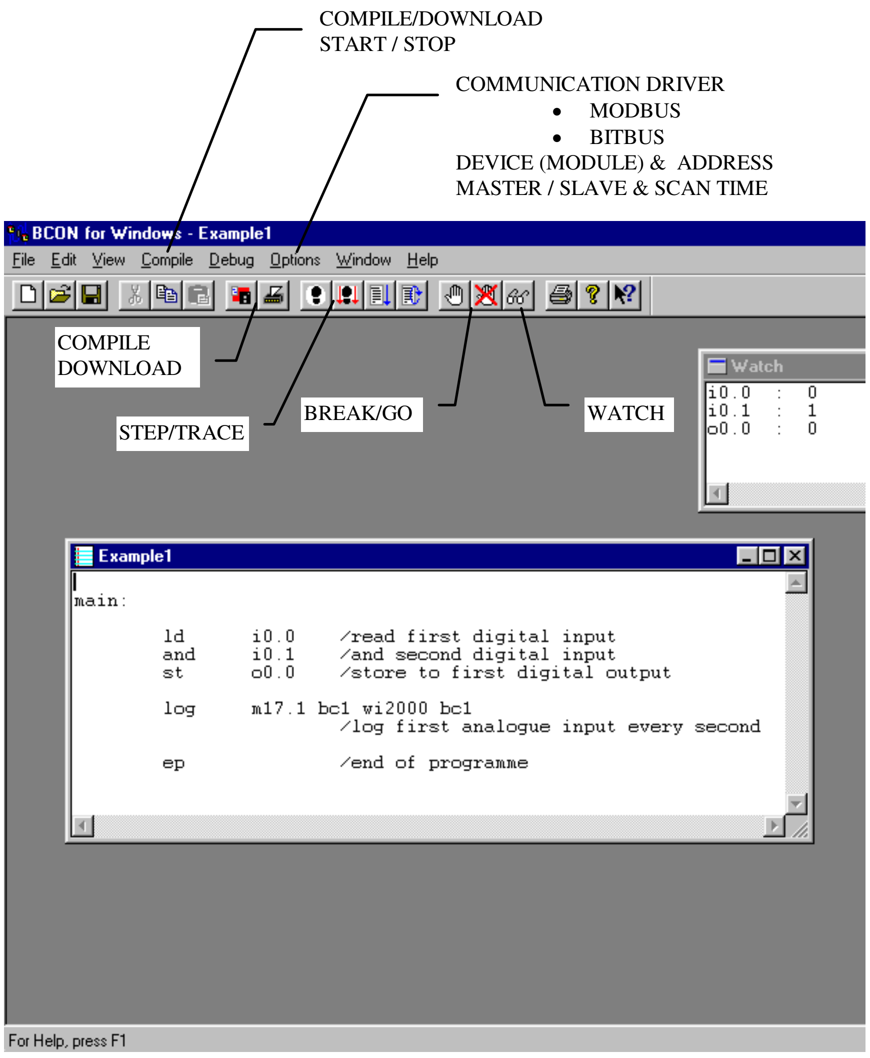
These define statements should be entered **before** the "main:" label. (See 2.5 for “How to build a B-CON programme”)

Example:


BCON for Windows - example1
File Edit View Compile Debug Options Window Help
example1
For Help, press F1
Ln 2
NUM

```
        #define di_cnt 1 /One DI word is accessable
        #define ai_cnt 8 /8 analogue inputs are accessable
        #define yo_cnt 2 /2 derived outputs are accessable


main:           /This is the start label
```

2.5 B-CON Environment/editer

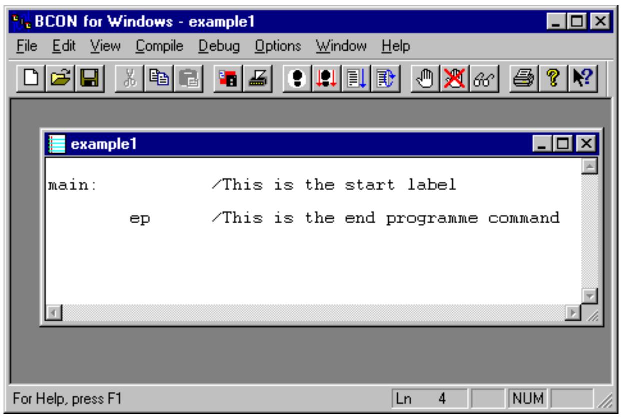


```
i0.0  :  0
i0.1  :  1
o0.0  :  0
```

```
main:

        ld      i0.0    /read first digital input
        and     i0.1    /and second digital input
        st      o0.0    /store to first digital output

        log     m17.1 bc1 wi2000 bc1
                        /log first analogue input every second

        ep              /end of programme
```

How to build a B-CON programme.

Click the icon "File" and choose "New".
You can now start editing the instruction list source code for your B-CON programme.
The label "**main:**" and an end command "**ep**" frames the programme. (See example below).


BCON for Windows - example1
File Edit View Compile Debug Options Window Help
example1
For Help, press F1
Ln 4
NUM

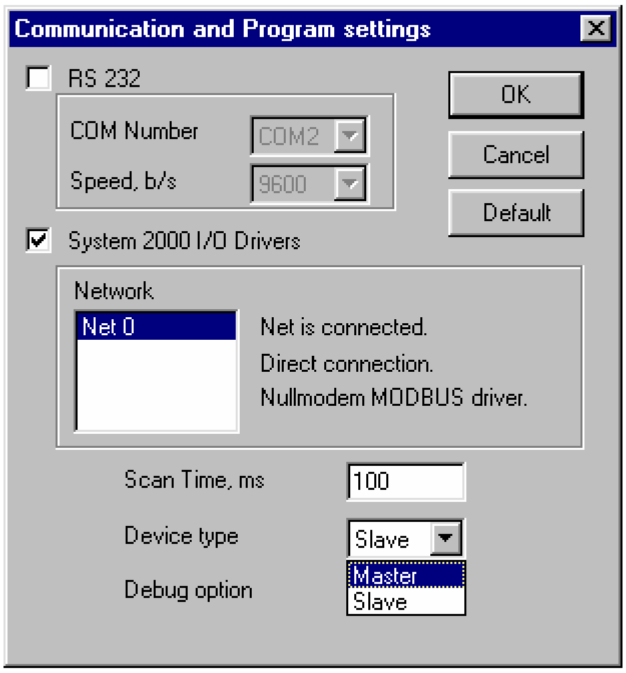
```
main:           /This is the start label

        ep      /This is the end programme command
```

Comments can be added to the source text as documentation for the code.
All comments should start with "/". Any text entered after a "/" in a line will not be compiled to CPU code.

Important features when programming a Master.

When programming a Master you must first make a "Device Table" for the modules which are connected to the Network.

1. Choose the menu **"Options"** and choose **"Communication and Program"**.
   Select "Master" and Click "OK".

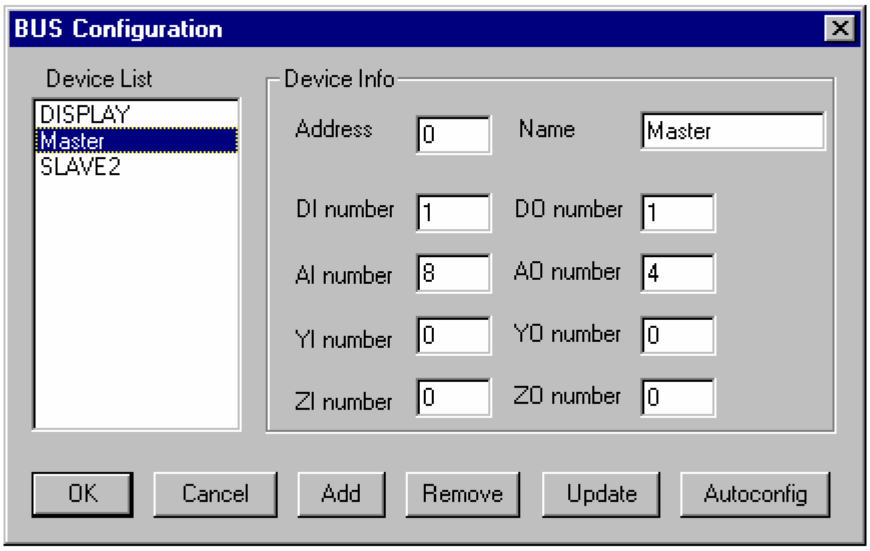


[✓] System 2000 I/O Drivers

Network
Net 0
Net is connected.
Direct connection.
Nullmodem MODBUS driver.

Scan Time, ms: 100
Device type: Slave
Debug option

2. Enter **"Options"** again and choose **"Device"**.
   For the Master and each slave a table has to be filled out. See Example.



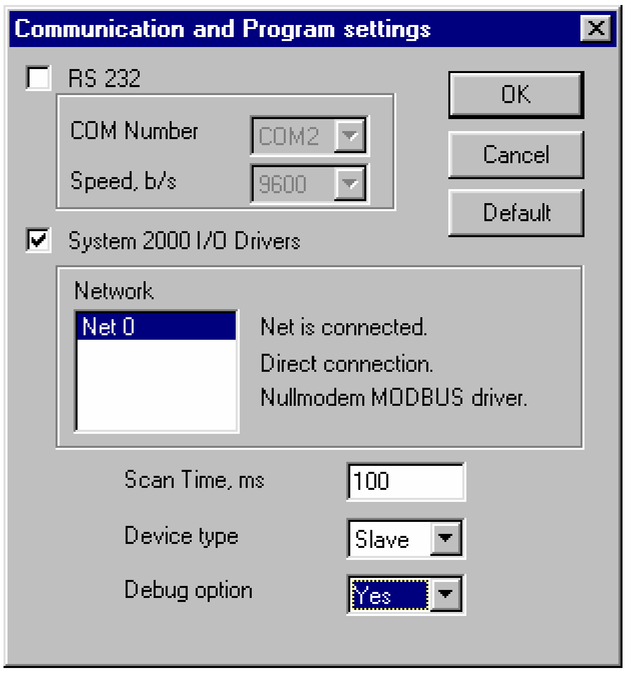
Address	0	Name	Master
DI number	1	DO number	1
AI number	8	AO number	4
YI number	0	YO number	0
ZI number	0	ZO number	0

Please note that the Master **must** have the address No. 0 in the device table. Address switch on the module should be set to 1.
The numbers which must be entered are the number of words of each type.
If all modules are connected you can also let the B-CON programme read the configuration if you press **"Autoconfig"**.

2.6 Debugging a B-CON programme.

To debug your B-CON program you can use the debug facilities available in the “Debug” menu.

Note! Before you enter the “Debug” menu you must choose “Debug Option” in the “Communication and Program” menu under “Option”.


Communication and Program settings

RS 232
COM Number: COM2
Speed, b/s: 9600

System 2000 I/O Drivers ✓

Network
Net 0
Net is connected.
Direct connection.
Nullmodem MODBUS driver.

Scan Time, ms: 100
Device type: Slave
Debug option: Yes

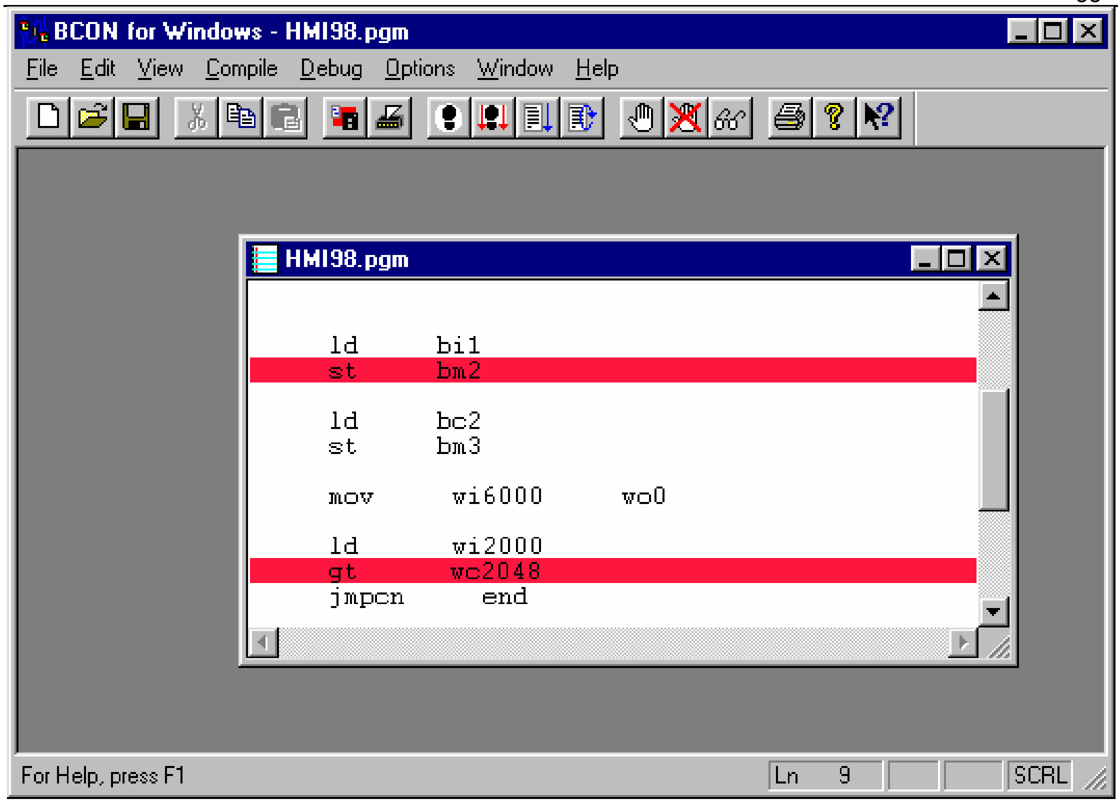
When choosing “Yes” for “Debug option” only a part of the application program will be down loaded to the System 2000 module, and the execution of the application program will be carried out in the PC instead.

In the “Debug” menu you will find different ways of executing the B-CON application:

STEP:	Each time F7 is pressed, the next line in the B-CON program is executed.
TRACE:	When F8 is pressed a slow execution of the B-CON application is started. In “Trace” mode it is possible to follow the execution of the B-CON program line by line.
RUN:	Execution of the program as fast as possible in the PC.
RESTART :	Press shift F2 to restart the execution of the application program.
STOP DEBUGGER:	Stops the Debug function.

To help debugging it is possible to make **break points** in the program to stop the **“Trace”** function at specific lines in the program.

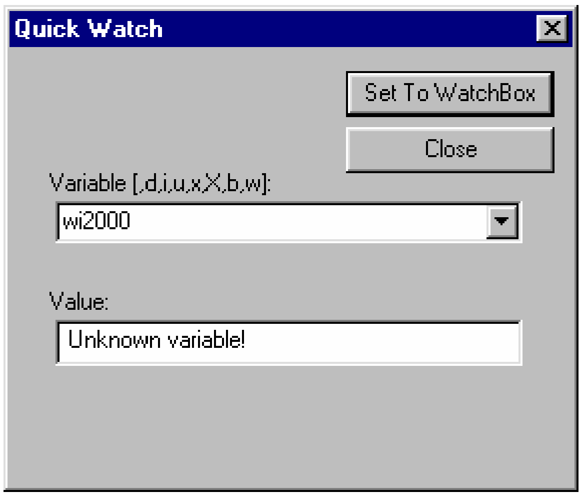
Place the cursor in the line where the **break point** should be and press **F4.**


BCON for Windows - HMI98.pgm
File Edit View Compile Debug Options Window Help
HMI98.pgm
For Help, press F1
Ln 9
SCRL

```
ld      bi1
st      bm2

ld      bc2
st      bm3

mov      wi6000      wo0

ld       wi2000
gt       wc2048
jmpcn      end
```

To monitor the M registers or inputs or outputs it is possible to activate Watch windows which will show the contents of the chosen registers during the execution of the application program.

To set up a Watch window you can either **press shift F7** or select the menu **"Watch"**.



Variable [,d,i,u,x,X,b,w]: wi2000

Value: Unknown variable!

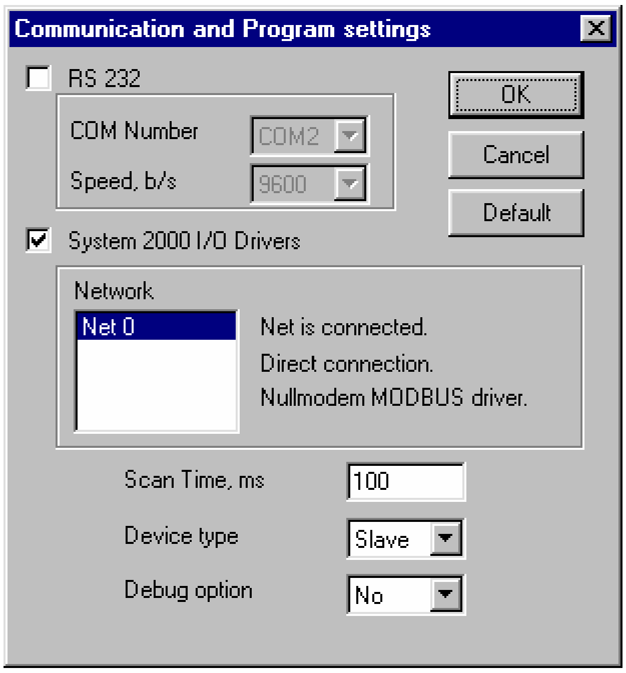
You can now enter the names of the variables you want to watch/monitor as they are placed in a separate window on the screen.

2.7 Compilation and Download.

After finishing the Instruction list programming you should convert the code for use in the Remote Slave, RTU8, or I/O slave module with B-CON facility.

It is carried out if you follow the below procedure:

1. Click the menu **"Option"** then **"Communication and Program"**.



Communication and Program settings

- [ ] RS 232
  - COM Number: COM2
  - Speed, b/s: 9600
- [x] System 2000 I/O Drivers

Network: Net 0
Net is connected.
Direct connection.
Nullmodem MODBUS driver.

Scan Time, ms: 100
Device type: Slave
Debug option: No

2. Choose the right communication way.

   "RS232" is used if you want to **download directly via one of the COM ports** (used if no System 2000 drivers are installed).

   "System 2000 I/O Drivers" is used if you want to **download via one of the IOTOOL32 drivers.**

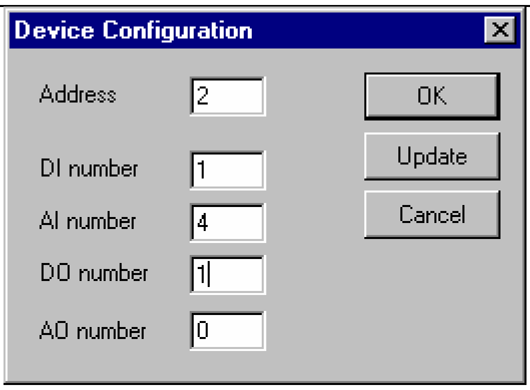
3. Check the settings and press **"OK"**.

4. Choose the right **"Device type"**, **Master** or **Slave** as well as if you want to operate with the **"Debug option"** or not.

5. If you choose **"Slave"** you have to specify the **"Scan time"** for your B-CON programme.

6. Click **"OK"** when the right choices have been made.

7. Before downloading to a **slave**, it is necessary to choose the correct slave **address**.
   Click **"Options"** and enter **"Device"**.
   Enter the address of the slave you want to download to and click **"Update"**.
   Click **"OK"** when the configuration of the Slave module has been read.



Address: 2
DI number: 1
AI number: 4
DO number: 1
AO number: 0

When downloading to a Master, keep the **"Device table"** already defined in chapter 2.5 **"Programming a Master"**.

8. Enter the menu **"Compile"**. Choose the command "Compile".
   Correct the errors if the compiler returns an Error file and repeat the compilation procedure until the programme is error free.

9. When the programme is compiled and free of errors, you can use the **"Down load"** command to transfer it to the module.

10. Use the **"Start"** command to activate the down loaded programme in the module.

3. Programming

3.1 Description of the Language

IEC 1131 is an international standard for the development of PLC controllers and PLC programming languages. It was agreed upon in 1990 and is generally applied for new PLC systems.
The standardization of instructions and command patterns contributes to the fact that programmers do not need to change their familiar programming techniques when a PLC system is changed.

The programming language was developed according to direction IEC 1131 of working group 65A, which sets the standard for programmable controls. The standard comprises 3 types of presentation:

- Structured text
- Instruction list
- Function sets (macros)

In the following description of the **instruction language**, all the available IEC 1131 commands that are supported by B-CON are discussed in detail.

3.1.1 Organization of Programs

A control program consists of a number of instructions. Each instruction is terminated with <RETURN>.
An instruction line can be supplemented by a comment.
In editing a program it must be noted that there must be an indentation of at least one <SPACE> or better a tab before an instruction is written. In this free place the editor expects a jump label.

Commands
An instruction contains an OPERATOR (command) along with an optional MODIFIER, then follows a <SPACE> or a <TAB> and, if necessary, one or more operands separated from each other by commas or <SPACE>.

Comments
A comment can be written either before the beginning of the program or in the instruction after the last element in the line. It is introduced by the sign (/).

Labels
An instruction can be preceded by a label for the identification of this place. A label can have a length of up to 8 characters including the end character. Its first sign must be a letter, not a figure. Special characters are not permitted. The label must be terminated by a colon as end character (:). In the jump instruction, however, this colon must not be contained.

Example, section of the program:

```
        LD IO.1        /load input variable
        JMPC MARK1     /jump target without colon

 MARK1  LD  M1.1       /jump label with colon
        AND M5.5
```

The instruction (instruction line) was preceded by a mark called „MARK1“.

Data types of operands
Operands can take the following data types:

- Input variable
- Output variable
- Memory variable (marker variable)
- Constant

The range of values for a variable depends on the selected operand format (bit, byte or word format).

Formats of operands
Operands can have 3 data formats:

Bit	It has the data length of 1 bit. This data format has 2 values, namely „logical true“ = 1 or „logical false“ = 0.
Byte	It has the data length of 8 bits, i.e. 1 Byte. This data format has a value range between +127 and -128, each value with a sign.
Word	It has the data length of 16 bits, i.e. 2 bytes. This data format has a value range between +32767 and -32768, each value with a sign.

Define symbols
The application programmer can define symbolically his own operand characteristics for variables and constants via the operator #DEFINE. Thus he produces a relation to the process to be operated. In this way the control program becomes more comprehensible.

It is a common practice to place DEFINE directives at the beginning of the program. Thus the programmer has good control of the process inputs, process outputs and the assignment to markers. DEFINE directives can also be given within a program. This, however, is not recommendable.

Examples:

```
#DEFINE terminator switch_S1   I1.1/assignment input bit I1.1
#DEFINE overload forward current_F1   I1.2/assignment input bit I1.2
#DEFINE pump_M1 O2.0 /assignment output bit O2.0
```

Application programming commands
The commands of the programming language according to IEC 1131 are described in detail. For each command a sample program of general validity was developed referring in its application to a control module with input and output ports.

In order to access control modules from the B-CON card a reference protocol has to be defined in the program head declaring the symbolical name of the device and the name of the module control program as illustrated above.
When inputs, outputs or markers of a module are to be operated from the master card, the symbolical name of the device must always precede the variable definition separated by the point operator.

In the following the changes are shown necessary to transform a program of general validity (B-CON program) with module-integrated inputs and outputs into a program capable of running under B-CON M:

B-CON sample program:

```
/P_AND.PGM
/Example: AND conjunction in bit format

ld   i0.1   /OP1: load input
and  i0.2   /OP2: AND conjunction OP1 with OP2
st   o0.0   /RES: store in output
ep          /end of program
```

Logic instructions
The current programming language according to the directions of IEC 1131 provides an extended set of instructions which, contrary to other PLC languages, permits instructions with nesting and the application of operands as stack.

The nesting rule consists in an instruction which represents the result of operand 1, operator (command) and operand 2.
The general nesting rule is determined as follows:

Result:= operand 1 operator operand 2

<ACC>   <ACC>

Example: (after preceding DEFINE directives!)

```
LD   OPERAND1   /load OPERAND1 in ACC
AND  OPERAND2   /AND nesting with OPERAND2
ST   RESULT     /output of result
```

Operand 1 must principally be loaded into the ACC before the execution of a command or it is deliberately used as the result of a previous operation.

According to the instruction this contents of the ACC is then connected with operand 2 and kept in the ACC as the new result.

Stack operations (without operand signifies)
Instructions (operations) with stack operands are a special option in the programming language at hand.
When there is no following operand or only a format signifies in an instruction, it is understood that this operand (operand 2) is in the stack and a connective of the contents of the ACC and the stack is produced. Such instructions are permitted in bit, byte and word format.
The allocation of the operands (which is which?) differs from the general pattern of instructions (sequence of operand allocation) in operations with a stack operand.

Example: (after previous DEFINE directives!)

```
LD   OPERAND1  /load OP1
LD   OPERAND2  /load OP2 as stack operand
AND  B         /AND connective OP1 and OP2 as stack
 operand
ST   RESULT    /output of the result
```

Instructions for the different formats

Bit format	AND	/without operand signifier
Byte format	AND B	/operand sifnifier B=byte
Word format	AND W	/operand signifier W=word

Operator modifiers
Operator modifier „N“ refers to the boolean negation of operand 2, which is carried out before the execution of the instruction.

Example: (after previous DEFINE directives!)

```
LD      OPERAND1    /load OPERAND1 into ACC
ANDN    OPERAND2    /AND connective with OPERAND2,
                    /which is negated before the operation
ST      RESULT      /output of the result
```

Jump label modifiers
Jump operators can be modified with modifiers „C“ and in combination with „CN“. Modifier „C“ means that the jump instruction is carried out when the result of the connective is „logical true“, i.e. the boolean value is „1“ (contents of the ACC).

Modifier „CN“ means that the jump instruction is carried out when the result of the connective is „logical false“, i.e. the boolean value is „0“ (contents of the ACC).

Examples:

```
JMPC  MARK1/jump to MARK1 when
          /ACC contents is „1“
          /i.e. condition „TRUE“
JMPCN MARK1/jump to MARK1 when
          /ACC contents is „0“
          /i.e. condition „false“
JMP   MARK3/jump to MARK3 unconditional
 /i.e. without any condition
```

End of program
Every program has to be terminated with the instruction „EP“.

Macros

Macros are defined sequences of commands which are similar to the operation of functions. They are invited by a macro name with arguments as pass parameters. Different from functions they do not put out a return code.
A macro is not an invitation of special function routines but it is created by the internal combination of different B-CON command sequences (B-CON operators).
A macro can be compared to a subprogram (subroutine) invited by the main program and passing on the necessary parameters.

The general invitation of a macro is declared as follows:

```
M_name[(][ARG1[,arg2[,...,argn]]][)]
```

Explanations:

M_name	macro name to address and operate the macro
arg1,arg2 ... argn	macro arguments prescribed by the corresponding macro
[...]	elements in these brackets are optional arguments of a macro

Include

The include directive is used to include a files in current program position.

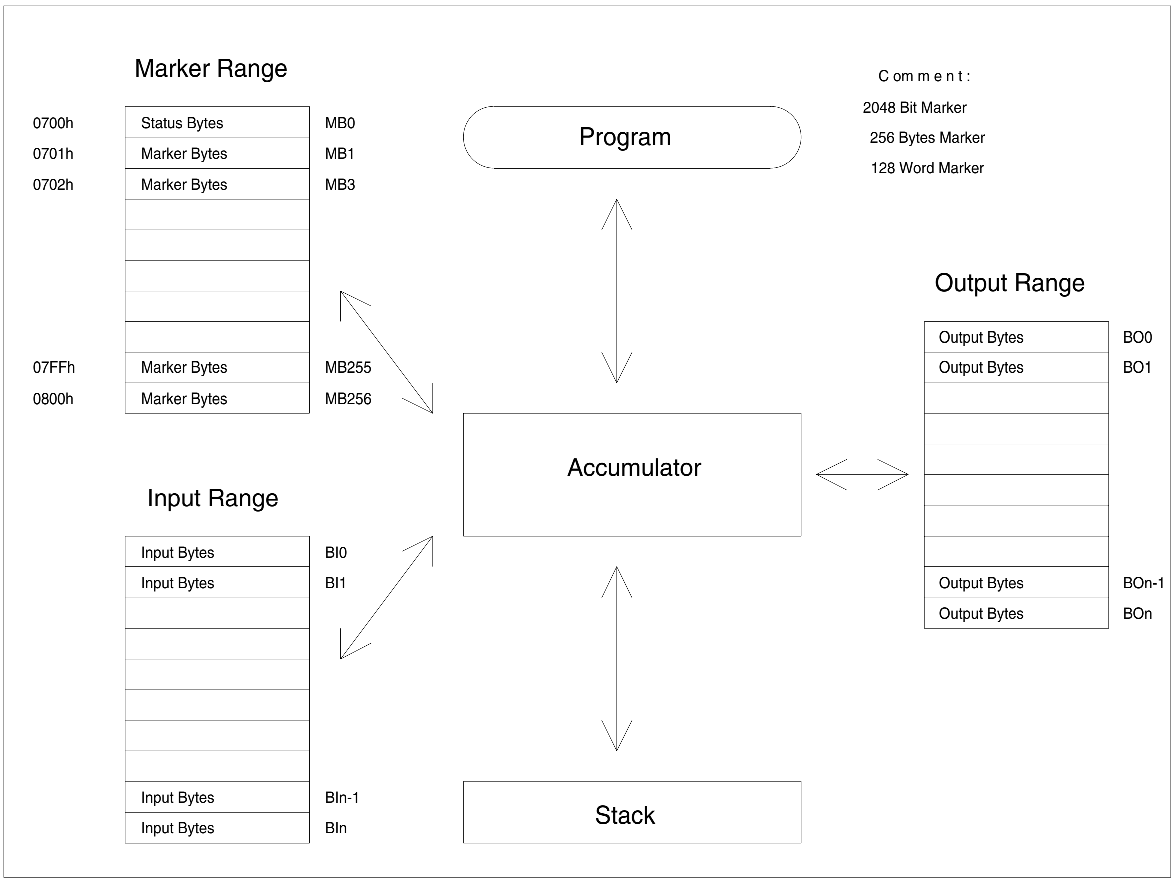
Using “#INCLUDE” files with program, definitions or data can included in the user program. The parameters, defined in the file, that is included can be used as a standard parameters.
It is impossible to debug the file, included in the program. However, the values of parameters can be changed and watched.

Examples:

```
#include PrgmFl.pgm
#include DataFl.pgm
```

3.1.2 Elements for Program Control

In order to define how the control is to work, the handling of the programming language and the concept of the control must be known.
Concerning the development of the user program it must be reagarded that the control consists of the following main elements:



Accumulator
The accumulator is a logical unit wich consists either in loaded operand 1 or in the result after the execution of an instruction.
According to the data format used it has to be noted that the ACC has a format length of 8 or 16 bits.
The format length of 1 bit is represented by 1 byte, only bit Nr 0 being evaluated and bits 1 to 7 remaining unregarded.

Stack
The stack, also called nesting storage or push-down store, is a logical unit which stores results or operands temporarily.
The stack has a fixed data length of 8 bits. During the execution of the program a total of 8 bytes can be stored in the stack.

The operating mode of the stack is according to the LIFO principle (last in - first out), i.e. that the value stored last is unloaded first.
Cooperation with the ACC takes place automatically, e.g. by loading or unloading with each loading or storing.
Physically the stack is organized in such a way that its beginning is with the address assigned uppermost. Its symbolical address 1 thus lies on the storing address assigned uppermost and its symbolical address 8 lies on the lowest storing address.

With a load command the latest contents of the ACC is pushed into the stack and the stack address is decremented.
With store commands the stack contents loaded last is transported back into the ACC and the stack address is incremented.
The format length of 1 bit principally takes byte format, i.e. 8 bits. Only bit 0 is evaluated and bits 1 to 7 remain unregarded.

Inputs/Outputs

Inputs and outputs are process input signals which correspond to the control module applied. Their values are copies of the hardware input/output channels.

They can be defined in the following operand formats:

Format	Preset Identification
Bit	Without
Byte	B
Word	W

I/O addresses

In B-CON the 4 different input/output types are accessed by means of different address ranges as followed:

I/O Type	System 2000 reference	Address Range
Digital	DI/DO	0000 - 1999
Analog	AI/AO	2000 - 3999
Cnt/GW/etc	ZI/ZO	4000 - 5999
Derived	YI/YO	6000 - 7999

The actual useable addresses relates of cause to the actual physical installation (number and type of expansion modules etc.) and the defined registers for communication with the master (YI/YO in B-CON S see below).

ZI/ZO variables are normally not available in BCON-S.
In order to write YIxx variable an instruction ST WOxx should be used.
In order to read YOxx variable an instruction LD WIxx should be used.

In order to make local Inputs available to the master, their number should be defined using “#DEFINE” directive.

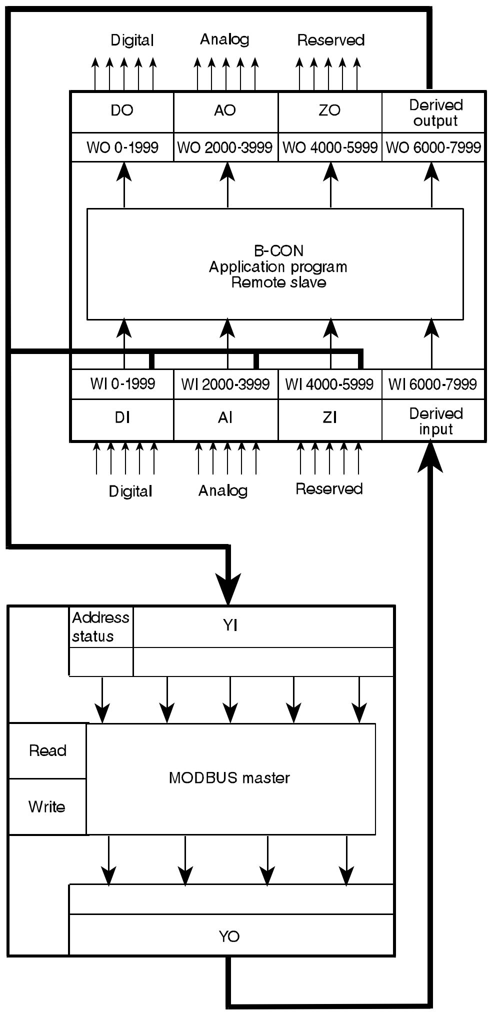
Example:

```
#DEFINE DI_CNT 1        // Master can read 1 DI Word
#DEFINE AI_CNT 2        // Master can read 2 AI Words
#DEFINE YI_CNT 16       // Master can read 16 YI Words
#DEFINE YO_CNT 1        // Master can write 1 YO Word
```

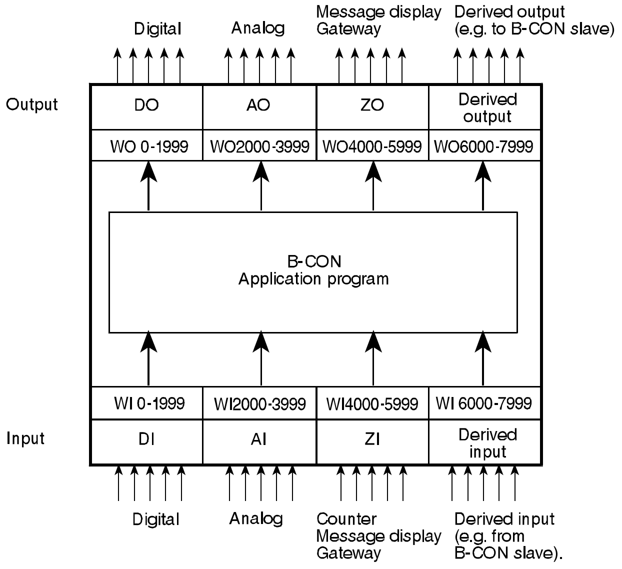
If the actual installation contains more inputs than specified only the lowest inputs will be transferred, e.g.: DI_CNT 1 will transfer first 16 inputs (word 0) whereas following inputs will not be transferred.

The user cannot define Digital and Analog outputs to be available to the master as only the local application program in the slave can control the outputs.
In case outputs should be controlled “directly” from the master the programmer must copy the data form YO (LD wi6000..) to DO (ST wo0...).

B-CON S Database interface/register layout



B-CON M Database interface/register layout



B-CON registers are numbered in units of bytes whereas elsewhere in SYSTEM 2000 the general unit is words, example:

System 2000	B-CON	
DI 0	BI0 BI1	WI 0
DI 1	BI2 BI3	WI 2
DI 2	BI4 BI5	WI 4

Memory variables (Markers)
Markers are storage cells in the internal memory of the control module applied with a memory address of fixed allocation.

Thus are available:

512 byte markers or
256 word markers or
4096 bit markers.

The address allocation of the markers is organized by the operating system according to the following convention:

Marker byte	Memory address
BM0	Error/overflow
BM1	System indicator
BM2	Internal registers
BM3 BM4 BM5	Reserved for modem control (B-CON RM, RS and RTU8).
BM6 . . BM19	Used by RTU8 modules
BM20 . . BM511	(can be used as bit, byte or words)
BM512 . . BM1024	(can be used as bit, byte or words, RTU8 only)

Please observe that the first 2 memory bytes (BM0 and BM1) is reserved for error indication, warnings and special facilities. They must therefore NOT be used as normal internal registers.

The operand formats can be selected in the same way as with inputs and outputs:

Format	Preset Identification
Bit	Without
Byte	B
Word	W

Constants
Constants enable the presetting of fixed variables for operands.
They are also available in the usual operand formats:

Format	Syntax	Value range
Bit	Without	log. „0“ or „1“
Byte	B	-128 0 +127
Word	W	-32768 0 +32767

The corresponding value range always has to be adjusted to the further application (bit, byte or word operation).

Operand calls
The programming language uses 3 labels identities for operands calls, i.e. operand addresses:

- operand format (bit, byte, word)
- operand type (input, output, marker or constant)
- operand counting number in the control system

Format	Syntax
Bit	Without
Byte	B
Word	W

Data type	Syntax
Input variable	I
Output variable	O or Q
Marker (storage variable)	M
Constant	C

As constants can take different data value types, they must be stated by an extended label. This label is placed after the address C.

Value type	Extended label
Binary value	Default boolean integer
Decimal value	Defaults
Hexadecimal	H

Examples:

```
c1          = binary constant of boolean value „1“
c0          = binary constant of boolean value „0“
bc127       = byte constant of decimal value +127
bc-128      = byte constant of decimal value -128
bch0A       = byte constant of value 0Ahex (decimal 10)
bch0  FF    = byte constant of value FFhex (decimal 255)
```

The addresses of operands and operators can be written in small, capital or mixed characters in the source program.
For reasons of clearness using small or capital characters is recommended.

Examples of correct generation of operand addresses:
The following examples show the generation of correct operand addresses:

Operand	Address	Explanation
1.2	input bit variable	input 1, bit No 2
I0.7	input bit variable	input 0, bit No 7
Bi3	input byte variable	input 3
Wi	input word variable	input 0
o1.3	output bit variable	output 1,bit No 3
Q0.5v	output bit variable	output 0, bit No 5
Bo2	output byte variable	output 2
m1.2	bit memory variable	marker 1, bit No 2 (0701hex)
BM3	byte memory variable	marker 3 (0703hex)
wM31	word memory variable	marker 31 (0731hex)
Wm11	word memory variable	marker 11 (0711hex)
C1	bit constant	boolean value 1 (TRUE)
c0	bit constant	boolean value 0 (FALSE)
bc12	byte constant	decimal value 12
bch0a	byte constant	hex value 0A (decimal 10)
wc-1234	word constant	decimal value -1234

Examples of false operand addresses:

Operand	Address	Explanation
1.2		Missing parameter for operand
I0.8	Bit	No greater than 7
bm1.2	Byte	Format with bit marker
dBM3	Undefined	Marker "d"
C11	Bit	Constant grater than 1
bc 128	Byte	Constant greater than 127
wC123.4	Word	Constant with bit marker

Generation of instructions
The generation of an instruction is carried out by selecting an operator followed by an operand as parameter.
The format of the selected operand defines the bit range in which the operation is to be executed. As you know all operand types (inputs, outputs, markers and constants) can also take all formats (bit,byte or word).
As everywhere, there are exceptions here,too, i.e. with some operators (commands) only certain operand formats are permitted.

instructions for bit processing (1-bit length)

AND C0	/AND conn. ACC with bit constant, value log. 0
AND	/AND conn. ACC with stack
AND I1.1	/AND conn. with input bit 1.1

Instructions for byte processing (8-bit length)

AND BM1	/AND conn. ACC with marker byte 1
AND B	/AND conn. ACC with stack
AND BO2	/AND conn. with output byte 2

Generation of instructions for word processing (16-bit length)

AND WM3	/AND conn. ACC with marker word 3
AND W	/AND conn. ACC with stack
AND WI1	/AND conn. with input word

List of commands
In the following list of commands all the commands (operators) are listed with a short description also mentioning the legal operand formats. Derivative commands such as commands in subordinating brackets are not listed. Their handling is illustrated in the corresponding detailed descriptions.

Command	Format	Description
LD LDN	all all	operand is loaded into ACC operand is negated and loaded intoACC
ST STN	all all	ACC contents are loaded into operand ACC contents are negated and loaded into op.
S R	bit bit	operand is set „1“ storing operand is reset „0“ storing
AND/& ANDN/&N	all all	log. AND conjunction, result in ACC OP2 negated, log.AND conjunction, result in ACC
OR OR	all all	log. OR conjunction, result in ACC OP2 negated, log. OR conjunction, result in ACC
XOR XORN	all all	log. XOR conjunction, result in ACC OP2 negated, log. XORN conjunction,result in ACC
JMP JMPC JMPCN	bit bit bit	unconditional jump to label jump to label if condition log. 1 jump to label if condition log. 0
ADD SUB MUL DIV	byte, word byte, word byte, word byte, word	add operation, result in ACC subtract operation, result in ACC multiply operation, result in ACC divide operation, result in ACC
ROR ROL	byte, word byte, word	rotate ACC contents from „High“ to „Low“ by n digits rotate ACC contents from „Low“to „High“ by n digits
GT GE EQ LE LT	byte, word byte, word byte, word byte, word byte, word	compare signed to >, result: Boolean value in ACC compare signed to =>,result: Boolean value in ACC compare signed to =,result: Boolean value in ACC compare signed to <=,result: Boolean value in ACC compare signed to <,result: Boolean value in ACC
UGT UGE UEQ ULE ULT	byte, word byte, word byte, word byte, word byte, word	compare unsigned to >, result: Boolean value in ACC compare unsigned to >=,result: Boolean value in AC compare unsigned to =, result: Boolean value in ACC compare unsigned to <=,result: Boolean value in ACC compare unsigned to <,result: Boolean value in ACC
DUP DUPN	all all	duplicate ACC contents into stack duplicate negated ACC contents into stack
NOP EP	_ _	no operation (no command) end of program

List of functions (macros)
In the following list of macros all the macro notations are listed with a short task description. Their operation is illustrated in the corresponding detailed descriptions.

Macro Notation	Short Description
RTR STR	RS trigger, reset dominant RS trigger, set dominant
CNTUP	counter up counting range: 0..+.32767 max. frequency: 35 Hz
CNTDN	counter down counting range: +32767...0 max. frequency: 35 Hz
CNTUD	counter up - down counting range up: 0...+32767 counting range down: +32767...0 max. frequency: 35 Hz
UPLS DNPLS	positive edge recognition, creates single output pulse negative edge recognition, creates single output pulse
DTR	data trigger, synchronises input signal with system clock max. frequency: 35 Hz
MEQ	Masked equal
LIM	Inlimit check
MOV MVM	Move value Move masked
COPYB COPYW	Move multiple bytes Move multiple words

Timers

Function Notation	Short Description
TMRx	universal timer function x = Timer No number: max. 4 byte argument 0 = timer as edge recognition byte argument 1 = timer as on delay byte argument 2 = timer as off delay

Special functions

Function Notation	Short Description
LOG	LOG instruction (RTU8 only)
CODE	Insert assembler code in program
FUNCTION/ SUBROUTINE	Subroutine

3.1.3 Compiler Errors

According to the PLC model and the control program developed for it, different errors are checked.
The essential error checks can be divided into:
- syntax errors
- semantic errors

Syntax errors
Syntax errors are errors which are produced in connection with failures to comply with syntax rules.
A typical instruction line looks like this:

[LABEL:] instruction code [Operand]

The brackets show that marker definitions (labels) and operand definitions are optional in an instruction.

Labels start in the first column position of a line.
The following syntax errors can occur in compiling in the case of false label definition:

- Label doesn't begin with a letter;
- Label is longer than 8 symbols;
- Label ended without „:" (colon);
- Duplicated label etc.

The instruction code (operator code) is used to define an operation in the program line. As instruction code any of the operators stated in the stock of commands can be applied. During the compiler operation all the commands are checked for correct spelling. Furthermore all the parameters of instructions and functions are checked.
The following syntax errors can be displayed:

- Undefined variable;
- No place PREFIX; (no identifier for variable type stated)
- Too big input (output, memory) number; (input, output or marker have too big an allocation number, which exceeds the PLC control)
- Too big BIT number;
- No bit number for BIT type variable;
- Incorrect BYTE (WORD) type variable with bit number:
- BIT (Byte, word) constant of range etc. (bit, byte or word constant exceeds the value range).

Semantic errors
Semantic errors are errors that occur in connection with the false application of commands and operands.

During the compiler operation the following conditions are checked:

-Is the command compatible with the operand type according to the table of commands?
-Does the data type in the ACC correspond to the data type of the command currently compiled?
-Does the data type in the stack correspond to the data type of the current command (if the operation needs a value from the stack)?
- Is the stack full?
- Is the stack empty (if the operation needs a value from the stack)?
- Has the jump operation to the jump label been executed (if the data in the ACC and in the stack are of different data types)?

A comprehensive evaluation of the compiler operation is filed in the „LST FILE" of the compiler. This file can be watched with the B-CON Editor or with any other text editor.

Diagnostic of delay inequalities
The following statements refer to the adapter card that produces the connection between master and slave in the BITBUS system. The check of delay inequalities is implemented. If a program runs correctly, the red LED only blinks for a short time for initializing a processor test.

List of Error Reports (in alphabetic order)

Argument Expected
The instruction expects an argument

Conditional Jump Expected BIT Result
"JMPC" or " JMPCN" expects an operation result of BIT type in the ACC

Define Function Reqiures Two Operands
The second operand is missing in the function definition

Error Code
An unknown command was used

Example

```
  label:      WEISSNICHTm33
```

Error Opening Debug File
Error Opening Listing File
Error Opening Temporary File

B-CON cannot create a file. The reason may be a defective harddisk or there is no more memory space on the harddisk.

Extra Parameters in xx
There are too many parameters in the instruction.

Example:

```
  ld  bm23 bm24    //illegal!
```

Function Name Expexted
In a function invitation the name of the function to be invited was not declared.

Function xx Is Incorrect Used In xx Module+
Some functions can only be applied to certain modules.

Incorrect Argument Type In ??? For Functions!!
In a function invitation there was a parameter of a false type.

Incompatible Constant Operation With xx Instruction
It was attempted to apply an illegal operation to a constant.

Incompatible Byte Operation With xx Instruction
It was attempted to apply an illegal operation to a byte type. variable.

Incompatible Word Operation With xx Instruction
It was attempted to apply an illegal operation to a word type variable.

Incorrect Argument In xx Operation
An argument of this type cannot be used in this operation.

Incorrect Use Of) Operation
The opening bracket „(„ before this instruction is missing.

Incorrect Use Of xx Variable In xx Operation
Data of the type declared cannot be used in the operation declared.

Jump Requires Label
In a jump instruction no jump label was declared.

Jump To The Label From Line xx With Incorrect Stack Length
After the jump not all the data in the stack were processed.

Example:

```
   ld   bm23
   ld   bm24
   ld   bm25
   jmp  error
   ld   bm26
   ld   bm27
error
```

Jump To The Label From Line xx With Incorrect Status
After a jump a stack operation was carried out that did not correspond to the data type in the stack.

Label Must Begin With Letter

Label Too Long
A jump label must begin with a letter and can have a maximum length of 8 symbols.

MACRO xx Is Incorrect Used In Module
Some macros can only be used with certain devices.

MACRO Requires More Parameters
The macro was invited with too few parameters.

No Closed Comment
Cf. „Unclosed Comment“

No Free Space In Memory
This error message appears when there is not enough memory space at the start of B-CON. Please check if programs resident on your PC are active.

No Such Code
The code used does not exist.

No Variables In Stack
It was attempted to read variables from an empty stack.

Parameter Addressing In xx
The parameter list for a function or a macro is incorrect.

Stack Is Full
The stack depth is limited to 8 bytes. The error message appears when there is an attempt to store more than 8 bytes in the stack.

Stack Overflow
It was attempted to store more than 8 bytes in the stack.

Stack overflow In MACRO
During the execution of a macro it was attempted to store more in the stack than there is space for.

Too Much Labels
This error message appears when more than 200 jump labels were used. The number of jump labels is limited to 200.

Type Of Operands
The operand in the instruction is incompatible with the operand in the stack.

Example:

```
  ld   bm23      //illegal!
  and  wc33
```

Type Of The Value In The Stack Is Not The Same As The Type Of The Value In The Accumulator
The operands in the instruction are incopatible.

Example:

```
  ld   bm23      //illegal
  ld   wm23
  and  w
```

Unable To Open Include File xx Or No Such File
The file specified by an „INCLUDE“ instruction cannot be opened or does not exist.

Unclosed Comment
B-CON provides two comment closing symbols:

1) //Comment
2) {Comment}

When the compiler finds the symbols //, the rest of the line is treated as comment; with the pair of symbols {}, comment can be inserted in any place in the program text.

Example:

```
  ld bm22               //comment
  ld{comment}bm22       //no error message
  ld{comment   bm22     //error message
```

Undefined Argument xx
The operand declared was not defined.

Unexpected Argument xx
In the stated line there is an instruction with too many operands.

Unexpected End Of File
The instruction „EP“ is missing or on reaching the „EP“ identifier there are still data in the stack or ACC.

Unexpected End Of Macro
The macro end identifier „EOMAC“ was found without a macro invitation having taken place.

Unexpected) Found
A closing bracket was found in an incorrect position.

Unrecognized BIT Number In ???
It was attempted to carry out an operation with a bit operand that is out of the legal range.

Example:

```
  ld   i0.0
  st   o0.9        //This instruction cannot be carried out
```

Unrecognized Input IN I???
It was attempted to carry out an operation with a bit operand that is out of the legal range.

Example:

```
  ld   i10.3       //This instruction cannot be carried out.
   st   o0.1
```

Unrecognized Memory IN M???
It was attempted to carry out an operation with a bit operand that is out of the legal range

Example:

```
   ld   m3990.7    //This instruction cannot be carried out.
   st   o0.1
```

Unrecognized Output In O???

Unreachable Code
„JMP“ was used without the declaration of a jump label.

Unknown Argumen
An undefined variable was used

Variable xx Is Greater Than The Maximal Value
The value of the variable declared exceeds the legal maximum value.

Variable xx Is Less Than The Minimal Value
The value of the variable declared remains below the legal minimum value.

ERROR - Opening Debug File xx
The file declared could not be found or opened.

ERROR - Compiler Table Limit
The memeory of the target module provides no more space for variables.

ERROR - Full Table Of Definitions
Per program a maximum of 1000 variable definitions are legal.

ERROR - Opening File xx.Dat
The file with information about the target device could not be opened. This file is created by B-CON when a faultless compiler run has taken place.

ERROR - More Open Cycles Than Closed
At the end of the program there are still data in the stack or ACC.

ERROR - More Load Operations Than Store
At the end of the program there are still data in the stack or ACC.

3.1.4 Runtime Error / Warning flags

BCON utilizes BM0 to give additional information to the user regarding overflow etc.

M0.0 **Communication error (B-CON S)**
M0.0 is set if there is communication with a master.

During the execution of arithmetic operators if borrow, carry, or overflow occurs, bits M7, M6, M5, M4 of external memory BM0 are set, and user can check them.

M0.4 **Division by zero / overflow**
If dividing by zero memory bit M0.4 is set and the maximal value with a sign is returned as a result.

M0.5 **Multiplication overflow**
If an overflow occurs memory bit M0.5 is set and the maximal value with a sign is returned as a result.

M0.6 **Substruction overflow /borrow**
If an overflow /borrow occurs memory bit M0.6 is set and the maximal value with a sign is returned as a result.

M0.7 **Addition overflow / carry**
If an overflow / carry occurs memory M0.7 is set and the maximal value with a sign is returned as a result.

BM1 is used to change value for controlling the System LED.

BM1	System indicator
0	ON (default)
1	OFF
2	Flash (symmetrical)
3	Flash (symmetrical)
4	Flash (asymmetrical)
>4	Not defined

Status/run-time errors
The general System 2000 STATUS word is available to the user by the LD STATUS instruction.

Error checking is implemented in the execution time also.
If the application program runs correctly the System LED ON.
If it flashes then the application program doesn't run. It means it has not been started or an error is detected and the program stops.

If an error is detected during program start or execution a bit H_STATUS.6 is set.
The byte L_STATUS indicates the type of the error:

L_STATUS = 50H, means time-out elapsed without an event occured. This could appear if there is a serious error in t he system.
L_STATUS = 60H, means scan time defined by a user is shorter than the execution time of a Scan_task.
L_STATUS = 70H, means the application program is longer than scan time. If another error is detected in the system operation not
connected with application task execution, the application is not stopped and error can be determined by reading
of STATUS word, as defined in the System 2000 users manual.
The application program is then responsible to handle these excepttions.

3.1.5 Examples of B-CON programs

B-CON S program utilising symbolic names.

```
/INTRODUCTION
/This a program example for BCON-S.
/The program is written for an island having 16DI, 16DO and 8AI.
/Use F5 to obtain full screen!

/IMPORTANT NOTE!!
/Definitions and instructions must be placed separated from
/the left margin (e.g. with 1 or 2 tabs).
/Any information written far to the left are consider as a label!

/DEFINITION OF REGISTERS FOR THE COMMUNICATION TO MASTER
/A program should always start with definition of registers to be used in
/the communication between master and slave  and I/Os used
/Registers used for communication with the BITBUS master has to be defined.
/5 register banks are available DI/AI/ZI for direct transfer from input to
/master and YI/YO for passing values between master and B-CON program
/If outputs are to be controlled from the master YO must used and values
/must be copied in the module from e.g. wi600 to wo0 using ld-st.

/example:

#define DI_CNT 1   /The number equals number of words
#define AI_CNT 8
#define ZI_CNT 0   /This line is not needed as default is 0
#define YI_CNT 2
#define YO_CNT 2
```

```
/DEFINITION OF SYMBOLIC NAMES
/It could be very convenient to use symbolic names for all physical
/IO signals as it makes changes easier and it makes it easier to
/overview the program.

/example of definition (values used for examples below):

#define    up i0.0           /clock for upwards
#define    down i0.1         /clock for downwards
#define    cntval bo0        /counter value

#define    analogin wi2006   /analog input 3
#define    setpoint wi6000   /setpoint from master (YO 0)
#define    error wo6000      /error to master (YI 0)
#define    alarm o1.0        /digital output 0

#define    timerin i0.2      /start of timer (digital input 2)
#define    delayout o1.1    /output from timer (digital out 1)

#define    this i0.3         /digital input 3
#define    that i0.4         /digital input 4
#define    andout o1.2       /output from and (digital out 9)
#define    orout o1.3        /output from or (digital out 10)

/also internal variables could be defined with symbolic names

/example

word delay
byte john
bit test
```

```
/START OF PROGRAM
/The program must start with the label "main"

main:

/LOAD-STORE
    ld      i0.0
    st      test

    ld      test
    st      o1.4

/AND
    ld      this
    and     that
    st      andout

/OR
    ld      this
    or      that
    st      orout

/UP DOWN COUNTER
     cntud up,down,c0,c0,wm2,wc0,m4.0,m4.1
    ld      bm2
    st      cntval

    ld      wm2
    st      wo6002    /transfer counter value to master

/OUTPUT FROM MASTER
    ld      i6002.0   /3 lowest bits to output 13-15
    st      o1.5
    ld      i6002.1
    st      o1.6
    ld      i6002.2
    st      o1.7

/ANALOG COMPARATOR
    ld      analogin
    gt      setpoint  /derived from the master via YI
    st      alarm

/DIFFERENCE BETWEEN INPUT AND SETPOINT
    ld      analogin
    sub     setpoint
    st      error     /transferred to the master

/TIMER (ON DELAY)
     tmr1   bc1,timerin,delayout,wc100  /wc100=100x10ms


/END OF PROGRAM
    /The program must always be terminated with "ep"

ep
```

B-CON M program Example using direct I/O addressing

/This is a demo program for master and message display
/The master (M) is UCB-16DIO + UCL-08AI
/The display (DIS) is UCT-42 (variable digits 0,1,2,3 used)

```
main:
    ld    m.wi0             /read digital input
    st    dis.wo4000        /select text no.

    ld    m.wi2006          /read analog input
    st    dis.wo4002        /display analog

    ld    dis.wi4000        /read keypad
    st    m.wo0             /write digital out

    ep
```

B-CON M program example using symbolic names (define)

```
  /This is a demo program for master and message display
  /The master (M) is UCB-16DIO + UCL-08AI
  /The display (DIS) is UCT-42 (variable digits 0,1,2,3 used)

    #define digin m.wi0
    #define text dis.wo4000
    #define analog m.wi2006
    #define var1 dis.wo4002
    #define key dis.wi4000
    #define digout m.wo0

  main:
    ld    digin       /read digital input
    st    text        /select text no.

    ld    analog      /read analog input
    st    var1        /display analog

    ld    key         /read keypad
    st    digout      /write digital out

    ep
```

3.1.6 List of sample programs

The CD-Rom provides the sample programs listed below in the sub-directory EXAMPLES and in further subregisters as source program file *.PGM.

The programs refer to the application and handling of a command, a function or a smaller task. The name of the program file is self-explanatory and as a rule points to the application of the command, the function or the problem.

Sample programs in subdirectory "EXAMPLES"

In subdirectory „EXAMPLES“ the sample programs are listed in alphabetical order. The application examples refer to the stock of basic commands such as logical conjunction, fixed-point arithmetic, compare and jump commands.

Program Name

demoror	pgm	p_orpp	pgm
grenzen	pgm	p_r	pgm
p_add	pgm	p_resdom	pgm
p_addpp	pgm	p_rol	pgm
p_and	pgm	p_ror	pgm
p_andn	pgm	p_s	pgm
p_andnpp	pgm	p_setdom	pgm
p_andpp	pgm	p_st	pgm
p_div	pgm	p_stn	pgm
p_divpp	pgm	p_sub	pgm
p_divpp1	pgm	p_subpp	pgm
p_dup	pgm	p_ueq	pgm
p_dupn	pgm	p_uegpp	pgm
p_dupnpp	pgm	p_uge	pgm
p_duppp	pgm	p_ugepp	pgm
p_eq	pgm	p_ugt	pgm
p_eqpp	pgm	p_ugtpp	pgm
p_ge	pgm	p_ule	pgm
p_gepp	pgm	p_ulepp	pgm
p_gt	pgm	p_ult	pgm
p_gtpp	pgm	p_ultpp	pgm
p_jmp	pgm	p_xor	pgm
p_jmpc	pgm	p_xorn	pgm
p_jmpcn	pgm	p_xornpp	pgm
p_ld	pgm	p_xorpp	pgm
p_ldn	pgm	p_cntdn	pgm
p_le	pgm	p_cntud	pgm
p_lepp	pgm	p_cntup	pgm
p_lt	pgm	p_dnpls	pgm
p_ltpp	pgm	p_dtr	pgm
p_mul	pgm	p_rtr	pgm
p_mulpp	pgm	p_str	pgm
p_or	pgm	p_upls	pgm
p_orn	pgm		
p_ornpp	pgm		

3.2 Basic Logical Instructions

3.2.1 Load Instructions

LD Load value.
This load instruction puts the operand, which is stated in the command, into the ACC.
The value that was stored in the ACC before, is automatically pushed down into the stack. By this process, the depth of the stack is decremented.

The load instruction (operator) works with all data formats and data types.

Declaration examples:

```
LD BI0          /load input byte No 0
LD I1.7         /load input bit 1.7
```

Programming example:

```
/P_LD.PGM
 /Example: data exchange instruction LD
 /LD = load operand into ACC
 /ST = load contents of ACC into operand

 ld   i0.1 /load input bit into ACC
 st   o0.7 /store contents of ACC in output bit
 ep        /end of programme
```

LDN Load value negated (load value inverted).
This load instruction negates the operand stated as parameter in the command and loads it into the ACC.
The value that was stored in the ACC before is automatically pushed down into the stack.

The LDN instruction (operator) works with all data formats and data types provided they are not constants. Constants cannot be negated.

Declaration examples:

```
  LDN BI0     /load input byte No 0 negated
  LDN I1.7    /load input bit 1.7 negated
```

Programming example:

```
 /P_LDN.PGM
 /example: data exchange instruction LDN
 /LDN = load operand negated into ACC
 /ST = load contents of ACC into operand

 ldn  i0.1       /load input bit negated into ACC
 st   o0.7       /store contents of ACC in output bit
 ep              /end of program
```

3.2.2 Store Instructions

ST Store value.
This storage instruction stores the value resident in the ACC into the operand stated as parameter in the instruction.
After execution of the command the previous contents of the ACC are lost and the value stored in the stack last is pushed down into the ACC. By this process the depth of the stack is incremented.

The ST instruction works with all data formats and with data types; outputs and markers, not with inputs.

Declaration examples:

```
 ST     BO0       /store Acc in output byte No 0
 ST     O1.7      /store Acc in output bit 1.7
```

Program example:

```
/P_ST.PGM
/example: data exchange instruction ST
/LD = load operand into ACC
/ST = load contents of ACC into operand

ld   bi0  /load input byte into ACC
st   bo0  /store contents of ACC in output byte
ep        /end of program
```

STN Store value negated.
This storage instruction negates the value resident in the stack and stores it in the operand stated in the command as parameter.
After the execution of the command the previous contents of the ACC is lost and the value stored in the stack last is pushed down into the ACC. By this process the depth of the stack is incremented.

The STN instruction works with all data formats and with data types, outputs, and markers - not with inputs.

Declaration examples:

```
STN  BO0  /store ACC negated in output byte No 0
STN  O1.7 /store ACC negated in output bit 1.7
```

Programming example:

```
/P_STN.PGM
/example: data exchange instruction STN
/LD = load operand into ACC
/STN = load contents of ACC negated into operand

ld   bi0  /load input byte into ACC
stn  bo0  /store contents of ACC negated in output
     byte
ep        /end of program
```

3.2.3 Set and Reset Instructions

With this kind of commands, operands in bit format can be set or reset storing.

S Set instruction.
The S instruction sets the operand, stated in the operand, as parameter to the Boolean value 1 (TRUE).

The operand is set storing to 1 when the contents of the ACC have the value 1 through a previous operation. In case the contents of the ACC have the value 0 at this time, the operand declared is not changed.
After the execution of the command, the contents of the ACC are lost and a new value is automatically pushed down into the ACC. By this process the stack address is incremented.

The S instruction can only be used in bit format. As operand types only marker and output variables can be applied. Byte and word formats are not permitted!

Declaration examples:

```
 S      M10.1     et marker bit 10.1 to log.1
        o0.5      t output bit 0,5 to log.1
```

Programming example:

```
P_S.PGM
example: S = set storing

ld    c1      /load bit constant value 1 into ACC
s     o0.1    /set output variable to log.1
ep            /end of program
```

R Reset instruction.
The R instruction sets the operand declared in the command as parameter to the Boolean value 0 (FALSE).

The operand is set storing to 0, when the contents of the ACC have the value 1, through a previous operation. In case the contents of the ACC have the value 0 at this time, the operand declared is not changed.
After the execution of the command the contents of the ACC are lost and a new value is automatically pushed down from the stack into the ACC. By this process the stack address is incremented.

The R instruction can only be used in bit format. AS operand types only marker and output variables can be applied. Byte and word formats are not permitted!

Declaration examples:

```
M10.1     /reset marker bit 10.1 to log.0
O0.5      /reset output bit 0.5 to log.0
```

Programming example:

```
/P_R.PGM
/example: R = reset storing

ld    c1        /load bit constant value 1 into ACC
r     o0.1      /reset output variable to log.0
ep              /end of program
```

3.2.4. Dominant Setting and Resetting

An application of the combination of the R and S operator frequently used in control technique is the RS flip-flop.

This principle consists in both operators accessing a common target variable.
The sequence of commands decides whether the setting (S) or resetting (RS) is being dominant.
In the co-operation of these two operators the one that is the last in the sequence of the instruction is always dominant.

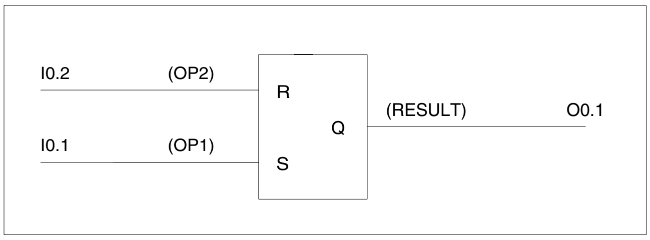
In this instruction of the target, variables are only possible in bit format and with marker and output variables.

RS Flip-Flop, dominant setting.
In the table of functions the correlation's between operators S an R in their relation to the common output variables are illustrated. Dominant setting is the state of SET and RESET input having log.1 signal and the output being set to log.1.

Table of functions:

S (OP1)	**R** (OP2)	**Output** (RESULT)
0	0	Previous history
0	1	0
1	0	1
1	1	1 (dominant position)

Function chart:



Programming example:

```
/P_SETDOM.PGM
/example: S dominant = set target variable dominant

ld    i0.2      /OP2: load input variable into ACC
r     o0.1      /RESULT: reset target variable
ld    i0.1      /OP1: load input variable
s     o0.1      /RESULT: set target variable
ep              /end of program
```

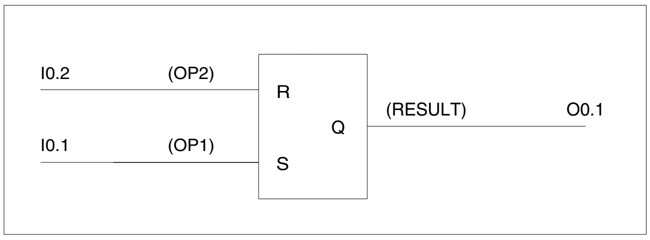
RS Flip-Flop, dominant resetting.

In the table of functions the correlation's between operators S and R in their relation to the common output variable are illustrated. Dominant resetting is the state of SET and RESET input having the log. 1 signal and the output being reset to log.0.

Table of functions:

S (OP1)	**R** (OP2)	**Output** (RESULT)
0	0	Previous history
0	1	0
1	0	1
1	1	0 (dominant position)

Function chart:



Programming example:

```
/P_RESDOM.PGM
/example: R dominant = reset target variable dominant

ld   i0.1 /OP1: load input variable
s    o0.1 RESULT: set target variable
rd   i0.2 load input variable into ACC
r    o0.1 /RESULT: reset target variable
ep        /end of program
```

3.2.5 Logical AND Instructions

Logical conjunctions serve the purpose of realising combinatorial circuits or networks. They follow the rules of Boolean algebra and are an important means in digital technique.
The AND conjunction creates a result of two binary variables, which can be illustrated by a series connection of contacts.

AND Logical AND instruction.
The AND command (operator) actuates a bitwise logical AND conjunction between operand 1 (OP1) and operand 2 (OP2).

OP1 must previously be loaded into the ACC and OP2 is activated by the parameter following in the instruction. The result of the conjunction (RESULT) is available in the ACC. Thereby the former contents of the ACC are overwritten. The contents and depth of the stack are not changed.

When an AND instruction is to be carried out without the declaration of operands, it is understood that OP1 is in the stack and OP2 was loaded into the ACC. Again the RESULT is available in the ACC the former contents of the ACC being overwritten. In this case the stack is incremented.

The AND instruction can be carried out with all data formats and data types.

Declaration examples:

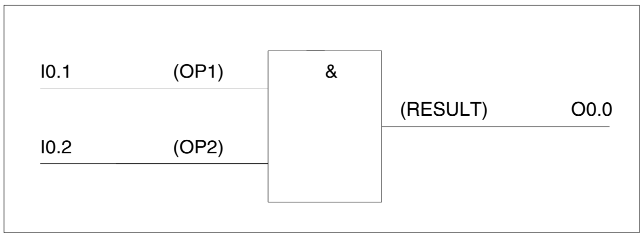
```
AND   I0.1  /AND conjunction ACC with input byte 0.1
AND   BCHFA /AND conjunction ACC with byte  constant FAh
```

In the following table of functions the relations in the bit version between OP1 and OP2 to the result variable (RESULT) are illustrated.

Table of AND functions:

(OP1)	(OP2)	RESULT
0	0	0
0	1	0
1	0	0
1	1	1

Function chart:



Programming example:

```
/P_AND.PGM
 /example: AND conjunction in bit format
ld   i0.1 /OP1: load input
and  i0.2 /OP2: AND conjunction OP1 with OP2
s    o0.0 /RESULT: store in output
ep        /end of program
```

ANDN Logical AND with negation.

The ANDN instruction actuates a bitwise logical AND conjunction between operand 1 (OP1) and operand 2 (OP2), which was negated.

OP1 must previously be loaded into the ACC and OP2 is activated by the parameter following in the instruction. The conjunction result is available in the ACC as RESULT. By this process the former contents of the ACC are overwritten. Contents and depth of the stack are unchanged in this case.

When an AND instruction has to be executed without the declaration of operands, it is understood that OP1 is in the stack and OP2 was loaded into the ACC. Again the result is available in the ACC as RESULT, the former contents of the ACC having been overwritten. In this case the stack address is incremented.

The ANDN instruction can be carried out with all data formats and data types. However, OP2 cannot be a constant.

Note: Constants cannot be inverted.

Declaration examples:

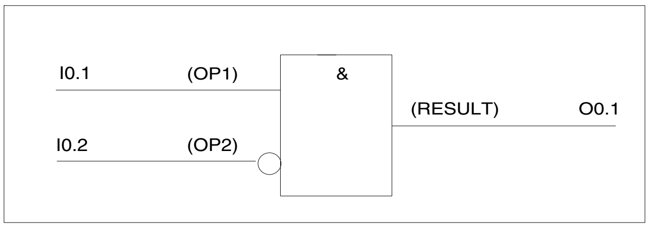
```
ANDN I0.1 /AND conjunction ACC with negated input bit
          0.1
ANDN B00  /AND  conjunction  ACC  with  negated  output
          byte 0
```

In the following table of functions the relations in the bit version of OP1 and inverted OP2 to result variable RESULT are illustrated.

Table of ANDN functions:

(OP1)	(OP2)	OP2 inverse	RESULT
0	0	1	0
0	1	0	0
1	0	1	1
1	1	0	0

Function chart:



Programming example:

```
/P_ANDN.PGM
/example: ANDN = AND conjunction with previous
/negation of OP2

ld     i0.1    /OP1: load input bit
andn   i0.2    /OP2: negation of operand 2 and AND
conjunction with OP1
st     o0.1    /RESULT: store in output bit
ep             /end of program
```

AND(...) Logical AND with nesting.

The AND instruction with command nesting in brackets actuates a bitwise logical AND conjunction between operand 1 (OP1) and operand 2 (OP2).
OP1 must previously be loaded into the ACC and OP2 is activated by the parameter following in the instruction. The conjunction result is available in the ACC as RESULT. By this process, the former ACC contents are overwritten.
The contents and depth of the stack are unchanged after the execution of the instruction. When, however, an operand in the bracket is a value of the stack, the stack depth is incremented.

Declaration example:

```
AND( i0.2  /AND conjunction of ACC with the temporary
OR i0.3    /result of the bracket
)
```

The sequence in the execution of the command is this: First the operations in the bracket are executed. They create OP2.

Example, execution of the AND(...) instruction:

Program listing	Sequence of execution	Coments
LD I0.0 AND(I0.1 OR I0.2)AND I0.O	LD I0.0 LD I0.1 OR I0.2 /OP1	/OP1 /(result represent /(OP2

The AND(...) instruction can be carried out with all data formats and data types.

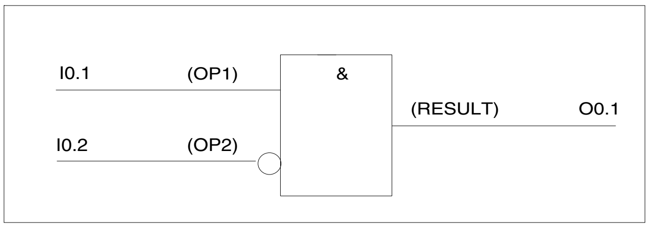
The compiler permits up to sixfold nesting of such combinations of commands provided the stack is not overloaded. Maximum stack depth is 8 bytes.

In the following table of functions the relations in the bit version of OP1 and OP2 to the result variable (RESULT) is illustrated.

Table of AND(...) functions:

OP1	**OP2 (...)**	**RESULT**
0	0	0
0	1	0
1	0	0
1	1	1

Function chart:



Programming example:

```
/P_ANDPP.PGM
/example: AND(...) = AND conjunction with succeeding
/commands in brackets
ld      i0.0    /OP1: load input bit
and(    i.0.1   /OP21: OR conjunction with all bracketed
or      i0.3    /OP23: temporary operand OP2
)       /AND conjunction of OP1 and OP2
st      o0.0    /RESULT: store in output bit
ep              /end of program
```

ANDN(...) **Logical AND with negation and nesting.**

The ANDN instruction with command nesting in brackets actuates a bitwise logical AND conjunction between operand 1 (OP1) and operand 2 (OP2), which is the temporary result of the bracket previously negated.
OP1 must previously be loaded into the ACC and OP2 is activated by the parameter following in the instruction. This parameter is created by the command nesting in the bracket. The conjunction result is available in the ACC as RESULT. By this process the former ACC contents are overwritten.
The stack contents and depth are unchanged after the execution of the instruction. When, however, an operand in the bracket is a value of the stack, the stack depth is incremented.

Declaration example:

```
ANDN(i0.2   /AND conjunction ACC with negated
OR    i0.3  /temporary result of the bracket
)
```

The sequence in the execution of the instruction is this: First the operations declared in the bracket are executed. They create OP2.

Example, execution of the ANDN(...) instruction:

Program listing	Sequence of execution	Coments
LD I0.0	LD I0.0	/OP1
ANDN(I0.1	LD I0.1	/(result represent
OR I0.2	OR I0.2	/(OP2
)ANDN I0.O		/OP1

The ANDN(...) instruction can be carried out with all data formats and data types.

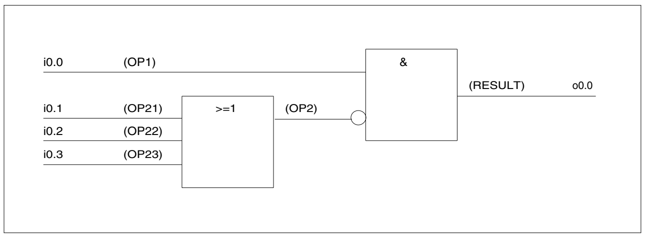
The compiler permits up to sixfold nesting of such combinations of commands. Maximum stack depth is 8 bytes.

In the following table of functions the relations in the bit version of OP1 and OP2 to the result variable (RESULT) are illustrated.

Table of ANDN(...) functions:

(OP1)	(OP2) **(...)**	OP2 **negated**	RESULT
0	0	1	0
0	1	0	0
1	0	1	1
1	1	0	0

Function chart:



Programming example:

```
/P_ANDPP.PGM
/example: ANDN(...) = AND conjunction with succeeding
/commands in brackets and with OP1 negation

ld      i0.0     /OP1: load input bit
andn(   i0.1     /OP21: OR conjunction of all the bracketed
or      i.0.2    /OP22: operands OP21...23 to
or      i0.3     /OP23: temporary operand OP2
)       /OP1 negation and AND conjunction with OP2
st      o0.0     /RESULT: store in ooutput bit
ep               /end of program
```

3.2.6 Logical OR Instructions

The OR conjunction, also called disjunction, creates a result between two binary variables which can be illustrated by the shunting of contacts.

OR Logical OR instruction.

The OR instruction (operator) actuates a bitwise logical OR conjunction between operand 1 (OP1) and operand 2 (OP2).

OP1 must previously be loaded into the ACC and OP2 is activated by the parameter following in the instruction. The conjunction result is available in the ACC as RESULT. By this process the former ACC contents are overwritten. In this case the stack contents and depth are unchanged.
When an OR instruction is to be executed without the declaration of operands, it is understood that OP1 is in the stack and OP2 has been loaded into the ACC. Again the result is available in the ACC as RESULT the former contents of the ACC having been overwritten. In this case the stack is incremented.

The OR instruction can be carried out with all data formats and data types.

Declaration examples:

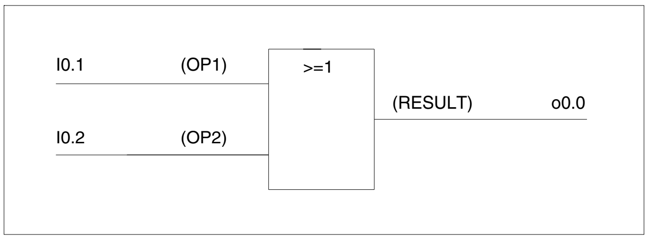
```
OR    I0.1     /OR conjunction ACC with input bit 0.1
OR    BCHFA  /OR conjunction ACC with byte constant FAh
```

In the following table of functions the relations in the bit version of OP1 and OP2 to the result variable (RESULT) are illustrated.

Table of OR functions:

OP1	**OP2 (...)**	**RESULT**
0	0	0
0	1	1
1	0	1
1	1	1

Function chart:



Programming example:

```
/P_OR.PGM
/example: OR conjunction in bit format

ld     i0.1    /OP1: load input
or     i0.2    /OP2: OR conjunction OP1 with OP2
st     o0.0    /RESULT: store in output
ep             /end of program
```

ORN Logical OR with negation.

The ORN instruction (operator) actuates a bitwise logical OR conjunction between operand 1 (OP1) and operand 2 (OP2), which was previously negated.
OP1 must previously be loaded into the ACC and OP2 is activated by the parameter following in the instruction. The conjunction result is available in the ACC as RESULT. By this process the former contents of the ACC are overwritten. In this case the stack contents and depth are unchanged.

When an ORN instruction is to be executed without the declaration of operands, it is understood that OP1 is in the stack and OP2 was loaded into the ACC. Again the result is available in the ACC as RESULT the former contents of the ACC having been overwritten. In this case the stack address is incremented.

The ORN instruction can be carried out with all data formats and data types. However, OP2 cannot be a constant.

Note: Constants cannot be inverted!

Declaration examples:

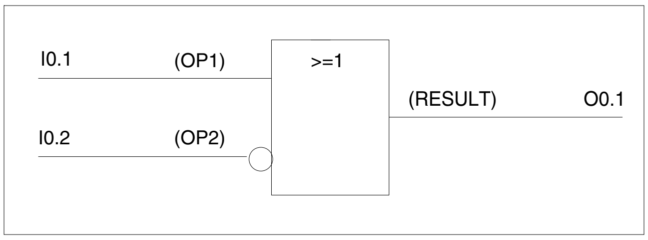
```
ORN  I0.1 /OR conjunction ACC with negated input bit 0.1
ORN  B00  /OR conjunction ACC with negated output byte 0
```

In the following table of functions the relations in the bit version of OP1 and negated OP2 to the result variable (RESULT) are illustrated.

Table of ORN functions:

(OP1)	(OP2)	OP2 **inverse**	RESULT
0	0	1	1
0	1	0	0
1	0	1	1
1	1	0	1

Function chart:



Programming example:

```
  /P_ORN.PGM
  /example: ORN conjunction with previous negation of OP2

  ld     i0.1     /OP1: load input bit
  orn    i0.2     /OP2:  negation  and  OR  conjunction  with
OP1
  st     o0.0     /RESULT: store in output bit
  ep              /end of program
```

The ORN instruction with command nesting in brackets actuates a bitwise logical OR conjunction between operand 1 (OP1) and the previously negated temporary result of the bracket as operand 2 (OP2).
OP1 must be previously loaded into the ACC and OP2 is activated by the parameter following in the instruction. The conjunction result is available in the ACC as RESULT. By this process the former ACC contents are overwritten.

The stack contents and depth are unchanged after the execution of the command. When, however, an operand in the bracket is a value of the stack, the stack depth is incremented.

Declaration example:

```
ORN(     i0.2     /OR conjuction ACC with temporary
AND      i0.3     /result of the bracket
)
```

The sequence in the execution of this instruction is this: First the operations declared in the bracket are executed. They create OP2.

Example, execution of the ORN(...) instruction:

Program listing	Sequence of execution	Coments
LD I0.0 ORN(I0.1 AND I0.2)	LD I0.0 LD I0.1 AND I0.2 ORN I0.0	/OP1 /(result represent /(OP2 /OP1

The ORN(...) instruction can be carried out with all data formats and data types.

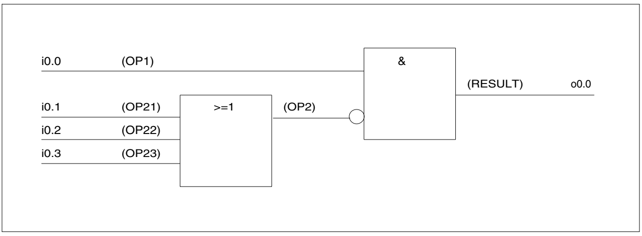
The compiler permits up to sixfold nesting of such combinations of commands provided the stack is not overloaded. Maximum stack depth is 8 bytes.

In the following table of functions the relations in the bit version of OP1 and OP2 to the result variable (RESULT) are illustrated.

Table of ORN(...) functions:

OP1	OP2 (...)	OP2 negated	RESULT
0	1	0	0
0	0	1	1
1	1	0	1
1	0	1	1

Function chart:



3.2.7 Logical XOR Instructions

The XOR conjunction also called „exclusive OR“ is a combinatorial switching circuit between AND, OR, and NOR members of two binary variables. In the classic case it represents non-equivalence. Through the negation of one of the two inputs equivalence is created.

XOR Logical XOR (non-equivalence).

The XOR instruction (operator) actuates a logical XOR conjunction between operand 1 (OP1) and operand 2 (OP2).
OP1 must previously be loaded into the ACC and OP2 is activated by the parameter following in the instruction. The conjunction result is available in the ACC as RESULT. By this process the former ACC contents are overwritten. In this case the stack contents and depth are unchanged.

When an XOR instruction without operands is to be executed, it is understood that OP1 is in the stack and OP2 was loaded into the ACC. Again the result is available in the ACC as RESULT the former ACC contents having been overwritten. In this case the stack address is incremented.

The XOR instruction can be carried out with all data formats and data types.

Declaration examples:

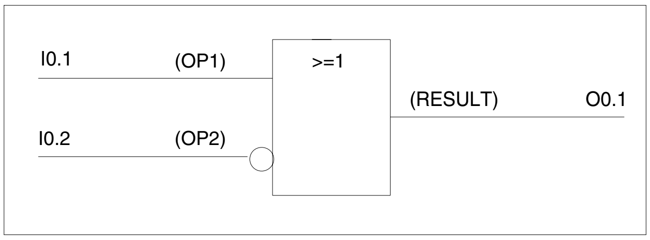
```
XOR  I0.1   /XOR conjunction ACC with input bit 0.1
XOR  BCHFA  /XOR conjunction ACC with byte constant FAh
```

In the following table of functions the relations in the bit version of OP1 and OP2 to the result variable (RESULT) is illustrated:

Table of XOR functions (non-equivalence):

OP1	OP2	RESULT
0	0	0
0	1	1
1	0	1
1	1	0

Function chart (non-equivalence):



Programming example:

```
/P_XOR.PGM
/example: XOR conjunction (non-equivalence)

ld      i0.1    /OP1:load input bit
xor     i0.2    /OP2: XOR conjunction with OP1
st      o0.0    /RESULT: store in output bit
ep              /end of program
```

Programming example:

```
/P_ORNPP.PGM
/example: ORN(..) = OR conjunction with succeeding
/commands in brackets and negation of OP2

ld      i0.0    /OP1: load input bit
orn(    i0.1    /OP21: AND conjunction of all bracketed
and     i0.2    /OP22: operands OP21...OP23 to
and     i0.3    /OP23: temporary operand OP2
)               /OP2 negation and OR conjunction with OP1
st      o0.0    /RESULT: store in output bit
ep              /end of program
```

XORN **Logical XOR with negation** (equivalence).

The XORN instruction (operator) actuates a bitwise logical XOR conjunction between operand 1 (OP1) and previously negated operand 2 (OP2).
OP1 must be previously loaded into the ACC and OP2 is activated by the parameter following in the instruction. The conjunction result is available in the ACC as RESULT. By this process the former contents of the ACC are overwritten. In this case the stack contents and depth are unchanged.
When an XORN instruction is to be executed without the declaration of an operand, it is understood that OP1 is in the stack and OP2 was loaded into the ACC. Again the result is available in the ACC as RESULT the former contents of the ACC having been overwritten. In this case the stack address is incremented.

The XORN instruction can be carried out with all data formats and data types. However, OP2 cannot be a constant.

Note: Constants cannot be inverted!

Declaration example:

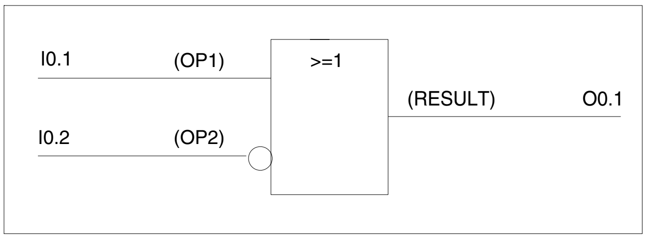
```
XORN 10.1 /XORN conjunction ACC with input bit 0.1
XORN BCHFA  /XORN conjunction ACC with byte constant FAh
```

In the following table of functions the relations in the bit version of OP1 and OP2 to the result variable (RESULT) are illustrated.

Table of XORN functions (equivalence):

OP1	**OP2**	**OP2 inverse**	**RESULT**
0	0	1	1
0	1	0	0
1	0	1	0
1	1	0	1

Function chart (equivalence):



Programming example:

```
/P_XORN.PGM
/example: XORN conjunction (equivalence)

ld   i0.1 /OP1: load input bit
xorn i0.2 /OP2: XORN conjunction with OP1
st   o0.0 /RESULT: store in output bit
ep        /end of program
```

XOR(...) Logical XOR with nesting.

The XOR instruction with command nesting in brackets actuates a bitwise logical XOR conjunction between operand 1 (OP1) and operand 2 (OP2).
OP1 is the current ACC contents. OP2 is created of the temporary result of the instructions in brackets following the command as parameters. The conjunction result is available in the ACC as RESULT. By this process the former ACC contents are overwritten. The stack contents and depth are unchanged. When, however, an operand in the bracket is a value of the stack, the stack depth is incremented.

Declaration example:

```
XOR( i0.2 /XOR conjunction ACC with the temporary
AND  i0.3 /result of the bracket
)
```

The sequence of the execution of the instruction is this: First the operations declared in the bracket are executed. They create OP2.

Example, execution of the XOR(...) instruction:

Program listing	Sequence of execution	Coments
LD I0.0 XOR(I0.1 AND I0.2)	LD I0.0 LD I0.1 AND I0.2 OR I0.0	/OP1 /(result represent /(OP2 /OP1

The XOR(..) instruction can be carried out with all data formats and data types.

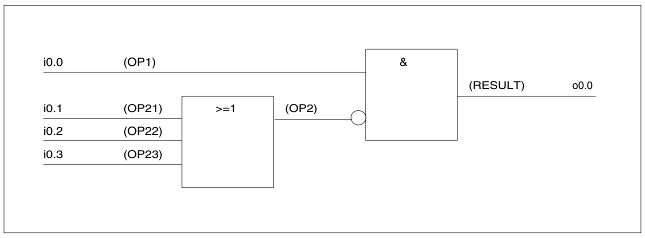
The compiler permits up to sixfold nesting of such command combinations provided the stack is not overloaded. Maximum stack depth is 8 bytes.

In the following table of functions the relations in the bit version of OP1 and OP2 to the result variable (RESULT) are illustrated.

Table of XOR(...) functions:

OP1	OP2 (...)	RESULT
0	0	0
0	1	1
1	0	1
1	1	0

Function chart:



Programming example:

```
/P_XORPP.PGM
/example: XOR(...) = XOR conjunction with succeeding
/commands in brackets
/non-equivalence function (extended)

ld      i=.0     /OP1: load input bit
xor(    i0.1     /OP21: AND conjunction of all bracketed
and     i0.2     /OP22: operands OP21...OP23 to the
and     i0.3     /OP23: temporary operand OP2
)                /XOR conjunction OP1 with OP2
st      o0.0     /RESULT: store in output bit
ep               /end of program
```

XORN(...) **Logical XOR with negation and nesting.**

The XORN instruction with command nesting in brackets actuates a bitwise logical XOR conjunction between operand 1 (OP1) and the previously negated temporary result of the bracket as operand 2 (OP2).
OP1 must previously be loaded into the ACC and OP2 is activated by the parameter following in the instruction. It is created by the command nesting in the bracket. The conjunction result is available in the ACC as RESULT. By this process the former ACC contents are overwritten.
The stack contents and depth are unchanged after the execution of the command. When, however, an operand in the bracket is a value of the stack, the stack depth is incremented.

The sequence in the execution of the instruction is this: First the operations declared in the bracket are executed. They create OP2.

Example, execution of the XORN(...) instruction:

Program listing	Sequence of execution	Coments
LD I0.0	LD I0.0	/OP1
XORN(I0.1	LD I0.1	/(result represent
AND I0.2	AND I0.2	/(OP2
)	XORN I0.0	/OP1

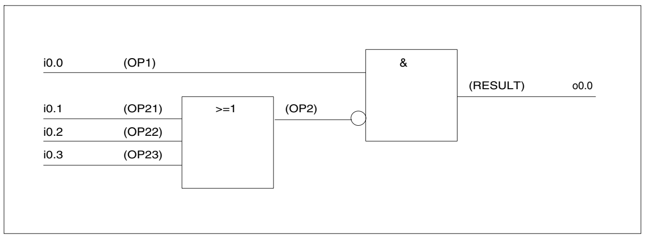
The XORN(...) instruction can be carried out with all data formats and data types.
The compiler permits up to sixfold nesting of such command combinations provided the stack is not overloaded. Maximum stack depth is 8 bytes.

In the following table of functions the relations in the bit version of OP1 and OP2 to the result variable (RESULT) are illustrated.

Table of XORN(...) functions:

OP1	OP2 (...)	OP2 negated	RESULT
0	1	0	0
0	0	1	1
1	1	0	1
1	0	1	0

Function chart:



Programming example:

```
/P_XORNPP.PGM
/example: XORN(...) = XOR conjunction with succeeding
/commands in brackets and negation of OP1,
/equivalence function (extended)

 ld      i0.0     /OP1: load input bit
 xorn(   i0.1     /OP21: AND conjunction of all bracketed
 and     i0.2     /OP22: operands OP21...OP23 to
 and     i0.3     /OP23: temporary operand OP2
 )                /OP2 negated and XOR conjunction OP1 with OP2
 st      o0.0     /RESULT: store in output bit
 ep               /end of program
```

3.3 SHIFT Instructions

ROR Rotate ACC contents to the right.

The ROR instruction (rotate right) actuates the rotation of the value or bit pattern respectively in the ACC to the right.

Only byte and word operands are permitted.

Declaration example (byte format):

```
LD   bch80  /load bit pattern 80h into ACC
ROR  bch01  /rotate ACC contents to the right by 1 bit
```

The operation actuates the rotation of the current ACC contents to the right by the parameter value declared in the instruction. Thus the parameter decides on the number of bit digits to be moved.

The command „rotate right“ means that the ACC contents (bit pattern) are moved from higher-ranking to lower-ranking bit digits by the number n. According to arithmetic logic the rotation to the right has the effect of a divide operation by the following rule:

<ACC> by n digits = division by 2^n .

Programming example:

```
/P_ROR:PGM
/example: ROR = rotate right (ACC contents)
/the loaded bit combination is rotated from the higher to the
/ lower position by the number of bit digits declared in the
/ instruction.

ld      m1.7      /switch for rotation
jmpc    zyklus
ld      c1
st      m1.7      /switch for start condition
ld      bch80     /bit pattern for start condition
st      bm10
zyklus:
ld      bm10
ror     bc1       /rotate right by 1 (n) bit digits
st      bm10
ld      bm10
st      bo0       /readout bit pattern
ep
```

ROL Rotate contents of ACC to the left.

The ROL instruction (rotate left) actuates the rotation of the value or bit pattern respectively in the ACC to the left.

Only byte and word operations are permitted.

Declaration example (byte format):

```
LD      bch01     /load bit pattern 01h into the ACC
ROL     bch01     /rotate ACC contents to the left by one bit digit
```

The execution of the instruction has the effect that the current ACC contents is rotated (moved) to the left by the parameter value following the command. Thus the parameter decides on the number of bit digits to be rotated.
The instruction „rotate left“ means that the ACC contents (bit pattern) are moved from lower to higher bit digits by the number n. According to arithmetic logic the rotation to the left has the effect of multiply operation by the following rule:

<ACC> by n digits = multiplication by 2^n .

Programming example:

```
/P_ROL.PGM
/example: ROL = rotate left (ACC contents)
/The loaded bit combination is rotated from  a lower
/to a higher position by the number of bit digits
/declared in the instruction

ld   m1.7 /switch for rotation
jmpc zyklus
ld   c1
st   m1.7 /switch for start condition
ld   bch0.1 /bit pattern for start condition
st   bm10
zyklus:
ld   bm10
rol  bc1 /rotate left by 1 (n) bit digit
st   bm10
ld   bm10
st   bo0 /readout bit pattern
 ep
```

SHL Shift Left

The operator shifts left the operand, zero is inserted in the empty spaces.

It works only with BYTE, and WORD operand types.

The first operand is accumulator contents (the value in the accumulator is shift), the second one is given in the instruction (the number of shifts to be done), and the result is left in the accumulator.
The stack contents and depth is not changed in the common case.

Example:

```
                // BM33=2h, Acc = 91h
SHL   bm33      // Acc = 44h (68dec)
```

Example:

```
                // BM33=8h, Acc = 12h
SHL   wm33      // Acc = 1200h
```

SHR Shift Right

The operator shifts right the operand, zero is inserted in the empty spaces.

It works only with BYTE, and WORD operand types.

The first operand is accumulator contents (the value in the accumulator is shift), the second one is given in the instruction (the number of shifts to be done), and the result is left in the accumulator.
The stack contents and depth is not changed in the common case.

Example:

```
                // BM33=2h, Acc = 91h
SHR   BMm33     // Acc = 24h (36dec)
```

Example:

```
                // BM33=8h, Acc = 1255h
SHR   wm33      //Acc=12h
```

3.4 Arithmetic Instructions

Arithmetic instructions enable the execution of mathematic operations of two fixed-point numbers.

Fixed-point numbers are signed integer numbers. In the binary system plus and minus signs of numbers can only be represented by 0 or 1. For this reason the amount of numbers available in a digital computer had to be divided into two halves.

Numbers belonging to the upper half of the total amount of numbers with 0 as MSB are allocated to positive numbers.
Numbers belonging to the lower half of the total amount of numbers with 1as MSB are allocated to negative numbers.

For the provided facility of the different application of variable formats the following considerations always have to be regarded:

Format	Negative Number	Positive Number	Unsigned Number
8-bit	-128............-1 80h...FFh	0 ....+127 0h..........7Fh	0.....255 0h....FFh
16-bit	-32768.........-1 8000h....FFFFh	0...........+32767 0h.......7FFFh	0......65535 0h...FFFFh

3.4.1 ADD Instructions

ADD

The ADD instruction adds two signed fixed-point numbers. These values are represented in the instruction by operands.

Operand permitted: Byte and word

The first operand is the ACC contents and the second operand is the parameter following the command. The result is available in the ACC as RESULT. By this process the former ACC contents are overwritten. The stack contents and depth are unchanged.

When an ADD instruction is to be executed without the declaration of operands, it is understood that OP1 was loaded into the ACC and OP2 is in the stack. Again the result is available in the ACC as RESULT the former ACC contents having been overwritten. In this case the stack address is incremented.

According to the selection of the operand format (byte or word), an overflow bit in the formation of the result, M0.7 = 1, is created by the operating system if the range of values is exceeded. This overflow bit can be queried after the execution of the operation:

Byte format:	value >+127	M0.7=1
Word format:	value >+32767	M0.7=1

When an overflow bit occurs, the result must be corrected by appropriate programming steps.

Declaration example in byte format:

```
LD   BC100  /OP1:     +100
ADD  BC20   /OP2:     +  20
ST   BO     /RESULT:  +120
```

Programming example:
Creation of bit marker M0.7 and indication of result correction

```
/P_ADD.PGM

/example: ADD=adding with byte operands
/overflow is filed by the system in bit marker M0.7
/if result >+127 in byte processing
/(or when result >32767 in word processing)
/task: 126+3=129
/output, however, shows 81hex=-127, false result
/because range was exceeded -> overflow!

ld   bc126  /OP: load constant „+126“ into ACC
add  bc3 /OP2: add constant „+3“
st   bm10 /RESULT: store in marker byte
ld   m0.7 /load status byte
jmpc overf  /jump if status bit „1“
ld   bm10 /load result
st   bo0 /readout result
jmp  end /unconditional jump to the end
overf: nop  /e.g. correction routine
ld   bm10 /load corrected result
st   bo0 /readout corrected result
 end:   ep     /end of program
```

ADD(...) ADD with nesting.

The ADD instruction with command nesting in brackets executes an add operation between operand 1 (OP1) and operand 2 (OP2) with fixed-point numbers.

Operand permitted: Byte and word

The sequence of the operation is: First the instructions in the bracket create OP2 as temporary result. Then the add operation with OP1, which was previously loaded into the ACC is carried out. The result is available in the ACC as RESULT. By this process the former ACC contents are overwritten.

The stack contents and depth are unchanged after the execution of the instruction. When, however, an operand in the bracket is a value of the stack, the stack address is incremented.

According to the selection of the operand format (byte or word), an overflow bit (M0.7 = 1), is created by the operating system in the formation of the result if the range of values is exceeded.This overflow bit can be queried after the execution of the operation:

Byte format:	value >+127	M0.7=1
Word format:	value >+32767	M0.7=1

When an overflow bit occurs, the result must be corrected by appropriate programming steps.

Example, execution of the ADD(...) instruction:

Program listing	Sequence of execution	Coments
LD OP1 ADD(BI0 MUL BI1)	LD OP1 LD BIO MUL BI1 ADD OP1	/(result represents / OP2)

The compiler permits up to sixfold nesting of such combinations of commands provided the stack is not overloaded. Maximum stack depth is 8 bytes.

Programming example:

```
/P_ADDPP.PGM
/example: ADD8...) = add operation with command nesting
/in byte format
/task: 4+(3*2)=10—>0Ah

ld   bc4 /OP1:=4
add( bc3 /OP2:=2*3=6
mul  bc2
)
st   bo0 /RESULT:=10
ep
```

3.4.2 Subtract Instructions

SUB Subtract.
The SUB instruction subtracts two signed fixed-point numbers. These numbers are represented by operands in the instruction.

Formats permitted: Byte and word

The first operand is the ACC contents and the second operand is the parameter following the command. The result is available in the ACC as RESULT. By this process the former ACC contents are overwritten. In this case the stack contents and depth are unchanged.
When a SUB instruction is to be executed without the declaration of operands, it is undestood that OP1 was loaded into the ACC and OP2 is in the stack. Again the result is available in the ACC as RESULT the former ACC contents having been overwritten. In this case the stack address is incremented.

According to the selection of the operand format (byte or word), an overflow bit, M0.6 = 1, is created by the operating system in the formation of the result if the range of values is exceeded. This overflow bit can be queried after the execution of the operation:

Byte format:	value <-128	M0.6 = 1
Word format:	value <-32768	M0.6 = 1

When the overflow bit occurs, the result must be corrected by appropriate programming steps.

Declaration in byte format:

```
LD    BC100    /OP1: +100
SUB   BC20     /OP2: -20
ST    BO       /RESULT: +80
```

Programming example:
Formation of bit marker M0.6 and indication of result correction

```
/P_SUB.PGM
/example: SUB = subtraction with byte operands
/overflow is filed by the system in bit marker M0.6
/if result <-128 in byte format
/(or if result <-32768 in word format)
/task: -126-(+3) = -129
/output, however, shows 7Fhex = +127, false result
/because range is exceeded —> overflow!

ld   bc-126 /OP1: load constant „-126“ into ACC
sub  bc3 /OP2: subtract constant „+3“
st   bm10 /RESULT: store result in marker byte
ld   m0,6 /load status bit
jmpc overf  /jump when status bit „1“
ld   bm10 /load result
st   bo0 /readout result
jmp  end /unconditional jump to the end
overf: nop  /e.g. correction routine
ld   bm10 /load corrected result
st   bo0 /readout corrected result
 end:                                  ep
 /end of program
```

3.4.3 Multiply Instructions

MUL **Multiply.**

The MUL instruction multiplies two signed fixed-point numbers. These values are represented in the instruction by operands.

Operand permitted: Byte and word

The first operand (OP1) is the ACC contents and the second operand is the parameter following the instruction. The result is available in the ACC as RESULT. By this process the former ACC contents are overwritten. In this case the stack contents and depth are unchanged.
When a MUL instruction is to be executed without the declaration of operands, it is understood that OP1 was loaded into the ACC and OP2 is in the stack. Again the result is available in the ACC as RESULT the former contents of the ACC having been overwritten. In this case the stack address is incremented.

According to the selection of the operand format (byte or word) an overflow bit, M0.5 = 1, is created by the operating system in the formation of the result if the range of values was exceeded. This overflow bit can be queried after the operation:

Byte format:	value <-128	or >+127	M0.5 = 1
Word format:	value <-32768	or >+32767	M0.5 = 1

When the overflow bit occurs, the result must be corrected by appropriate programming steps.

Declaration example in byte format:

```
LD   BC10 /OP1:   +10
MUL  BC8 /OP2: +  8
ST   BO  /RESULT: +80
```

Programming example:
Creation of bit marker M0.5 and indication of the correction of the result

```
/P_MUL.PGM
/example: MUL = multiply operation with byte operands
/overflow is filed by the system in
/bit marker M0.5 if result <-128 or >+127
/(or if result <-32768 or >+32767
/in word processing)
/task: 16*8 = 128
/output, however, shows 80hex = false result
/because range was exceeded —> overflow!

ld   bc16 /OP1:load value „+16“ into ACC
mul  bc8 /OP2: multiply by constant „+8“
st   bm10 /RESULT: store in marker byte
ld   m0.5 /load status byte
jmpc overf  /jump if status bit „1“
ld   bm10 /load result
st   bo0 /readout result
jmp  end /unconditional jump to the end
overf: nop  /e.g. correction routine
ld   bm10 /load corrected result
st   bo0 /readout corrected result
end: ep  /end of program
```

MUL(...) Multiply instruction with nesting.

The MUL instruction with command nesting in brackets executes a multiply operation between operand 1 (OP1) and operand 2 (OP2) with fixed-point numbers.

Operand permitted: Byte and word.

The sequence in the execution is this: First the instructions in the bracket create OP2 as temporary result. Only then is the multiply operation with OP1, which was previously loaded into the ACC, carried out. The result is available in the ACC as RESULT. By this process the former ACC contents are overwritten.

The stack contents and depth are unchanged after the execution of the command. When, however, an operand in the bracket is a value of the stack, the stack address is incremented.

According to the selection of the operand format (byte or word) an overflow bit, M0.5=1, is created by the operating system in the formation of the result if the range of values is exceeded. This overflow bit can be queried after the operation:

Byte format:	value <-128	or >+127	M0.5=1
Word format:	value <-32768	or >+32767	M0.5=1

When the overflow bit occurs, the result must be corrected by appropriate programming steps.

Example, execution of the MUL(...) instruction:

Program listing	Sequence of execution	Coments

`LD   OP1`
`MUL( BI0`
`ADD  BI1`
`)`
	`LD   OP1`
	`LD   BI0`
	`ADD  BI1`
	`MUL  OP1`
		`/(result represents`
		`/(OP2`

The compiler permits up to sixfold nesting of such combinations of commands provided the stack is not overloaded. Maximum stack depth is 8 bytes.

Programming example:

```
/P_MULPP.PGM
/example: MUL(...) = multiply operation with
/command nesting in byte format
/task: 8*(3+2) = 40 –> 28hex

ld   bc8 /OP1
mul( bc3 /(OP2: = 3+2 = 5
add  bc2
)
st   bo0 /result: = 40
 ep
```

3.4.4 Divide Instructions

DIV Divide.
The divide instruction divides two signed fixed-point numbers. These values are represented by operands in the instruction. Occurring digits after the decimal point are cut off without further processing (without rounding).

Operand permitted: Byte and word.

The first operand (OP1) is the ACC contents and the second operand (OP2) is the parameter following the command. The result is available in the ACC as RESULT. By this process the former ACC contents are overwritten. The stack contents and depth are unchanged.

When a DIV instruction is to be executed without the declaration of operands, it is understood that OP1 was loaded into the ACC and OP2 is in the stack. Again the result is available in the ACC as RESULT the former ACC contents having been overwritten. In this case the stack address is incremented.

Independent of the operand format, the operating system creates an overflow bit, M0.4 = 1, in a divide operation by 0. This overflow bit can be queried after the operation.

When the overflow bit occurs, the result must be corrected by appropriate programming steps.

Declaration example in byte format:

LD	BC100	/OP1:	100
DIV	BC20	/OP2:	20
ST	BO	/RESULT:	5

Programming example:
Creation of bit marker M0.4 and indication of result correction

```
/P_DIV.PGM
/example: DIV = divide operation with byte operands
/overflow is filed by the system in bit marker M0.4
/when there is a divide operation by 0
/task: 32:0 = 0
/output, however, shows FFhex = -1, false result
/because operation is illegal!

ld   bc32 /OP1: load value „+32“ into ACC
div  bc0 /OP2: divide operation by constant „0“
st   bm10 /RESULT: store in marker byte
ld   m0.4 /load status byte
jmpc overf  /jump if status byte „1“
ld   bm10 /load result
st   bo0 /readout result
jmp  end /unconditional jump to the end
overf: nop  /e.g. correction routine
ld   bm10 /load corrected result
st   bo0 /readout corrected result
 end:      ep        /end of program
```

DIV(...) Divide with nesting.

The DIV instruction with command nesting in brackets executes a divide operation between operand 1 (OP1) and operand 2 (OP2) with fixed-point numbers. The result is an integer value. Occurring digits after the decimal point are cut off without further processing (rounding).

Operand permitted: Byte and word

The sequence of the operation is this: First the instructions in the bracket create OP2 as temporary result. Only then is the divide operation with OP1, which is previously loaded into the ACC, carried out. The result is available in the ACC as RESULT the former ACC contents having been overwritten.

The stack contents and depth are unchanged after the operation. When, however, an operand in the bracket is a value of the stack, the stack address is incremented.

Independent of the operand format (byte or word), an overflow bit, M0.4 = 1, is created by the operating system. This overflow bit can be queried after the operation.

When the overflow bit occurs, the result must be corrected by appropriate programming steps.

Example, execution of the DIV(...) instruction:

Program listing	Sequence of execution	Coments
LD OP1 DIV(BI0 ADD BI1)	LD OP1 LD BI0 ADD BI1	/(result represents /OP2)

The compiler permits up to sixfold nestimg of such combinations of commands provided the stack is not overloaded. Maximum stack depth is 8 bytes.

Programming examples:

```
  P_DIVPP.PGM
  example: DIV(...) = divide operation with command nesting
  in byte format

  task: 8/(2+2) = 2 —> 02hex

 ld   bc8 /OP1
 div( bc2    /(OP2: = 2+2
 add  bc2    /
 )
 st   bo0    /result: = 2
 ep

  /P_DIVPP.PGM
  /example: DIV(...) = divide operation with twofold
  /command nesting in byte format
  /task: 10/(3+(3-1) = 2 —> 02hex

 ld   bc10 /OP1: = 10
 div( bc3 /OP2: = (3+(3-1)
 add( bc3
 sub  bc1
 )
 )
 st   bo0 /result: = 2
  ep
```

3.5 Compare Instructions

Compare With Signed Operands.
Compare instructions cause the recognition of the value relation between two signed operators to be compared. For this operation the value ranges of signed operands in byte and word formats are to be regarded.

Compare With Unsigned Operands.
Compare instructions cause the recognition of the value relation between two unsigned operators to be compared. For this operation the value ranges of unsigned operands in byte and word formats are to be regarded.

Bit format	<———Signed———> negative	positive	<—Unsigned—> total amount of numbers
	range of numbers		
8 bit	-128 ... -1	0 ... +127	0 ... 255
	80h ... FFh	0h ... +7Fh	0 ... FFh
16 bit	-32768 ... -1	0 ... +32767	0 ... 65535
	8000h ... FFFFh	0h ... +7FFFh	0 ... FFFFh

3.5.1 Compare, Greater

Compare instructions to greater state the value relation to "greater than" between signed operators OP1 and OP".

GT Signed - Compare to „>“.
The GT instruction compares two signed operands and states in the result if operand 1 > operand 2.

Only operands in byte and word formats can be applied.

For the execution of the instruction it is required that OP1 is in the ACC and OP2 is declared in the instruction as a parameter. After the operation the result, which is a Boolean statement (log. 1 or 0), is available in the ACC in bit format for evaluation for jump instructions. The former ACC contents is overwritten. In this case the stack is unchanged.

When no operand (OP2) is declared in the GT instruction, it is understood that OP2 is in the stack. Again the result is available in the ACC as a Boolean value. After the operation the stack address is incremented correspondingly.

Table of compare conditions:

Condition	Result
OP1 > OP2	1
OP1 = OP2	0
OP1 < OP2	0

Declaration example with OP2 declaration in standard form:

```
LDBC4  /OP1: load constant of value 4
GTBC2  /OP2: compare OP1>OP2
```

Declaration example with OP2 as stack operand:

```
LDBC4  /OP1: load constant of value 4 into ACC
LDBC2  /OP2: load constant of value 2 into stack
GTB  /compare OP1>OP2 (stack operand)
```

In operations with a stack operand the allocation of operands (which is which) differs from the general command notation (sequence of operand allocations).

Programming example:

```
 /P_GT.PGM
 /example: GT = greater than
 /
 /For illustration the program sets
 /with OP1>OP2: output bit 0.0 „1“ and
 /with OP1<=OP2: output bit 0.7 „1“.

 ld    bi0 /OP1
 gt    bch04 /OP2
 jmpc  big /jump if „>“
 ld    bch01
 st    bo0 /01h output if „<=“
 jmp   ende
big:
 ld    bch80
 st    bo0 /80h output if „>“
ende:ep
```

UGT **Unsigned compare to „>“.**

The UGT instruction compares two unsigned operands and states in the result if operand 1 > operand 2.

Only operands in byte and word formats can be applied.

For the execution of the instruction it is required that OP1 is in the ACC and OP2 is declared in the instruction as a parameter. After the operation the result, which is a Boolean statement (log. 1 or 0), is available in the ACC in bit format for evaluation for jump instructions. The former ACC contents is overwritten. In this case the stack is unchanged.

When no operand (OP2) is declared in the UGT instruction, it is understood that OP2 is in the stack. Again the result is available in the ACC as a Boolean value. After the operation the stack address is incremented correspondingly.

Table of compare conditions:

Condition	Result
OP1 > OP2	1
OP1 = OP2	0
OP1 < OP2	0

Declaration example:

```
 LD   BC4     OP1: load constant of value 4
 UGT  BC2     /OP2: compare OP1>OP2
```

Programming example:

```
/P_UGT.PGM
/example: UGT = greater than
/
/For illustration the program sets
/">": output byte 0 to F0h.

ld   bc255 /OP1
ugt  bch254/OP2
jmpc big /jump if „>"
ld   bch00 /put off output
st   bo0
jmp  ende
big:
ld   bchf0
s    bo0 /if „>": output F0h
 ende:  ep
```

GT(..) Signed compare to „>" with nesting.

The GT instruction with command nesting in brackets compares two signed operands and states in the result if operand 1 > operand 2.

Operand formats permitted: Byte and word

For the execution of the instruction it is required that operand 1 is the current ACC contents and operand 2 is the temporary result of the instructions in the bracket. After the operation the result in bit format is available in the ACC as a Boolean value (log. 1 or 0). The former ACC contents are overwritten.
The stack contents and depth are unchanged after the operation. When, however, an operand in the bracket is a value of the stack, the stack address is incremented.

The compiler permits up to sixfold nesting do such combinations of commands provided the stack is not overloaded. Maximum stack depth is 8 bytes.

Table of compare conditions

Condition	**Result**
OP1 > OP2	1
OP1 = OP2	0
OP1 < OP2	0

The sequence in the execution of the GT instruction is this: First the operations declared in the bracket are executed. They create OP2.

Declaration example:

Program listing	Sequence of execution	Coments
LD BC4	LD BC4	/OP1: load constant of value 4
GT(BC2	LD BC2	/
ADD BC1	ADD BC1	/OP2: add result: value 3
) GT	B	/RESULT: OP1>OP2 = log. 1

Programming example:

```
/P_GTPP.PGM
/example: GT(...) greater than
/
/if > : RESULT „1“

/For illustration the program sets...
/if <=: output bit 0.0 to „1“,
/if >: output bit 0.7 to „1“

ld   bi0 /OP1 as variable
gt(  bch04  /OP2: = 4+1
add  bch01  /(
)
jmpc groesse /jump if „>“
ld   bch01
st   bo0 /if „<=“
jmp  ende
 groesse:
ld   bch80
st   bo0 /if „>“
 ende: ep
```

UGT(..) Unsigned compare to „>“ with nesting.

The UGT instruction with command nesting in brackets compares two unsigned operands and states in the result if operand 1 > operand 2.

Operand formats permitted: Byte and word

For the execution of the instruction it is required that operand 1 is the current ACC contents and operand 2 is the temporary result of the instructions in the bracket. After the operation the result in bit format is available in the ACC as a Boolean value (log. 1 or 0). The former ACC contents are overwritten.
The stack contents and depth are unchanged after the operation. When, however, an operand in the bracket is a value of the stack, the stack address is incremented.

The compiler permits up to sixfold nesting do such combinations of commands provided the stack is not overloaded. Maximum stack depth is 8 bytes.

Table of compare conditions

Condition	Result
OP1 > OP2	1
OP1 = OP2	0
OP1 < OP2	0

The sequence in the execution of the UGT instruction is this: First the operations declared in the bracket are executed. They create OP2.

Declaration example:

Program listing	Sequence of execution	Coments
LD BC4 UGT(BC2 ADD BC1)	LD BC4 LD BC2 ADD BC1 UGT B	/OP1: load constant of value 4 / /OP2: add result: value 3 /RESULT: OP1>OP2 = log. 1

Programming example:

```
/P_UGTPP.PGM
/example: UGT(...) greater than
/if „>“ : RESULT „1“

/For illustration the program sets...
/if „>=“: output bit 0.0 to „1“,
/if „<„: output bit 0.7 to „1“

ld   bc240  /OP1 as constant
ugt( bc200  /OP2: = 200+3
add  bch39  /(
)
jmpc groesse /jump if „>“
ld   bch01
st   bo0 /if „<=“
jmp  ende
  groesse:
ld   bch80
st   bo0 /if „>“
  ende:ep
```

3.5.2 Compare greater or equal

Compare instructions to greater - equal state the value relation to greater than or equal between signed operators OP1 and OP2, which are to be compared.

GE Signed, compare to „>=“.
The GE instruction compares two signed operands and states in the result if operand 1 >= operand 2.

Operand formats: Byte and word

For the execution of the command it is required that OP1 is in the ACC and OP2 is declared as a parameter in the instruction. The result, which is a Boolean statement (log. 1 or 0), is available in bit format in the ACC after the operation for evaluation for jump instructions. The former ACC contents are overwritten. The stack is unchanged.

When no operand (OP2) is declared in the GE instruction, it is understood that OP2 is in the stack. Again the result is available in the ACC as a Boolean value. After the operation the stack address is incremented correspondingly.

Table of compare conditions:

Condition	Result
OP1 > OP2	1
OP1 = OP2	1
OP1 < OP2	0

Declaration example with OP2 declaration in standard form:

```
LD   BC4 /OP1: load constant of value 4
GE   BC2 /OP2: compare to >= OP1 with OP2
```

Declaration example with OP2 as stack operand:

```
LD     BC2       /OP1: load constant of value 2 into ACC
LD     BC4       /OP2: load constant of value 4 into stack
EQ     B         /compare to >= OP1 with stack operand (OP2)
```

In operations with a stack operand the allocation of operands (which is which) differs from the general command notation (sequence of operand allocations).

Programming example:

```
/P_GE.PGM
/example: GE = greater than or equal
      /
/For illustration the program sets
/if >=: output bit 0.0 „1“ and
/if <: output bit 0.7 „1“.
/
ld    bi0  /OP1
ge    bi1  /OP2
jmpc  groesse /jump if „>=“
ld    bch01
st    bo0  /01h output if „<„
jmp   ende
 groesse:
ld    bch80
st    bo0  /80h output if „>=“
 ende:ep
```

UGE Unsigned compare to „>="

The UGE instruction compares two unsigned operands and states in the result if operand 1 >= operand 2.

Operand formats : Byte and word

For the execution of the command it is required that OP1 is in the ACC and OP2 is declared as a parameter in the instruction. The result, which is a Boolean statement (log. 1 or 0), is available in bit format in the ACC after the operation for evaluation for jump instructions. The former ACC contents are overwritten. The stack is unchanged.

When no operand (OP2) is declared in the UGE instruction, it is understood that OP2 is in the stack. Again the result is available in the ACC as a Boolean value. After the operation the stack address is incremented correspondingly.

Table of compare conditions:

Condition	Result
OP1 > OP2	1
OP1 = OP2	1
OP1 < OP2	0

Declaration example:

```
LD    BC4  /OP1: load constant of value 4
GE    BC2  /OP2: compare to >= OP1 with OP2
```

Programming example:

```
  /P_UGE.PGM
  /example: UGE = greater than or equal

  /For illustration the program sets
  /if „>=“: output byte 0 to F0h.
  /
  ld    bc230   /OP1
  uge   bc230  /OP2
  jmpc  groegl/jump if „>=“
  ld    bch00  /put off output
  st    bo0  /01h output if „<„
  jmp   ende
groegl:
  ld    bcf80
  st    bo0  /if „>=“: output F0h
ende:  ep
```

GE(...) Signed compare to „>=“ with nesting.

The GE instruction with command nesting in brackets compares two signed operands and states in the result if operand 1 >= operand 2.

Operands formats: Byte and word

The execution of the instruction requires that operand 1 is the current ACC contents and operand 2 is the temporary result created by the bracketed instructions. After the operation the result in bit format is available in the ACC as a Boolean statement (log. 1 or 0). The former ACC contents are overwritten.
The stack contents and depth are unchanged after the operation. When, however, an operand in the bracket is a value of the stack, the stack address is incremented.

The compiler permits up to sixfold nesting of such command combinations provided the stack is not overloaded. Maximum stack depth is 8 bytes.

Table of compare conditions:

Condition	Result
OP1 > OP2	1
OP1 = OP2	1
OP1 < OP2	0

The sequence in the execution of the GE(...) instruction is this: First the operations in the bracket are executed. They create OP2.

Declaration example:

Program listing	Sequence of execution	Coments
LD BC4	LD BC4	/OP1: load constant of value 4
GE(BC2	LD BC2	/
ADD BC1	ADD BC1	/OP2: add result: value 3
)	GE B	/RESULT: OP1>=OP2 = log.1

Programming example:

```
  /P_GEPP:PGM
  /example: GE(...) greater than or equal
  /
  /if > and =: RESULT is „1“
  /For illustration the program sets
  /if <: output bit 0.0 to „1“
  /if >=: output bit 0.7 to „1.

  ld    bi0  /OP1 as variable
  ge(   bch04 /(OP2: = 4+2
  add   bch02 /(
  )
  jmpc  groesse /jump if „>=“
  ld    bch01
  st    bo0  /if „<„
  jmp   ende
groesse:
  ld    bch80
  st    bo0  /if „>=“
  ep
```

UGE(...) Unsigned, compare to „>=“ with nesting.

The UGE instruction with command nesting in brackets compares two unsigned operands and states in the result if operand 1 >= operand 2.

Operands formats: Byte and word

The execution of the instruction requires that operand 1 is the current ACC contents and operand 2 is the temporary result created by the bracketed instructions. After the operation the result in bit format is available in the ACC as a Boolean statement (log. 1 or 0). The former ACC contents are overwritten.
The stack contents and depth are unchanged after the operation. When, however, an operand in the bracket is a value of the stack, the stack address is incremented.

The compiler permits up to sixfold nesting of such command combinations provided the stack is not overloaded. Maximum stack depth is 8 bytes.

Table of compare conditions:

Condition	**Result**
OP1 > OP2	1
OP1 = OP2	1
OP1 < OP2	0

The sequence in the execution of the UGE(...) instruction is this: First the operations in the bracket are executed. They create OP2.

Declaration example:

Program listing	Sequence of execution	Coments
LD BC4	LD BC4	/OP1: load constant of value 4
UGE(BC2	LD BC2	/
ADD BC1	ADD BC1	/OP2: add result: value 3
)	UGE B	/RESULT: OP1>=OP2 = log.1

Programming example:

```
 /P_UGEPP:PGM
 /example: UGE(...) greater than or equal
 /if „>“ and „=: RESULT is „1“
 /For illustration the program sets
 /if „<„: output bit 0.0 to „1“
 /if „>=“: output bit 0.7 to „1.

 ld    bc250 /OP1
 uge(  bc255 /(OP2: = 255-5
 sub   bc5  /(
 )
 jmpc  groesse /jump if „>=“
 ld    bch01
 s     bo0  /if „<„
 jmp   ende
groesse:
 ld    bch80
 st    bo0  /if „>=“
ende: ep
```

3.5.3 Compare, equal

Compare instructions to equal state the value relation to equal between signed operands OP1 and OP2.

EQ Compare signed to equal.
The EQ instruction compares two signed operands and states in the result“ if operand 1 = operand 2.

Operand formats: Byte and word

For the execution of the instruction it is required that OP1 is in the ACC and OP2 is declared in the instruction as a parameter. After the operation the result, which is a Boolean statement (log.1 or 0), is available in bit format in the ACC for evaluation for jump instructions. The former ACC contents are overwritten. In this case the stack is unchanged.
When no operand is declared in the EQ instruction, it is understood that OP2 is in the stack. Again the result is available in the ACC as a Boolean value. After the operation the stack address is incremented correspondingly.

Table of compare conditions:

Condition	Result
OP1 > OP2	0
OP1 = OP2	1
OP1 < OP2	0

Declaration example with OP2 declaration in standard form:

```
LD   BC4     /OP1: load constant of value 4
EQ   BC2     /OP2: compare to „=“: OP1 with OP2
```

Declaration example with OP2 as stack operand:

```
LD   BC2   /OP1: load constant of value 2 into ACC
LD   BC4   /OP2: load constant of value 4 into stack
EQ   B       /compare to „=“: OP1 with stack operand (OP2)
```

n operations with a stack operand, the allocation of operands (which is which) differs from the general command notation (sequence of operand allocations).

Programming example:

```
  /P_EQ.PGM
  /example: EQ = equal
  /
  /For illustration the program sets:
  /if „=“: output byte 0 to F0h.
  /
  ld    bi0  /OP1
  eq    bi1  /OP2
  jmpc  gleich/jump if „=“
  nop
  ld    bch00 /put off output
  jmp   ende
gleich:
  ld    bchf0
  st    bo0  /if „=“: output F0h
ende:   ep
```

UEQ **Unsigned compare to „=“**

The EQ instruction compares two unsigned operands and states in the result“ if operand 1 = operand 2.

Operand formats: Byte and word

For the execution of the instruction it is required that OP1 is in the ACC and OP2 is declared in the instruction as a parameter. After the operation the result, which is a Boolean statement (log.1 or 0), is available in bit format in the ACC for evaluation for jump instructions. The former ACC contents are overwritten. In this case the stack is unchanged.

When no operand is declared in the UEQ instruction, it is understood that OP2 is in the stack. Again the result is available in the ACC as a Boolean value. After the operation the stack address is incremented correspondingly.

Table of compare conditions:

Condition	Result
OP1 > OP2	0
OP1 = OP2	1
OP1 < OP2	0

Declaration example:

```
  LD    BC4  /OP1: load constant of value 4
  UEQ   BC2  /OP2: compare to „=“: OP1 with OP2
```

Programming example:

```
  /P_UEQ.PGM
  /example: UEQ = equal (gleich)
  /
  /For illustration the program sets:
  /if „=“: output byte 0 to F0h.
  /
  ld    bc255 /OP1
  ueq   bc255 /OP2
  mpc   gleich/jump if „=“
  nop
  ld    bc0  /put off output
  st    bo0
  jmp   ende
gleich:
  ld    bch0f0
  st    bo0  /if „=“: output F0h
ende:   ep
```

EQ(...) Signed compare to „=“ with nesting.

The EQ instruction with command nesting in brackets compares two signed operands and states in the result if operand 1 = operand 2.

Operand formats: Byte and word

The execution of the instruction requires that operand 1 is the current ACC contents and operand 2 is the temporary result created by the bracketed instructions. After the operation the result in bit format is available in the ACC as a Boolean value (log. 1 or 0). The former ACC contents are overwritten.
The stack contents and depth are unchanged after the operation. When, however, an operand in the bracket is a value of the stack, the stack address is incremented.

The compiler permits up to sixfold nesting of such command combinations provided the stack is not overloaded. Maximum stack depth is 8 bytes.

Table of compare conditions:

Condition	Result
OP1 > OP2	0
OP1 = OP2	1
OP1 < OP2	0

The sequence in the execution of the EQ(...) instruction is this: First the operations in the bracket are executed. They create OP2.

Declaration example:

Program listing	Sequence of execution	Coments
LD BC4	LD BC4	/OP1: load constant of value 4
EQ(BC2	LD BC2	/
ADD BC2	ADD BC2	/OP2: add result: value 4
)	EQ B	/RESULT: OP1=OP2 (=log.1)

Programming example:

```
  /P_EQPP.PGM
  /example: EQ(...) equal
  /gleich
  /if „=“: result is „1“

  /For illustration the program sets
  /if <>: output bit 0.0 to „1“ and
  /if =: output bit 0.7 to „1“

  ld    bi0  /OP1 as variable
  eq(   bch04 /(OP2: = 4+3
  add   bch03 /(
  )
  jmpc  gleich/jump if „=“
  ld    bch01
  st    bo0  /if „<>“
  jmp   ende
 gleich:
  ld    bch80
  st    bo0  /if „=“
  ep
```

UEQ(...) Unsigned compare to „=“ with nesting.

The UEQ instruction with command nesting in brackets compares two unsigned operands and states in the result if operand 1 = operand 2.

Operand formats: Byte and word

The execution of the instruction requires that operand 1 is the current ACC contents and operand 2 is the temporary result created by the bracketed instructions. After the operation the result in bit format is available in the ACC as a Boolean value (log. 1 or 0). The former ACC contents are overwritten.
The stack contents and depth are unchanged after the operation. When, however, an operand in the bracket is a value of the stack, the stack address is incremented.

The compiler permits up to sixfold nesting of such command combinations provided the stack is not overloaded. Maximum stack depth is 8 bytes.

Table of compare conditions:

Condition	Result
OP1 > OP2	0
OP1 = OP2	1
OP1 < OP2	0

The sequence in the execution of the UEQ(...) instruction is this: First the operations in the bracket are executed. They create OP2.

Declaration example:

Program listing	Sequence of execution	Coments
LD BC4	LD BC4	/OP1: load constant of value 4
UEQ(BC2	LD BC2	/
ADD BC2	ADD BC2	/OP2: add result: value 4
)	UEQ B	/RESULT: OP1=OP2 (=log.1)

Programming example:

```
  /P_UEQPP.PGM
  /example: UEQ(...) equal (gleich)
  /if „=“: result is „1“

  /For illustration the program sets
  /if <>: output bit 0.0 to „1“ and
  /if =: output bit 0.7 to „1“


  ld    bc220 /OP1 as constant
  ueq(  bc204 /(OP2: = 20*11
  add   bc11 /(
  )
  jmpc  gleich/jump if „=“
  ld     bch01
  st    bo0  /if „<>“
  jmp   ende
 gleich:
  ld    bch80
  st    bo0  /if „=“
 ende:  ep
```

3.5.4 Compare, less or equal

Compare instructions to less or equal state the value relation to less or equal between signed operands OP1 and OP2, which are to be compared.

LE Signed compare to „<=“.
The LE instruction compares two signed operands and states in the result if operand 1 <= operand 2.

Operand formats: Byte and word

For the execution of the instruction it is required that OP1 is in the ACC and OP2 is declared in the instruction as a parameter. After the operation the result, which is a Boolean statement (log. 1 or 0), is available in bit format in the ACC for evaluation for jump instructions. The former ACC contents are overwritten. The stack is unchanged.

When no operand (OP2) is declared in the LE instruction, it is understood that OP2 is in the stack. Again the result is available in the ACC as a Boolean value. After the operation the stack address is incremented correspondingly.

Table of compare conditions:

Condition	Result
OP1 > OP2	0
OP1 = OP2	1
OP1 < OP2	0

Declaration example with OP2 declaration in standard form:

```
LD    BC2  /OP1: load constant of value 2
LE    BC4  /OP2: compare to <=: OP1 with OP
```

Declaration example with OP2 as stack operand

```
LD    BC4  /OP1: load constant of value 4 into ACC
LD    BC2  /OP2: load constant of value 2 into stack
LE      /compare to <=: OP1 with stack operand (OP2)
```

In operations with a stack operand the allocation of operands (which is which) differs from the general notation of instructions (sequence of operand allocations).

Programming example:

```
   /P_LE.PGM
   /example: LE = less or equal
   /
   /For illustration the program sets
   /if „<=“: output byte 0 to F0h

   ld        bi0         /OP1
   le        bi1         /OP2
   jmpc      klglei      /jump if „<=“
   ld        bch00       /put off output
   st        bo0
   jmp       ende
klglei:
   ld        bchf0
   st        bo0         /if „<=“: output F0h
ende:        ep
```

ULE Unsigned compare to „<=“.

The ULE instruction compares two unsigned operands and states in the result if operand 1 <= operand 2.

Operand formats: Byte and word

For the execution of the instruction it is required that OP1 is in the ACC and OP2 is declared in the instruction as a parameter. After the operation the result, which is a Boolean statement (log. 1 or 0), is available in bit format in the ACC for evaluation for jump instructions. The former ACC contents are overwritten. The stack is unchanged.

When no operand (OP2) is declared in the ULE instruction, it is understood that OP2 is in the stack. Again the result is available in the ACC as a Boolean value. After the operation the stack address is incremented correspondingly.

Table of compare conditions:

Condition	Result
OP1 > OP2	0
OP1 = OP2	1
OP1 < OP2	1

Declaration example with OP2 declaration in standard form:

```
  LD    BC2  /OP1: load constant of value 2
  LE    BC4  /OP2: compare to <=: OP1 with OP
```

Declaration example:

```
  LD    BC2  /OP1: load constant of value 2
  ULE   BC4  /OP2: compare to <=: OP1 with OP2
```

Programming example:

```
  /P_ULE.PGM
  /example: ULE = less or equal
  /
  /For illustration the program sets
  /if „<=“: output byte 0 to F0h

  ld    bc254 /OP1
  ule   bc255 /OP2
  jmpc  klglei/jump if „<=“
  ld    bch00 /put off output
  st    bo0
  jmp   ende
klglei:
  ld    bchf0
  st    bo0  /if „<=“: output F0h
ende:   ep
```

LE(...) Signed compare to „<=“ with nesting.

The LE instruction with command nesting in brackets compares two signed operands and states in the result if operand 1 <= operand 2.

Operand formats: Byte and word

For the execution of the instruction it is required that operand 1 is the current ACC contents and operand 2 is the temporary result created by the instructions in the bracket. After the operation the result in bit format is available in the ACC as a Boolean value (log. 1 or 0). The former ACC contents are overwritten.

The stack contents and depth are unchanged after the execution of the instruction. When, however, an operand in the bracket is a value of the stack, the stack address is incremented.

The compiler permits up to sixfold nesting of such command combinations provided the stack is not overloaded. Maximum stack depth is 8 bytes.

Table of compare conditions:

Condition	**Result**
OP1 > OP2	0
OP1 = OP2	1
OP1 < OP2	0

The sequence in the execution of the LE(...) instruction is this: First the operations declared in the bracket are executed. They create OP2.

Declaration example:

Program listing	Sequence of execution	Coments
LD BC3	LD BC3	/OP1: load constant of value 3
LE(BC2	LD BC2	/
ADD BC2	ADD BC2	/OP2: add result: value 4
)	LE B	/RESULT: OP1<=OP2 =log.1

Programming example:

```
/P_LEPP.PGM
/example: LE(...) less or equal
/
/if „<=“: RESULT is „1“

/For illustration the program sets
/if „>“: output bit 0.0 to „1“ and
/if „<=“: output bit 0.7 to „1“

ld    bi0  /OP1 as variable
le(   bch04 /(OP2: = 4+4
add   bch04 /(
)
jmpc  klglei/jump if „<=“
ld    bch01
st    bo0  /if „>“
jmp   ende
klglei:
ld    bch80
st    bo0  /if „<=“
ep
```

ULE(...) Instruction - compare to „<=“ with nesting

The ULE instruction with command nesting in brackets compares two unsigned operands and states in the result if operand 1 <= operand 2.

Only operand in byte and word formats can be applied.

For the execution of the instruction it is required that operand 1 is the current ACC contents and operand 2 is the temporary result created by the instructions in the bracket. After the operation the result in bit format is available in the ACC as a Boolean value (log. 1 or 0). The former ACC contents are overwritten.

In this case the stack contents and depth are unchanged after the execution of the instruction. When, however, an operand in the bracket is a value of the stack, the stack address is incremented.

The compiler permits up to sixfold nesting of such command combinations provided the stack is not overloaded. Maximum stack depth is 8 bytes.

Table of compare conditions:

Condition	Result
OP1 > OP2	0
OP1 = OP2	1
OP1 < OP2	1

The sequence in the execution of the ULE(...) instruction is this: First the operations declared in the bracket are executed. They create OP2.

Declaration example:

Program listing Sequence of executionComents

Program listing	Sequence of execution	Coments
LD BC3	LD BC3	/OP1: load constant of value 3
ULE(BC2	LD BC2	/
ADD BC2	ADD BC2	/OP2: add result: value 4
)	ULE B	/RESULT: OP1<=OP2 =log.1

Programming example:

```
 /P_ULEPP.PGM
 /example: ULE(...) less or equal
 /if „<=“: RESULT is „1“

 /For illustration the program sets
 /if „>“: output bit 0.0 to „1“ and
 /if „<=“: output bit 0.7 to „1“

 ld         bc124      /OP1: = 124
 ule(       bc250      /(OP2: = 250 div 2 = 125
 div        bc2 4      /(
 )
 jmpc       klglei     /jump if „<=“
 ld         bch01
 st         bo0        /if „>“
 jmp        ende
klglei:
 ld         bch80
 st         bo0        /if „<=“
ende:       ep
```

3.5.5 Compare, less than

Compare instructions to less state the value relation to less between signed operands OP1 and OP2, which are to be compared.

LT Signed compare to „<„.

The LT instruction compares two signed operands and states in the result if operand 1 < operand 2.

Operand formats: Byte and word

For the execution of the instruction it is required that OP1 is in the ACC and OP2 is declared in the instruction as a parameter. After the operation the result, which is a Boolean statement (log. 1 or 0), is available in bit format in the ACC for evaluation for jump instructions. The former ACC contents are overwritten. The stack is unchanged.

When no operand (OP2) is declared in the LT instruction, it is understood that OP2 is in the stack. Again the result is available in the ACC as a Boolean statement. After the operation the stack address is incremented correspondingly.

Table of compare conditions:

Condition	Result
OP1 > OP2	0
OP1 = OP2	1
OP1 < OP2	0

Declaration example with OP2 declaration in standard form:

```
LD    BC2  /OP1: load constant of value 2
LT    BC4  /OP2: compare to „<„: OP1 with OP2
```

Declaration example with OP2 as stack operand:

```
LD    BC4  /OP1: load OP1 of value 4 into ACC
LD    BC2  /OP2: load constant of value 2 into stack
LT    B /compare to „<„: OP1 with stack operand (OP2)
```

In operations with a stack operand the allocation of operands (which is which) differs from the general command notation (sequence of operand allocations).

Programming example:

```
  /P_LT.PGM
  /example: LT = less than
  /
  /For illustration the program sets
  /if „<„: output byte 0 to F0h
  /
  ld    bi0  /OP1
  lt    bi1
  jmpc  kleiner /jump if „<„
  ld    bch00 /put off output
  st    bo0
  jmp   ende
kleiner:
  ld    bchf0
  st    bo0  /if „<„: output F0h
ende:   ep
```

ULT Unsigned compare to "<".

The ULT instruction compares two unsigned operands and states in the result if operand 1 < operand 2.

Operand formats: Byte and word

For the execution of the instruction it is required that OP1 is in the ACC and OP2 is declared in the instruction as a parameter. After the operation the result, which is a Boolean statement (log. 1 or 0), is available in bit format in the ACC for evaluation for jump instructions.
The former ACC contents are overwritten. The stack is unchanged.

When no operand (OP2) is declared in the ULT instruction, it its understood that OP2 is in the stack. Again the result is available in the incremented correspondingly.

Table of compare conditions:

Condition	Result
OP1 > OP2	0
OP1 = OP2	0
OP1 < OP2	1

Declaration example:

```
  LD    BC2  /OP1: load constant of value 2
  ULT   BC4  /OP2: compare to "<": Op1 with OP2
```

Programming example:

```
  /P_ULT.PGM
  /example:ULT=less than
  /
  /For illustration the program sets
  /if"<": output  byte 0 to F0h
  /ld   bc254 /OP1
  ult   bc256 /OP2
  jmpc  kleiner /jump if "<"
  ld    bch00 /put off output
  st    bc0
  jmp   ende
kleiner:
  ld    bchf0
  st    bc0  /if"<": output F0h
ende    ep
```

LT(...) Signed compare to „<„ with nesting.

The LT instruction with command nesting in brackets compares two signed operands and states in the result if operand 1 < operand 2.

Operand formats: Byte and word

For the execution of the instruction it is required that OP1 is the current ACC contents and OP2 is the temporary result created by the instructions in the bracket. After the operation the result in bit format is available in the ACC as a Boolean value (log. 1 to 0). The former ACC contents are overwritten.
The stack contents and depth are unchanged after the operation. When, however, an operand in the bracket is a value of the stack, the stack address is incremented.

The compiler permits up to sixfold nesting of such command combinations provided the stack is not overloaded. Maximum stack depth is 8 bytes.

Table of compare conditions:

Condition	Result
OP1 > OP2	0
OP1 = OP2	0
OP1 < OP2	1

The sequence in the execution of the LT(...) instruction is this: First the operations declared in the bracket are executed. They create OP2.

Declaration example:

Program listing	Sequence of execution	Coments
LD BC3	LD BC3	/OP1: load constant of value 3
LT(BC2	LD BC2	/
ADD BC2	ADD BC2	/OP2: add result: value 4
)	LT B	/RESULT: OP1<OP2 =log.1

Programming example:

```
  /P_LTPP.PGM
  /example: LT(...) = less than
  /
  /if „<„: result = „1“

  /For illustration the program sets
  /if „>=“: output bit 0.0 to „1“
  /if „<„: output bit 0.7 to „1“

  ld    bi0  /OP1 as a variable
  lt(   bch04 /(OP2: = 4+5
  add   bch05 /(
  )
  jmpc  kleiner /jump if „<„
  ld    bch01
  st    bo0  /if „>=“
  jmp   ende
kleiner:
  ld    bch80
  st    bo0  /if „<„
  ep
```

ULT(...) Unsigned compare to „<„ with nesting.

The ULT instruction with command nesting in brackets compares two signed operands and states in the result if operand 1 < operand 2.

Operand formats: Byte and word

For the execution of the instruction it is required that OP1 is the current ACC contents and OP2 is the temporary result created by the instructions in the bracket. After the operation the result in bit format is available in the ACC as a Boolean value (log. 1 to 0). The former ACC contents are overwritten.
The stack contents and depth are unchanged after the operation. When, however, an operand in the bracket is a value of the stack, the stack address is incremented.
The compiler permits up to sixfold nesting of such command combinations provided the stack is not overloaded. Maximum stack depth is 8 bytes.

Table of compare conditions:

Condition	Result
OP1 > OP2	0
OP1 = OP2	0
OP1 < OP2	1

The sequence in the execution of the LT(...) instruction is this: First the operations declared in the bracket are executed. They create OP2.

Declaration example:

Program listing	Sequence of execution	Coments
LD BC3	LD BC3	/OP1: load constant of value 3
LT(BC2	LD BC2	/
ADD BC2	ADD BC2	/OP2: add result: value 4
)	LT B	/RESULT: OP1<OP2 =log.1

Programming example:

```
  /P_LTPP.PGM
  /example: LT(...) = less than
  /if „<„: result = „1“

  /For illustration the program sets
  /if „>=“: output bit 0.0 to „1“
  /if „<„: output bit 0.7 to „1“

  ld    bi0  /OP1 as a variable
  lt(   bch04 /(OP2: = 4+5
  add   bch05 /(
  )
  jmpc  kleiner /jump if „<„
  ld    bch01
  st    bo0  /if „>=“
  jmp   ende
 kleiner:
  ld    bch80
  st    bo0  /if „<„
 ende:  ep
```

MEQ Masked Equal

MEQ (macro) checks if masked first variable is equal to the second variable. Set the result if they are equal.

Usage:
MEQ FrstVar Mask ScndVar,Rslt
MEQ FrstVar,Mask,ScndVar,Rslt

Number of parameters: 4.

Parameters description:

"FrstVar" - A first variable to be compared.
Type of Parameter: BYTE or WORD.
Location For the Parameter: INPUT or OUTPUT or MEMORY or CONSTANT.

"Mask" - The Mask used in comparison.
Type of Parameter: BYTE or WORD.
Location For the Parameter: INPUT or OUTPUT or MEMORY or CONSTANT.

"ScndVar" - A second variable to be compared.
Type of Parameter:BYTE or WORD.
Location For the Parameter: INPUT or OUTPUT or MEMORY or CONSTANT.

"Rslt" - A bit variable, where the result of comparison is stored.
The result is set True (1) if FirstVar & Mask is equal to ScndVar.
If FirstVar & Mask is not equal to ScndVar the Rslt should be False (0).
Type of Parameter: BIT.
Location For the Parameter: OUTPUT or MEMORY.

MEQ Usage is equivalent to the following BCON S program:

```
LD      FrstVar
AND     Mask
EQ      ScndVar
ST      Rslt
```

Stack Depth for macro - 2 BYTEs

Examples:

```
Main:
    BIT    Equal
    MEQ    bi02,bc16#0f,bc11,Equal
    // If byte input 02 & 0fhex is equal to 11,
    // than set Equal
    LD     Equal
    Jmpc Lab1       //      Jump      if      equal      to      label      Lab1
    .......
Lab1:     EP
```

LIM In Limit Check

LIM (macro) checks if value is in limits.

Usage:

LIM FrstVar LowVal HighVal,Rslt
LIM FrstVar,LowVal,HighVal,Rslt

Number of parameters: 4

Parameters description:

"FrstVar" - The variable to be checked for the limits.
Type of Parameter: BYTE or WORD.
Location For the Parameter: INPUT or OUTPUT or MEMORY.

"LowVal" - Low Limit in In Limit Check.
Type of Parameter: BYTE or WORD.
Location For the Parameter: INPUT or OUTPUT or MEMORY or CONSTANT.

"HighVal" - High Limit in In Limit Check.
Type of Parameter: BYTE or WORD.
Location For the Parameter: INPUT or OUTPUT or MEMORY or CONSTANT.

"Rslt" - The result value for In Limit Check. The Rslt var is Reset (False-0) if FrstVar is greater than low value LowVal and less than high value HighVal. The Rslt var is
Set (True - 1) otherwise.
Type of Parameter: BIT.
Location For the Parameter: OUTPUT or MEMORY.

LIM Usage is equivalent to the following BCON S program:

```
LD      FrstVar
LT      LowVal
LD      FrstVar
GT      HighVal
OR
STN     Rslt
```

Stack Depth for macro - 2 BYTEs

Example:

```
Main:
 BIT    Alarm
 LIM    bi02,bc16#0f,bc11,Alarm
        // If byte input 02 is Greater than 16 or less
        // than 11, than set Alarm
 EP
```

6 Jump Instructions

Jump instructions enable the interruption of the running program and its resumption at the target jump label, which has to be declared as a parameter in the jump instruction.

There are two groups of jump instructions:

- unconditional jump and
- jump dependent on a condition

3.6.1 Unconditional Jump

JMP Jump.

The JMP instruction continues the execution of the command in the place where its jump label is before an instruction in the program.
The execution of the JMP command does not depend on a condition. So the previous history, i.e. the stack contents, is irrelevant to the execution of the command.

Declaration of the instruction:

```
JMP LABEL1 /unconditional jump to LABEL1
```

The jump label can have a length of 8 characters. The first character must be a letter. The other characters can be letters and/or figures. The definition of the label has to be terminated by a colon (:).
After the execution of the command the stack contents and depth as well as the ACC contents are unchanged.

Programming example:

```
      P_JMP.pgm
      /example: „JMP“ - unconditional jump

label0:
      ld   m10.0 /jump target „label0“
      xor  i0.1
      st   o0.0
      jmp  label2/unconditional jump to „label2“

label1:    /jump target „label1“
      ld   i1.0
      o i1.1
      st   o0.7
      jmp  label0/unconditional jump to „label0“


label2:    /jump target „label2“
      ld   o0.0
      and  i0.2
      st   m10.0
      ep
```

3.6.2 Conditional Jump

JMPC Conditional jump, log. 1 ("True")

The JMPC instruction resumes the execution of the program in the place where its jump label is before an instruction in the program.
The execution of the jump instruction depends on the condition that the ACC contents takes the Boolean value „true“, i.e. log1, as the result of previous operations. So the previous history, i.e. the ACC contents, is relevant to the execution of this command.

Declaration of the instruction:

```
JMPC  LABEL1/jump to LABEL1 on condition „true“
```

The jump label can have a length of 8 characters. The first character must be a letter. The other characters can be letters and/or figures. The label definition has to be terminated by a colon (:).
After the execution of the command the former ACC contents are lost and a new value is automatically loaded from the stack into the ACC. The stack has changed. Its depth is incremented.

The conditional jump instruction can only be derived from an operation which created the ACC contents in bit format.

Programming example:

```
      /P_JMPC.PGM
      /example: „JMPC“ jump on condition „true“

label0:    /jump target „label0“
      ld   i0.0
      xor  i0.1
      st   m10.0
      ld   m10.0
      jmpc label2/jump to label2 if ACC „true“


label1:    /jump target „label1“
      ld   m10.0
      or   i1.1
      st   o0.7
      jmp  label0/unconditional jump to „label0“

label2:    /jump target „label2“
      ld   m10.0
      st   o0.0
      ep
```

JMPCN **Conditional jump, Log.0** („false“).

The JMPCN instruction resumes the execution of the command in the place where its jump label is before an instruction in the program.
The execution of the JMPCN instruction depends on the condition that the ACC contents take the Boolean value „false“, i.e. log.0, as a result of previous operations. So the previous history, i.e. the contents of the ACC, is relevant to the execution of this command.

Declaration of the instruction:

```
JMPCN LABEL1/jump to LABEL1 on condition „false“
```

The jump label can have a length of 8 characters. The first character must be a letter. The other characters can be letters and/or figures. The definition of the label has to be terminated by a colon (:).
After the execution of the instruction the former ACC contents are lost and a new value is automatically loaded from the stack into the ACC. The stack has changed. Its depth is incremented.

The conditional jump instruction can only be derived from a previous operation which created the ACC contents in bit format.

Programming example:

```
/P_JMPCN.PGM
      /example: jump on condition „false“

label0:     /jump target „label0“
      ld   i0.0
      xor  i0.1
      st   m10.0
      ld   m10.0
      jmpcn label2/jump to „label2“ if...
           /ACC contents „false“ (log.0)


label1:     /jump target „label1“
      ld   m10.0
      or   i1.1
      st   o0.7
      jmp  label0/unconditional jump to „label0“

label2:     /jump target „label2“
      ld   c0
      st   o0.7
        ep
```

3.7 Auxiliary / Special Instructions

Duplicate Instructions
Duplicate instructions provide for the programmer the opportunity to temporarily store the ACC contents direct or by manipulation (negation) into the stack in order to re-enter this value in further operations.

3.7.1 Duplicate

DUP Duplicate.
The DUP instruction duplicates the current ACC contents into the stack. The parameter following the command defines the data format to be processed (bit, byte or word). It must be in accordance with the preceding load operation.

All operand formats and types are permitted.

The execution of the instruction causes the current ACC contents to be duplicated (copied) into the stack. After the operation it is simultaneously available in the ACC and in the stack. The stack depth is decremented because the ACC contents was batched.

Declaration example in byte format:

```
LD   BC4  /load constant of value 4 into ACC
DUP  B    /duplicate ACC value into stack
```

Programming example:

```
/P_DUP.PGM
/example: DUP = duplicate the ACC value into the stack
/task: 4+4=8
ld    bi0  /OP1: = e.g. 04h
dup   b /duplicate 04h into stack as OP2
add   b /add OP1 to OP2
st    bo0  /RESULT: =  8
ep
```

DUPN Duplicate with negation.

The DUPN instruction duplicates the inverted current ACC contents into the stack.

The parameter following the command defines the data format to be processed (bit, byte or word). It must be in accordance with the preceding load operation.

All operand formats and types are permitted.

The execution of the instruction causes the inverted current ACC contents to be duplicated (copied) into the stack. Thereby the unchanged ACC contents and the corresponding inverted value in the stack are available after the operation for further use. The stack depth is decremented because the ACC contents was batched.

Note: Constants cannot be inverted!

Declaration example in byte format:

```
LD   BI0 /load input byte 0 e.g. 80h into ACC
DUPN B /duplicate value 7Fh into stack
```

Programming example:

```
/P_DUPN.PGM
/example: DUPN = duplicate the ACC value negated
/into the stack
ld    i0.0  /OP1: = e.g. „1“
dupn        /OP2: duplicate „1“ as „0“ into stack
and         /XOR conjunction OP1 with OP2
st    o0.0  /RESULT: = „1“
ep
```

DUP(...) Duplicate with nesting.

The DUP instruction with command nesting in brackets executes the duplication of the operation result resulting from the command nesting in brackets according to priority rules.
All operand formats and types are permitted.

Declaration example:

```
DUP(  I0.2 /AND conjunction I0.2 with I0.3, result of the
And   I0.3 /bracket is duplicated into stack
)
```

The sequence of the execution of the instruction is this: First the operations declared in the bracket are executed.

Example: Execution of DUP(...) instruction:

Program listing	Sequence of execution	Coments
DUP(C1 AND I0.1)	LD C1 LD I0.1 DUP	/result AND conjunction is /duplicated into stack /

Programming example:

```
/P_DUPPP.PGM
/example: DUP(...) = duplicate conjunction of bracket
/as ACC value into stack
/for the assumption of the variable value cf. comment lines

dup   i0.0 /load log. 1 and carry out AND conjunction
and   i0.1  /with log. 1
(     /load result into stack
st    o0.0  /RESULT1: = log. 1
ld    i0.2 /log. 0
and
st    o0.7 /RESULT2: = log.0
ep
```

DUPN(...) Duplication with negation and nesting.
The DUPN instruction with command nesting in brackets duplicates the negated operation result resulting from the command nesting in the bracket according to priority rules.

All operand formats and types are permitted.

Declaration example:

```
DUPN( I0.2 /AND conjunction I0.2 with I0.3, inverted result of
AND   I0.3 /the bracket is duplicated into stack
)      /
```

The sequence in the execution of the instruction is this: First the operations in the bracket are executed.

Example, execution of DUP(...) instruction:

Program listing	Sequence of execution	Coments
DUPN(C1	LD C1	/result AND conjunction is
AND I0.1	AND I0.1	/duplicated into stack
)	DUPN	/

Programming example:

```
/P_DUPNPP.PGM
/example: DUPN(...) = duplicate negated and conjunction
/of bracket expression of the stack with i0.2
/for the assumption of the variable value cf. comment line

dupn(  i0.0 /load log.1 and AND conjunction with
and  i0.1 /log.0
)      /load result into stack
st   o0.0 /RESULT1: = log. 1
ld   i0.2 /log. 1
and    /stack operand (RESULT1)
st   o0.7 /RESULT2: = log. 1
 ep
```

3.7.2 No Operation Instruction

NOP No operation.
The NOP command is a special instruction of command size. It causes no operational activities and has the function of a placeholder for planned instructions.

Declaration example:

```
NOP   /no operation
```

3.7.3 End of Program Instruction

EP End of program.
The EP command is an instruction that indicates the end of the program.
Every program must be terminated by „EP“.

The EP command can be applied only once in a program.

Declaration example:

```
EP    /end of program
```

3.7.4 Move Instructions

MOV Move Value

MOV (macro) moves from source to destination.

Usage:

MOV	SrcVar DstVal
MOV	SrcVar,DstVal

Number of parameters: 2.

Parameters description:

"SrcVar" - Source variable from where the value is moved.
Type of Parameter: BYTE or WORD or BIT.
Location For the Parameter: INPUT or OUTPUT or MEMORY or CONSTANT.

"DstVar" - Destination variable where the value is moved.
Type of Parameter: BYTE or WORD or BIT.
Location For the Parameter: OUTPUT or MEMORY.

MOV usage is equivalent to the following BCON S program:

```
LD          SrcVar
ST          DstVar
```

Stack Depth for macro - 2 BYTEs

Examples:

```
Main:
   MOV BC22,BM55     // Move constant byte 22 to byte memory location 55
   MOV BM22,BM55     // Move byte memory location 22 to byte memory location 55
   MOV WM22,WO55     // Move word memory location 22 to word output location 55
   EP
```

MVM Move Masked

Move a value from masked source to destination.

Usage:

MVM	FrstVar MSK LowVal
MOV	FrstVar,MSK,LowVal

Number of parameters: 3.

Parameters description:

"SrcVar" - Source variable from where the value is moved.
Type of Parameter: BYTE or WORD or BIT.
Location For the Parameter: INPUT or OUTPUT or MEMORY or CONSTANT

"MSK"	- The mask used in Move Masked MVM. Type of Parameter: BYTE or WORD or BIT. Location For the Parameter: INPUT or OUTPUT or MEMORY or CONSTANT
"DstVar"	-Destination variable where the value is moved. The destination variable DstVar is equal to SrcVar & MSK. Type of Parameter: BYTE or WORD or BIT. Location For the Parameter: OUTPUT or MEMORY.
MVM	Usage is equivalent to the following BCON S program: LD SrcVar AND MSK ST DstVar

Stack Depth for macro - 2 BYTEs

Examples:

```
Main:
  MVM    BM33,BC22,BM55     // Get byte memory 33, mask with constant byte 22
                            // and move to byte memory location 55
  MVM    BM33,BI22,BM55     // Get byte memory 33, mask with byte input 22
                            // and move to byte memory location 55
  EP
```

3.7.5 BLOCK MOVE (Remote Master and B-CON Master only)

The COPYB and COPYW instruction can be used to move multiple bytes or words from one I/O or memory area to another, e.g.: moving all inputs from one slave to outputs on another slave.

COPYB **Move Multiple Bytes.**

To copy a number of bytes you can use command

COPYB byte_source, Byte_destination, Byte_count

Example:

```
Main:
  copyb bio, bm10, bc5
  EP
```

Copies 5 bytes from input 0 to memory 10

COPYW Move Multiple Words.

To copy a number of words you can use command

COPYW word_source, word_destination, Byte_count

Example:

```
  Main:
      copyw  S2wi4000, S3wo4000, bc20
      EP
```

Copies 20 bytes (10 words) from slave 2 to slave 3 (ZI0-ZI9 to ZO0-ZO9).

3.7.6 Function / Subroutines

A subroutine (function) is defined as program organisation unit which, when executed, yields exactly one data element. Function do not contain any internal state information, i.e. invocation of a function with the same arguments will always give the same result / output.

Declaration:
FUNCTION <FUNC_TYPE> <FUNC_NAME> (TYPE>Par1, <TYPE> Par2,..,<TYPE> Par6)

The keyword is FUNCTION,
followed by TYPE identifier, specifying type of the value returned by the function,
followed by NAME identifier, specifying the name of the function, declared,
followed by an opening parantezis followed by function's input parameter(s) with them types and closing parantezis.
A function body, must specify the operations to be performed upon the input parameter(s).
An optional operator RET is used to return control to calling program, before reaching terminating keyword.
The terminating keyword is ENDFUNC.

The function return value is of FUNCTION type and is returned as current result in Accumulator.
Function type <FUNC_TYPE> can be: VOID, BIT, BYTE and WORD
Function parameters can be maximum 6.

Maximum 8 functions can be defined.
The function works with a copy of formal parameters, so the value of actual parameters is not changed.
Functions are not re-entrant, but one function can call another.
Functions are invoked via the CALL operator as follows:

CALL FUNC_NAME (list of actual parameters)

Programming Example 1

```
#define temp bm 10
#define tempw wm 11

function byte inc (byte argw)
    ld          argw
    add         bc1
    endfunc

function word power (word argw, byte pow)

    ld          pow
    lt          bc0
    jmpcn       cont
    ld          wc0
    ret

cont:
    ld          pow
    eq          bc0
    jmpcn       cont1
    ld          wc1
    ret

cont1:
    ld          argw
    st          tempw
    ld          pow
    st          temp

loopp:
    ld          temp
    sub         bc1
    dup         b
    st          emp
    eq          bc0
    jmpc        endf
    ld          tempw
    mul         argw
    st          tempw
    jmp         loopp

endf:
    ld tempw
    endfunc
```

```

//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *//

    main:
        call        inc (bo0)
        st          bo0
        call        power (wi0, bo0)
        st          wo2
        ep
```

Programing Example 2

```
/EXAMPLE using subroutines
/use 1 UCB-16DIO master plus 1 UCT-42 display


/function „scale“ converts 12 bit analog input to scale 0-100
/the function is defined in the 5 next instruction lines below

function word scale (word input)
   ld        input          /input parameter
   mul       wc8
   div       wc327
   endfunc                  /end of function (return)

/function „and4“ and’s 4 inputs and returns the result
/the function is defined in the 6 next instruction lines below

function bit and4 (bit inputa, bit inputb, bit inputc, bit inputd)
   ld        inputa         /input 1
   and       inputb         /input 2
   and       inputc         /input 3
   and       inputd         /input 4
   endfunc   /end of function (return)

main:

   mov       m.wi0 dis.wo4000          /select text no.
   call      scale (m.wi2000)          /subroutine
   st        dis.wo4002                /display value

   call      and4 (i0.0 i0.1 i0.2 i0.3)   /call subroutine
   st        o0.0                         /store output
   ep
```

3.7.8 Special Instructions

CODE Insert Assembler Code in Program

Code is used for special purpose if a user function which not already supported by the instructions in B-CON should performed. Code uses normal assembler code instructions available for the 8051-family micro controllers.
For detail information refer to data material for the micro controller.

Number of parameters: less than 7.

Usage:

CODE Par1 [Par2 ...[Par7]]

Parameter description:

"ParX" - Code to be inserted in the program.
Type of the Parameter: BYTE or WORD
Location For the Parameter: CONSTANT.

Stack Depth - depends on user definition, but no stack increase or decrease is enabled.
WARNING: **The assembler instruction must complete in one line.**
Using assembler code should be handled with care as it can easily cause disarstarous operation if not knowing exactly how and what to do.

Examples:

```
Main:
     LD     WC0
     ST     WM33
     CODE   BC16#12 WC16#9000      // LCALL 09000h
     CODE   BC16#90 WC16#2000      // Mov dptr,#2000h
     CODE   BC16#0E0               // Movx a,@dptr
     CODE   BC4                    // Inc a
     CODE   BC16#0f0               // Movx @dptr,a
     EP
```

LOG **Log instruction** (Brodersen RTU8/RTU-COM and BITBUS DL slaves)

The RTU8 module includes a log buffer which works as a ring buffer (FIFO).

All types of inputs and outputs as well as memory markers can be logged either cyclicly (time trigged) or on certain conditions (event trigged).
The logging is carried out Wordwise. E.g. WI0 or WM43.

The RTU8 includes a real time clock and the Memory marker BM17 includes time triggers:

M17.0	0.1 second
M17.1	1 second
M17.2	10 seconds
M17.3	1 min.
M17.4	10 min
M17.5	1 hour
M17.6	10 hours

The Syntax for the LOG instruction is:

log <trigger>,<logid>,<first word to log>,<number of words to log>

Trigger: When "trigger" has the value boolean 1, the logging starts.

Logid: Is the "identifier" of the log and can have a constant value from 0 to 31.

First word to log: Is the first Input word, Output word or Memory marker word which will be logged with the chosen log identifier.

Number of words to log: Is the total number of words including "first word to log" to log. Maximum is 120 words.

Programming example:

```
main:


     /Event log
     log i0.0, bc0,wi2000,bc4
     /Logs the first 4 Analogue inputs each time
     input i0.0 is activated.

     /Cyclic log
     log m17.3,bc1,wi0,bc1
     /Logs the first digital input word every
     minute
     .
     .
     ep
```

3.8 TIMERS (Only in slaves – BITBUS slaves and RTU8/RTU-COM/RTU-870)

TMRx Timer is a routine integrated in the operating system of B-CON S Controllers. It does not produce a return code.

The invitation is carried out by a function name and necessary arguments as pass parameters. The function name consists in a short format followed by the arguments. The arguments are put in brackets (not compulsory!) and separated from each other by separator characters such as comma or SPACE.

The TMRx functions provide the opportunity of a single function invitation by being declared in an initializing part.

The general function cue has the following form:

```
F_name [(][arg1[,arg2[,...,argn]]][)]
```

Explanations:

F_name	function name for inviting and executing the function
arg1,arg2,...argn	function arguments prescribed by the corresponding function
[...]	elements in these brackets are optional arguments of a macro

TMRx - Universal timer.

(Pulse Timer, On-Delay Timer, Off-Delay Timer)
This function creates up to 4 timers which work independent of each other. Each timer can be selected for a special function task. One can choose among 3 function types of timers which can be configured in the argument TYPE by the selection of a figure.

The following sequence is necessary:

```
TMRx Type Inp Out UpVal
```

TMRx function name for timer.
x = index of No, to select between 1 and 4
number: 4

Type defines the type of the timer
Type = 0 —> edge recognition
Type = 1 —> on delay
Type = 2 —> off delay
data format: byte
data type: constant

Inp activation input of the timer
data format: bit
data type: input, output, marker

Out timer output
data format: bit
data type: output, marker

UpVal delay time (time value)
data format: word
data type: marker, constant
measuring unit: 10ms step
value range: 0...65535

As described above the timer function allows the definition of up to 4 timers in a program. By their type assignment they can be configured for different tasks. Their working is simultaneous.
In the following programming example the simultaneous operation of the 3 different timer types is illustrated:

Programming example:

```
/P_TMR:PGM
Function: timer in simultaneous operation

#define   type0 bc0   /timer as edge recognition
#define   inp0  i0.0 /signal input
#define   out0  o0.0 /signal output
#define   upval0     wc100    /pulse duration = 1000 ms (100x10)

#define   type1 bc1   /timer as on delay
#define   inp1  i0.1 /signal input
#define   out1  o0.1 /signal output
#define   upval1     wc150    /pulse duration = 1500 ms (150x10)

#define   type2 bc2   /timer as off delay
#define   inp2  i0.2 /signal input
#define   out2  o0.2 /signal output
#define   upval2     wc200    /pulse duration = 2000 ms (200x10)

/────────────program─────────────

tmr1  type0 inp0 out0 upval0             /timer1 as edge recognition
tmr2  type1 inp1 out1 upval1             /timer2 as on delay
tmr3  type2 inp2 out2 upval2             /timer3 as off delay
ep
```

3.8.1 Edge Recognition

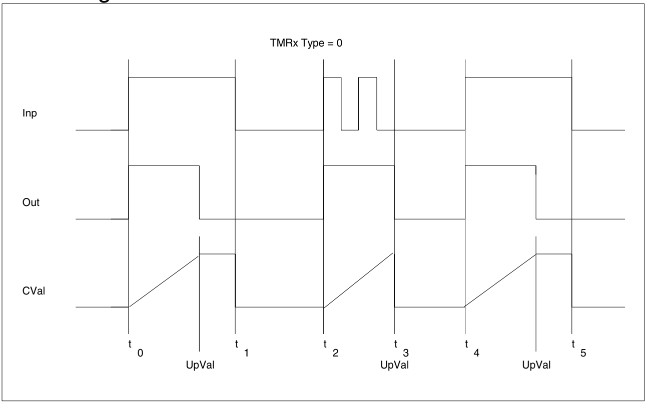
TMR Timer type 0.
By type selection TYPE = 0 the timer assumes the function of an edge recognition.

A positive edge (0 - 1 jump) at the input INP starts the timer. The output OUT is switched active (log. 1) and the internal counter starts counting up. When the current counter reading reaches the declared delay time UPVAL (CVal = UPVAL), the output gets inactive (log. 0).

Thus a positive edge at the input generates a single pulse of a pulse duration that is defined by UPVAL.

The resetting of the internal counter is dependent on the input pulse duration. The pulse diagram illustrates these processes.

Pulse diagram:



Example of a function :

TMRx BC0 I0.0 O0.0 WC100

Timer No 1 is configured for the function of edge recognition. A positive edge at input I0.0 creates a single pulse at output O0.0 of a pulse duration of 1000 ms (100x10=1000 ms).

Programming example:

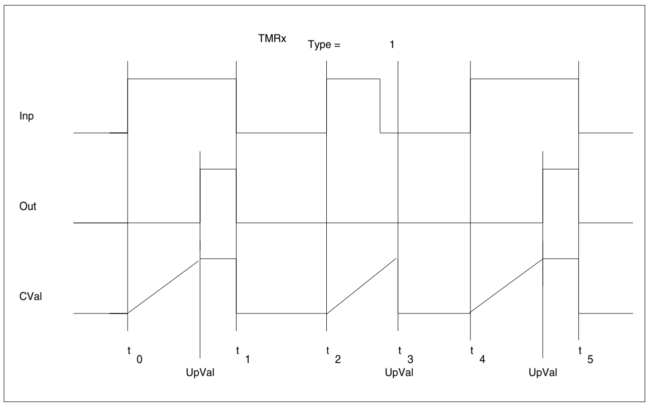
```
/P_TMR10.PGM
/Function: timer as edge recognition
/in a positive edge recognition a single
/output pulse of 1000ms is created

#define    type  bc0   /timer as edge recognition
#define    inp   i0.0  /signal input
#define    out   o0.0  /signal output
#define    upval wc100 /pulse duration = 1000ms (100x10)
/──────────program─────────────────
tmr1  type inp        /timer 1 as
ep    out upval       /edge recognition
```

3.8.2 On Delay

TMR Timer type 1.
By type selection TYPE = 1 the timer assumes the function of an on delay.
With a positive edge (0 -1 jump) at the input INP the on delay of the timer starts to work. The signal at the output OUT is switched active (log. 1) only when the internal counter (CVal) has reached the declared time value UPVAL (CVal = UpVal). In the further pulse process, the output signal follows the input signal.

The internal counter is reset when the input signal gets log. 0. The pulse diagram illustrates these processes.

Pulse diagram:



Example of a function:

```
TMR2 BC1 I0.1 O0.1 WC150
```

Timer No 2 is configured for the function of on delay. A signal pulse at input I0.1 creates a delayed switching pulse at output O0.1 with a delay of 1500 ms (150x10 = 1500 ms).

Programming example:

```

/P_TMR21.PGM
/Function: timer as on delay
/with a positive edge the output
/gets active with an on delay of 1500 ms

#define  type    bc1       /timer as on delay
#define  inp     i0.1      /signal input
#define  out     o0.1      /signal output
#define  upval   wc150     /pulse duration = 1500 ms (150x10)

/———————————program————————————————

tmr2 type inp     /timer 2 as
ep    out upval   /on delay

```

3.8.3 Off Delay

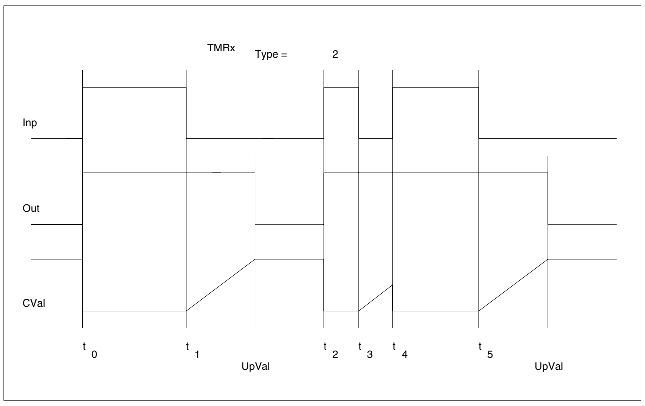
TMR Timer type 2.
By type selection TYPE = 2 the timer assumes the function of an off delay.

With a positive edge (1 - 0 jump) at the input INP the off delay of the timer starts to work.

First the signal at the output OUT follows the input signal. On the arrival of the negative edge at input INP the internal counter (CVal) is started and the delay time begins running until the internal counter (CVal) has reached the declared time value UPVAL (CVal = UPVAL). During this time the output signal is kept at log.1 and then turns to log. 0.

The internal counter is reset when the input signal gets log. 1. The pulse diagram illustrates these processes:

Pulse diagram:



Example of a function:

Timer No3 is configured for the function of an off delay. A signal pulse at input I0.2 creates a delayed off pulse at output O0.2 with a delay of 2000 ms (200x10 = 2000 ms).

Programming example:

```
/P_TMR32.PGM
/Function: timer as off delay
/with a positive edge the output
/gets active with an off delay of 2000 ms

#define type bc2  /timer as off delay
#define inp i0.2  /signal input
#define out o0.2  /signal output
#define upval  wc200   /pulse duration = 2000 ms(200x10)

/───────────program──────────────
tmr3 type inp  /timer3 as
ep    out upval   /off delay
```

3.8.4 Clock

CLOCK Clock Signal

Clock changes alternatively BitVar every time, the program pass through.

Usage:
    CLOCK   BitVar

Number of parameters: 1.

Parameters description:

"BitVar"
    A bit variable, that changes it's (from False - 0 to
    True -1 and vice versa) value every program pass.
    Type of Parameter: BIT
    Location For the Parameter: OUTPUT or MEMORY

CLOCK
    Usage is equivalent to the following BCON S program:
    LD   BitVar
    STN   BitVar
    Stack Depth for macro - 1 BYTE.

Example:

```
Main:
   CLOCK   Q0.5
   CLOCK   M3.5
   EP
```

CLOCK_N Pulse Output

Clock_N is an asymmetrical clock with defined pulse/pause width
In a period NOCLH (Number Of CLocks High) times the program pass through CLOCK_N the output BitVar remains High (True - 1), and
in a period NOCLL (Number Of CLocks Low) the output BitVar remains Low (False -0)

Usage:
    CLOCK_N NOCLH,NOCLL,CNTR,BitVar
    CLOCK_N NOCLH NOCLL CNTR BitVar

Number of parameters: 4.

Parameters description:

"NOCLH"
    Number of program passes high.
    Type of Parameter: WORD
    Location For the Parameter: OUTPUT or MEMORY or
    CONSTANT
"NOCLL"
    - Number of program passes low.
    Type of Parameter: WORD.
    Location For the Parameter: OUTPUT or MEMORY or
    CONSTANT

“CNTR”

- The location, where counting is done.
Type of Parameter: WORD.
Location For the Parameter: OUTPUT or MEMORY.

“OUT”

- The output of Clock_N. It stays True (1) NOClH times program passes and False (0) NOClL times program passes.
Type of Parameter: BIT.
Location For the Parameter: OUTPUT or MEMORY.

CLOCK_N Usage is equivalent to the following BCON S program:

```
        LD        CNTR
        SUB       WC1
        DUP        W
        ST        CNTR
        LE        WC0
        JMPCN     EOM
        LD        OUT
        DUP
        STN       OUT
        JMPCN     NO1
        LD        NOTH
        ST        CNTR
        JMP       EOM
NO1:    LD        NOTL
        ST        CNTR
EOM:
```

Stack Depth for macro - 4 BYTEs.

Example:

```
   Main:
         WORD CntrWrd1,CntrWrd2
         CLOCK_N    WM33, WM55,CntrWrd1,O7.3
         CLOCK_N    WC33,WC55,CntrWrd2,O7.3
         EP
```

3.9 Counters

3.9.1 COUNTER UP

CNTUP

The macro creates an upward counter which sets the counter output to log. 1 on reaching a predefined counter value. The value range of the upward counter applies to the signed positive 16-bit range, 0...+32767. In the function chart the relations are described diagrammatically.

The following invitation sequence is necessary:

CNTUP Clock,Reset,Count,UpValue,Output

Clock	counter input which is triggered with positive signal edges. maximum frequency permitted: 35 Hz data format: bit data type: input, output or marker
Reset	reset input which resets the counter (Count) with a positive signal edge data format: bit data type: input, output, marker, constant
Count	counter memory containing the current count value, after a positive count impulse, the counter contents are incremented by 1. The argument is available to the user and can be changed during the operation. Data format: word Data type: output, marker
UpValue	stop value which defines the count limit in counting up. On reaching this limit the output is set to log. 1. Data format: word Data type: input, marker, constant
Output	Counter output which is set to log. 1 on reaching the stop value. (Count>= UpValue) Data format: bit Data type: output, marker

Temporarily the macro needs 2 bytes for its operation. They are stored temporarily in the stack. The user has to provide for this required space in the stack before inviting the macro.
When there is no reset signal at the Reset input (reset = 0), the counter value (COUNT) is incremented by 1 with every positive edge (0 - 1 jump) at the counter input (CLOCK).

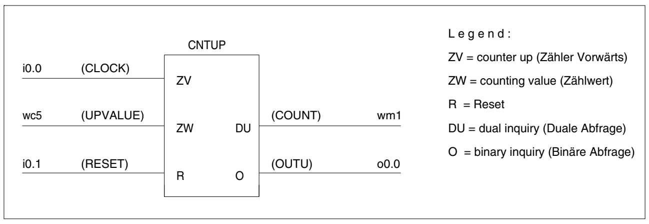
When the condition COUNT >= UPVALUE is fulfilled, the counter OUTPUT is set to log. 1.

After a repeated reset the counter returns to its normal position by COUNT and OUTPUT being reset. A new count operation can be started.

When more count impulses arrive at the counter input CLOCK after the stop value has been reached, the counting operation is not stopped. On passing the value range limit of +32767 there occurs an error situation which creates the following changes in the signal status:

- the counter (COUNT) is set to a negative value
- the counter output (OUTPUT) is reset to log. 0
- the status bit MO.7 is set to log. 1

Function chart:


CNTUP
i0.0 (CLOCK)
ZV
wc5 (UPVALUE)
ZW
DU
(COUNT) wm1
i0.1 (RESET)
R
O
(OUTU) o0.0
L e g e n d :
ZV = counter up (Zähler Vorwärts)
ZW = counting value (Zählwert)
R = Reset
DU = dual inquiry (Duale Abfrage)
O = binary inquiry (Binäre Abfrage)

Programming example:

```
/P_CNTUP.PGM
/counter up CNTUP
/CNTUP Clock,Reset,Count,UpValue,Output
/counting up from 0 to 5
/
#define    clock     i0.0/clock upward
#define    reset     i0.1        /reset
#define    count     wm1/count memory
#define    upvalue   wc5/stop value
#define    output    o0.0/control display

cntup clock,reset,count,upvalue,output
ld         m0.7      /(status bit reports error on
st         o0.7      /(exceeding value range of
ep                   /(>+32767
```

The counter up can also start from a defined start value which must be within the value range. For this purpose the argument COUNT must take the corresponding start value in an initial part before the macro is invited.

Example:

```
  #define    startwert    wc5    /assignment starting value 5
  #define    count        wm1    /assignment counter memory

  ld         startwert           /(initialization of the
  st         count               /(start value e.g. 5

mark1:
  cntup      clock, reset, count,upvalue,output
  jmp        mark1
  ep
```

To prevent counting after reaching the stop value, a logical ANDN conjunction of the counter input CLOCK with the counter output (OUTPUT) is advisable. Thus count impulses arrive at the counter input only as long as counter output is log. 0.

Example:

```
  #define     output      o0.0      /assignment output
  #define     clock1      i0.0      /assignment clock
  #define     clock       m10.0     /assignment clock latched

  ld          clock1                /clock
  andn        output                /negated output signal
  st          clock                 /clock latched
  cntup       clock,reset,count,upvalue,output
ep
```

3.9.2 Counter Down

CNTDN

The macro creates a counter down which sets the counter output to log. 1 on reaching a predefined counter value. The value range of the counter down applies to the signed positive 16-bit range, +32767...0. In the function chart the relations are described diagrammatically.

The following sequence is necessary:

CNTDN Clock,Load,Count,DnValue,Output

Clock	counter input which is triggered by positive signal edgespermitted maximum frequency: 35 Hz data format: bit data type: input, output or marker
Load	load input which takes over the DnValue (start - down value) in the counter (Count) with a positive signal edge data formnat: bit data type: input, output, marker, constant
Count	counter memory which contains the current counter value. After a positive count impulse at the counter input CLOCK, the counter contents are decremented by 1. The argument is available to the user and can be changed during the operation. data format: word data type: output, marker
DnValue	start value which defines the counting start for counting down data format: word data type: input, marker, constant
Output	counter output which is set to log. 1 on reaching the counter value limit of 0000h (Count<=0000h) data format: bit data type: output, marker

Temporarily the macro needs 2 bytes for its operation. The user has to provide for this required space in the stack before calling the macro.

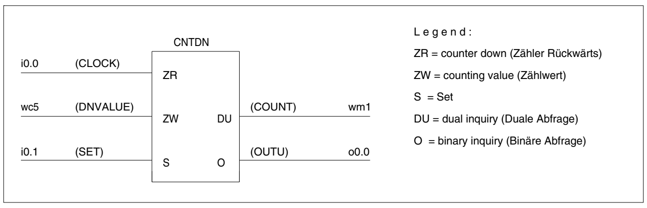
The start value for counting down is taken over by the counter (COUNT) with a positive edge (0-1 jump) at the load input (LOAD). With every positive edge (0-1 jump) at the counter input (CLOCK) the count value (COUNT) is decremented by 1.
When the condition COUNT <= 0000h is fulfilled, the counter output (OUTPUT) is set to log. 1.

After a repeated load impulse the counter is put back to its normal position by COUNT taking over the value of DNVALUE and OUTPUT being reset. A new counting operation can be started.

When more count impulses arrive at the counter input CLOCK after reaching the stop value, the counting operation is not stopped. There occurs an error situation which creates the following changes in the signal status:

- the counter (COUNT) is set to a negative value
- the counter output (OUTPUT) is reset to log. 0 when COUNT exceeds value - 32768

Function chart:


CNTDN
i0.0 (CLOCK)
ZR
wc5 (DNVALUE)
ZW
DU
(COUNT) wm1
i0.1 (SET)
S
O
(OUTU) o0.0
Legend:
ZR = counter down (Zähler Rückwärts)
ZW = counting value (Zählwert)
S = Set
DU = dual inquiry (Duale Abfrage)
O = binary inquiry (Binäre Abfrage)

Programming example:

```
/P_CNTDN:PGM
/macro - counter down CNTDN
/CNTDN Clock,Load,Count,DnValue,Output
/count down from 5 to 0

/
 #define     clock       i0.0      /clock down
 #define     load        i0.1      /take over DnValue
 #define     count       wm1       /counter memory
 #define     dnvalue     wc5       /start value
 #define     output      o0.0      /control display

cntdn clock,load,count,dnvalue,output
ep
```

In order to avoid counting on after reaching the stop value of 0000h, the logical ANDN conjunction of counter input Clock with counter output OUTPUT is advisable. Thus count impulses arrive at the counter input only as long as the counter output is log. 0.

Example:

```
 #define     output      o0.0      /assignment output
 #define     clock1      i0.0      /assignment clock
 #define     clock       m10.0     /assignment clock latched

 ld          clock1      /clock
 andn        output      /negated output signal
 st          clock       /clock latched
 cntup       clock,reset,count,upvalue,output
 ep
```

3.9.3 Reversive Up-Down Counter

CNTUD

The macro creates a combined up-down counter which has separate clock inputs and counter outputs for counting up and down. The value range of the combined counter applies to the signed positive 16-bit range, 0...+32767. In the function chart the relations are described diagrammatically.

The following sequence is necessary:

CNTUD ClockU,ClockD,Reset,Load,Count,LoValue,OutU,OutD

ClockU	counter input for counter up triggered by positive signal edges permitted maximum frequency: 35 Hz data format: bit data type: input, output or marker
ClockD	counter input for counter down triggered by positive signal edges permitted maximum frequency: 35 Hz data format: bit data type: input, output or marker
Reset	reset input which resets the counter (COUNT) through a positive signal edge data format: bit data type: input, output, marker, constant

Load	load input which takes over the LoValue (start-count down value) in the counter (COUNT) data format: bit data type: input, output, marker, constant
Count	counter memory containing the current counter value. After a positive impulse at the counter input ClockU, the contents of Count are incremented by 1. After a positive iimpulse at counter input ClockU, the contents of COUNT are incremented by 1. After a positive impulse at counter input ClockD, the contents of COUNT are decremented by 1. The argument is available to the user and can be changed during the operation. data format: word data type: output, marker
LoValue	start value which defines the start of counting down and simultaneously marks the stop value in counting up. When the up counter reaches this value, OutU is set to log. 1.
OutU	counter output for the counter up. On reaching the stop value it is set to log. 1. (Count > LoValue) data format: bit
OutD	counter output for the counter down. On reaching the stop value of 0000h it is set to log. 1. (Count <= 0000h) data format: bit data type: output, marker

Temporarily the macro needs 2 bytes for its operation. They can be intermediately stored in the stack. The user has to provide for this required space in the stack before calling the macro.

Macro operation:
The value of LoValue (start value for counting down and stop value for counting up) is taken over by the counter through a positive edge at the input LOAD (0 - 1 jump).

Counting up:
By a positive impulse on the input RESET, COUNT is set to 0000h. The stop value is predefined by LOVALUE. Counter impulses arriving at the input CLOCKU increment the counter (COUNT) by one step each.

When the condition COUNT > LOVALUE is fulfilled, the counter output OUTU is set to log. 1.

Counting down:
A positive impulse at input LOAD sets COUNT to the value of LOVALUE. Thereby the start value for counting down is set. Count impulses arriving at input CLOCKD decrement the counter (COUNT) by 1 step each.

When the condition COUNT <= LOVALUE is fulfilled, the counter output (OUTD) is set to log. 1

Combined operation:
When a counting operation, e.g. counting up, has been started, the counter (COUNT) is decremented by down clocks and vice versa. Predefined threshold values, such as 0000h or LOVALUE, are represented by the outputs OUTU and OUTD respectively. Reaching one of these threshold values does not inhibit the further counting of impulses.

With this combined counter one can choose freely between counting up and counting down.

Counter pulses that arrive after a stop value has been reached are not stopped. Dependent on the direction of counting there occurs an error situation which creates the following changes in the signal status:

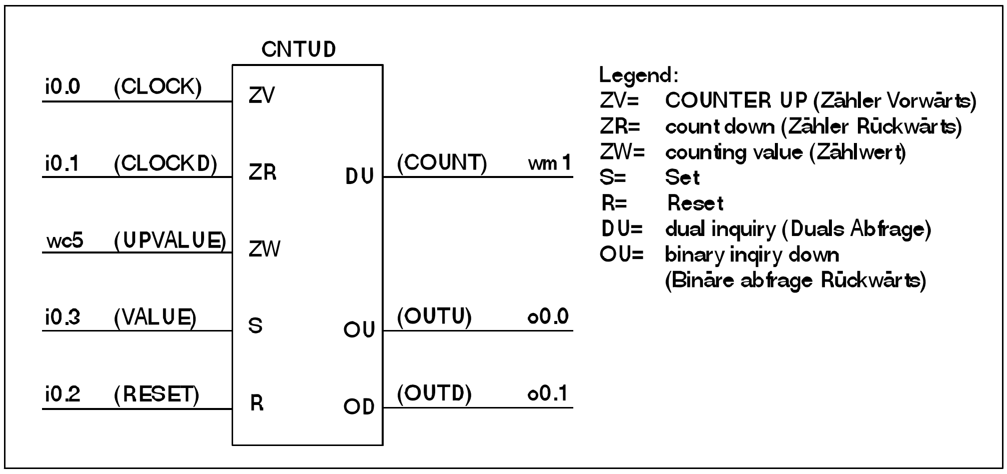
On exceeding the value range limit of +32767, the following changes in the signal status are created:

Counting up:
- the counter (COUNT) is set to a negative value
- the counter output (OUTU) is reset to log. 0
- status bit M0.7 is set to log. 1

Counting down:

- the counter (COUNT) is set to a negative value
- the counter output (OUTD) is set to log. 0 when COUNT exceeds the value of -32768

Function chart:



Programming example:

```
/P_CNTUD.PGM
/up - down counter CNTUD
/CNTUD ClockU,ClockD,Reset,Load,Count,LoValue,Outu,OutD
/counting up from 0 to 5
/counting down from 5 to 0
/
#define     clocku      i0.0      /clock up
#define     clockd      i0.1      /clock down
#define     reset       i0.2      /RESET
#define     load        i0.3      /loads LOVALUE
#define     count       wm1       /counter memory COUNT
#define     lovalue     wc5       /start value for DOWN
#define     outu        o0.0      /control display UP
#define     outd        o0.1      /control display DOWN

cntud       clocku,clockd,reset,load,count,lovalue,outu,outd
ld          m0.7                  /(status bit: reports error for
st          o0.7                  /(exceeding value range in
ep                                /(counting up at > +32767
```

In order to avoid counting after reaching the stop values of 0000h and LOVALUE respectively, the logical ANDN conjunction of counter inputs CLOCKU and CLOCKD with the corresponding counter outputs is advisable. Thus count pulses can only arrive at the corresponding counter input as long as the counter output belonging to it is log. 0. Cf. the examples listed under CNTUP and CNTDN.

3.10 Triggers

3.10.1 Reset Dominant Trigger

RTR

The macro creates an RS trigger (Reset - Set trigger) which resets the output dominant.

The following sequence is necessary:

RTR Reset,Set,Output

All arguments are only permitted in bit format.

Arguments	Data types
reset, set	inputs, outputs, markers, constants
output	outputs, markers

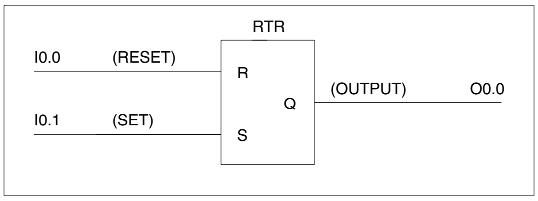
The RTR output is defined in the following table:

Reset	Set	Output
0	0	previous history
1	0	0
0	1	1
1	1	0

The trigger is called resetting dominant because its Reset input takes priority over its Set input.

During the macro operation 1 bit is temporarily filed in the stack. It follows that the user must provide for enough space for at least 1 byte in the stack before calling the macro.

Function chart:



Programming example:

```
/P_RTR.pgm
/RTR = RS trigger, reset dominant
/RTR Reset, Set, Output
/
#define     reset       i0.0      /Reset signal
#define     set         i0.1      /Set signal
#define     output      o0.0      /output as control display

rtr         reset, set, output
ep
```

3.10.2 SET RS DOMINANT TRIGGER

STR

The macro creates an RS trigger (Reset-Set trigger) which sets the output dominant.
The following invitation sequence is necessary:

STR Reset,Set,Output

All arguments are only permitted in bit format.

Arguments	Data types
reset, set	inputs, outputs, markers, constants
output	outputs, markers

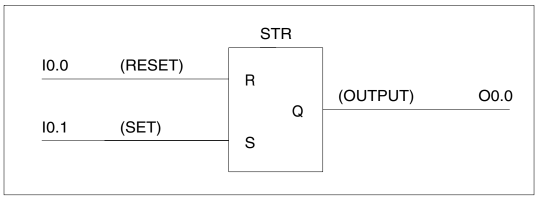
The STR output is defined in the following table:

Reset	Set	Output
0	0	previous history
1	0	0
0	1	1
1	1	0

The trigger is called setting dominant because its Set input takes priority over its Reset input.

During the macro operation 1 bit is temporarily filed in the stack. It follows that the user has to provide for enough space for at least 1 byte before calling the macro.

Function chart:



Programming example:

```
/P_STR.PGM
/STR = RS trigger, set dominant
/STR Reset,Set,Output
/
#define     reset     i0.0      /reset signal
#define     set       i0.1      /set signal
#define     output    o0.0      /output as control display

str reset,set,output
ep
```

3.10.3 Positive Edge Recognition

UPLS
The macro recognizes a positive signal edge arriving at the signal input BITINP (rising edge or 0 - 1 jump) and thereupon creates a single pulse.

The following sequence is necessary:

BitInp,BitOut

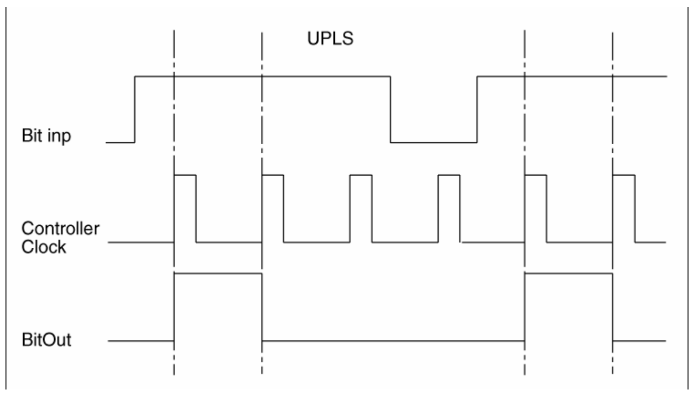
BitInp signal input which creates through a positive signal edge (0 1 jump) a single pulse at the signal output (BitOut).
permitted maximum frequency: 35 Hz
data format: bit
data type: input, output, marker

BitOut signal output which creates a single pulse depending on the signal input (BitInp) and the system clock of the control.
data format: bit
data type: output, marker

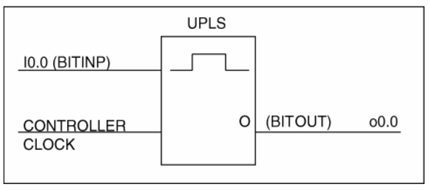
Temporarily the macro needs 2 bytes for its operation. They can be intermediately stored in the stack. The user has to provide for this required space before calling the macro.

When a positive signal edge (0-1 jump) is recognized at the signal input (BITINP), the macro creates a single pulse with the next system clock of the control. The duration of the pulse is defined by the duration of the system clock period. The pulse is available at the signal output (BITOUT).

Pulse clock diagram:



Function chart:



Programming example:

```
   /P_UPLS.PGM
   /positive edge recognition
   /UPLS BitInp,BitOut

   #define  bitinp   i0.0   /signal input
   #define  bitout   o0.0   /control display impulse

   upls     bitinp,bitout
   ep
```

3.10.4 Negative Edge Recognition

DNPLS
The macro recognizes a negative signal edge (falling edge or 1-0 jump) arriving at the signal input BITINP and thereupon creates a single pulse.

The following sequence is necessary:

DNPLS BitInp,BitOut

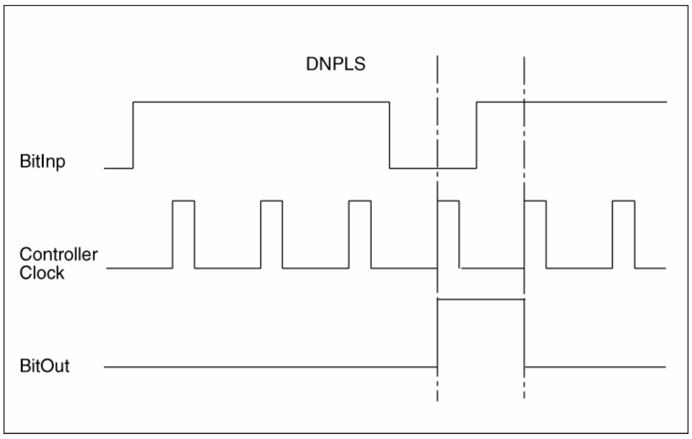
BitInp signal input which creates a single pulse at the signal output(BitOut) through a negative signal edge (or 1-0 jump respectively). Permitted maximum frequency: 35 hz
data format: bit
data type: input, output or marker

BitOut signal output which creates a defined pulse depending on the signal input (BitInp) and the system clock of the control.
data format: bit
data type: output, marker

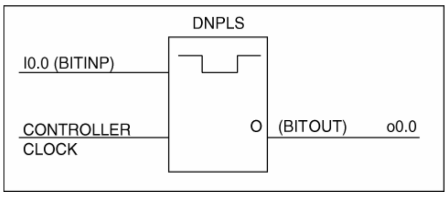
Temporarily the macro needs 1 byte for its operation. It is intermediately stored in the stack. The user has to provide for this required space before calling the macro.

When a negative signal edge (1-0 jump) is recognized at the signal input (BitInp), the macro creates a single pulse with the next system clock of the control. The duration of the pulse is defined by the duration of the system clock. The pulse is available at the signal output (BITOUT).

Pulse clock diagram:



Function chart:



Programming example:

```
/P_DNPLS.PGM
/negative edge recognition
/DNPLS BitInp,BitOut

#define  bitinp   i0.0     /signal input
#define  bitout   o0.0     /control display impulse

dnpls    bitinp,bitout
ep
```

Index