

PD Calculator Assembler

Version 3.00

Manual

© Copyright 1997 by PROCES-DATA A/S. All rights reserved.

PROCES-DATA A/S reserves the right to make any changes without prior notice.

P-NET, **Soft-Wiring** and **Process-Pascal** are registered trademarks of PROCES-DATA A/S.

Contents

	Page
1	Introduction to Calculator Assembler. 1
2	User Interface. 2
2.1	Editing a file 3
2.2	Assembling a program 3
2.3	Download a program 4
2.4	Debug a program 4
2.5	Online help. 5
3	Calculator programming. 6
3.1	Calculator registers. 6
3.2	Calculator channel. 8
3.3	Assembler program. 8
3.4	Number constants. 9
3.5	Identifiers. 11
3.6	Label. 12
3.7	Addressing modes. 12
3.8	Variables. 13
3.9	Instructions. 14
	Appendix A. Syntax diagrams. 20

1 Introduction to Calculator Assembler.

The calculator assembler makes it possible, on a PC, to edit, assemble and download programs to calculator channels in P-NET modules. The Calculator Assembler is an integrated program with an editor, an assembler, a debugger and a loader for P-NET. Calculator programs are written as assembler instructions in text files. By using the editor the source text is edited and saved. By using the assembler the source text is assembled to generate the calculator instructions. These instructions are downloaded to the Calculator channel, which then is started. It is possible to debug the downloaded program by single steps or a break point.

An example of a calculator program with some typical instructions.

```

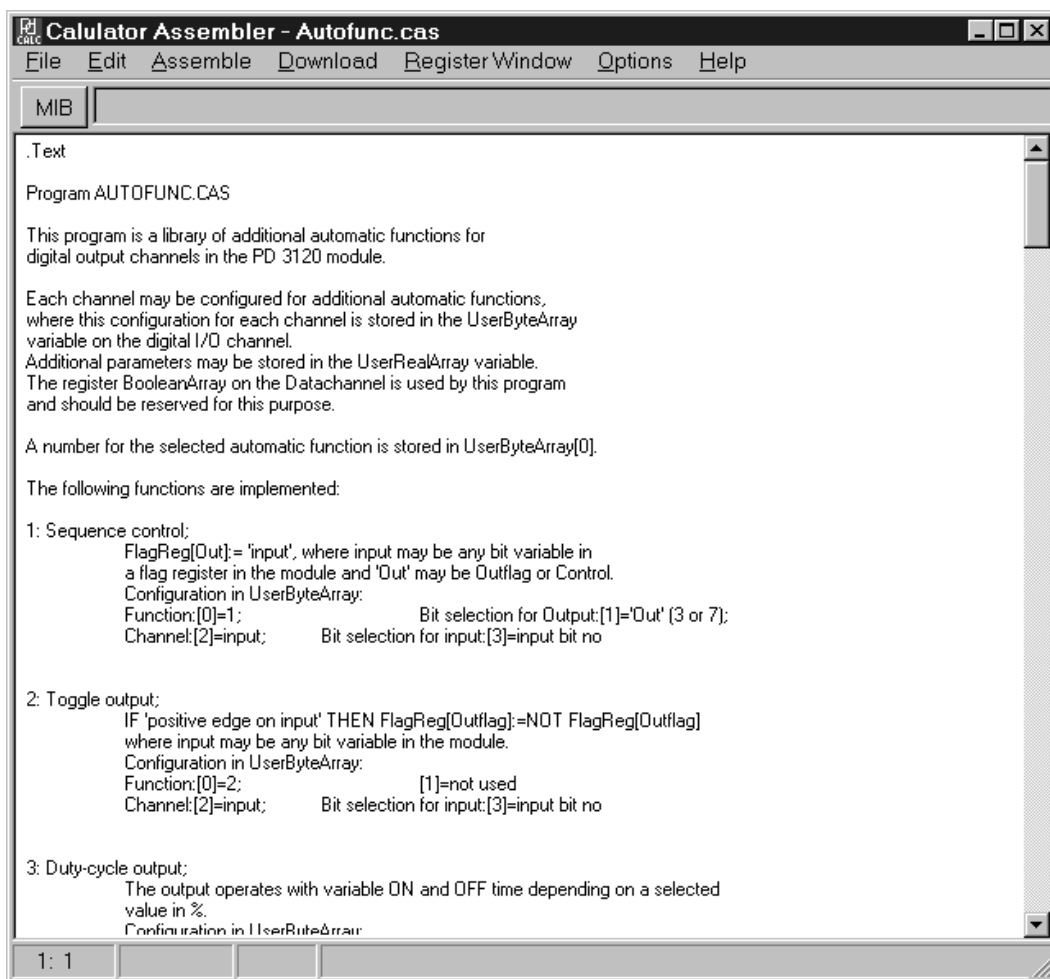
                                Move #D,CR1                ; Let CR1 point out the pulse processor
                                                                ; channel in a PD 3221
Start:                          Move #0,IR1                ; First element
Loop:                            Move CR1:#A[IR1], Acc      ;Load pulse processor registers[elementNo]
                                Add 100
                                Move Acc, CR1:#A[IR1]     ; Store back incremented value
                                Inc IR1                     ; Next element
                                Move IR1, Acc               ; Load IR1 in to Acc
                                Comp Acc > 15              ; Is last element treated?
                                Jump.False Loop             ; No: then repeat loop
Finish:                          .                          ; Yes: then ...
                                .
                                .
                                Jump Start                  ; Last instruction must be an unconditional
                                                                ; jump to a label
End                               ; End of program

```

2 User Interface.

The user interface is based on the Common User Access (CUA) standard normally used by window programs. Instructions of how to carry out the basic manipulation of programs, windows, dialogues and menus can be found in the Microsoft Windows User's Guide.

The program's main window contains a menu bar at the top and a status line at the bottom. A file can be opened for editing and assembling. Furthermore the program has a MIB button used to select the destination module for download and debugging.



In the following paragraphs, the different parts of the program are covered in detail separately.

2.1 Editing a file

The editor is used to edit the calculator programs. Files can be opened by the 'File | Open...' command in the menu. Files are also saved and printed from the File menu. Calculator assembler source files have CAS as default extension.

A new (blank) file window corresponding to new a assembler file can be created with the 'File | New' command.

The standard CUA edit commands (listed in the Edit menu) for copy, cut and paste of a selected text are available in the editor. Text can be exchanged to and from the windows system's clip board. The editor can undo the last cut command. Marked text will be replaced when new text is entered. The search and replace operations are listed in the Search Menu.

In the status line the cursor's position is shown. It is also indicated if the file has been modified since the last time it was saved.

Note, that when the register window (used when debugging) is shown, the contents of the edit window is locked. Close the register window, make the changes, assemble the program and download it again. Then reopen the register window to see the effect of the changes.

2.2 Assembling a program

The assembling of a source file is started from the Assemble Menu. A status dialogue displays the line number and size of the generated code during the assembling.

When an error is found, a dialogue will pop up to inform the user about the error. After pressing the OK button in the status dialogue, the assembling process continues. The assembling process can be interrupted by pressing the cancel button in the status dialogue.

During the assembling process a debug (*.deb) file is generated. It contains a list of line numbers, instruction addresses and label names. The list is used for debugging.

After a successful assembling (No errors found), it is possible to download the generated code to a calculator channel. The download command in the main menu do that.

The generated code can, by the Assemble menu, be saved to files. The 'Write to INC file' command in the 'Assemble' menu, will create a include file for Process-Pascal. The 'Save to CXE file' command will save to a file on disk.

- *.inc The generated calculator instructions as Process-Pascal source text. May be included in a Process Pascal program (ASCII).
- *.cxe The generated calculator instructions in a binary format.

2.3 Download a program

For downloading of calculator programs, a calculator channel must be selected as destination. A calculator channel is a channel with a physical ID ending with '.CALCULATOR'.

To select a calculator channel activate the MIB button at the top left of the screen. This changes the edit window into a MIB viewer. Use the mouse to expand the MIB structure to find the desired calculator channel. When the calculator channel is highlighted click on the MIB button again or double click the channel. This will close the MIB viewer and the edit window reappears. The selected channel is now shown to the right of the MIB button.

When the desired destination channel is shown, select the Download menu. This will start the standard P-NET downloader application. For further information on the download program, refer to the documentation for the downloader.

It is also possible to start and stop the program with the download application.

2.4 Debug a program

The Calculator Assembler supports interactive debugging of a calculator program downloaded in a calculator channel. The debugger makes it possible to single step through the calculator instructions or to set a break point.

After an assembling of a source text, the generated codes must be downloaded to a calculator channel, and the module must be reset to reinitialize the calculator. The debugger is started by selecting Register Window in the main menu. When debugging starts, the register window will show the calculator's registers. It is not possible to edit the program source when the register window is shown.

The user can single step through the program, by pressing a key (F7). To set a break point the user places the cursor at the line containing the instruction to stop at, and press the F4 key. The calculator then starts, and it runs until the break point is reached. When the execution stops the calculator's internal registers can be inspected in the register window. The register values can also be changed. To restart the calculator program from the first instruction, the F2 key can be pressed. The calculator can also be started and stopped with 'Start Calculator' and 'Stop Calculator' in the P-NET menu.

In the register window is also three buttons, which acts as fast shortcuts to the Debug Step, Debug Goto cursor and Debug Reset commands. The Debug Reset command stops the calculator if it is running, and resets the calculator's instruction pointer to the first program instruction. When the debugger is reset, has been single stepped or has reached a breakpoint, the line containing the next instruction to be executed, is highlighted as marked text.

2.5 Online help.








Through the help menu is online help available. Much of the text in this manual is also part of the help text.

3 Calculator programming.

3.1 Calculator registers.

The calculator has a set of internal registers which are used when the calculator programs runs. These registers are not available through the SoftWire numbers in the calculator channel.

The calculator's internal registers are:

Name	Mnemonic	Type
Accumulator:	 Acc	: real
Bit accumulator:	 BitAcc	: bit
Channel registers:	 CR1	: byte
	 CR2	: byte
Index registers:	 IR1	: byte
	 IR2	: byte
Bit Index register:	 BIR	: byte

The accumulator (Acc) is used both as operand and to hold the result for arithmetic operations. E.g. 'Add 5' means $Acc := Acc + 5$, and 'Div 7' means $Acc := Acc/7$. The accumulator is a real and all variables moved into it are automatically converted into a real. When the accumulator value is moved to a variable of the type byte, integer, word or longint, it is automatically converted into the new type. The operations on the accumulator are Move, Add, Sub, Div, Mul, Comp, LookUp.

The bit accumulator (BitAcc) is used as both an operand and to hold the result for logical operations. E.g. 'Not BitAcc' means $BitAcc := Not\ BitAcc$, and 'And CR1:7' means $BitAcc := BitAcc\ and\ variable\ addressed\ by\ CR1:7$. The operations on the bit accumulator are Move, Not, And, Or, XOr, Set, Clr, Comp.

Channel registers are used to hold the number of a channel in the module. The channel number are used in instructions together with a register number to make the address of a variable (that is a SoftWire number). Normally, a channel register is loaded with a channel number and then used in the following instructions to access registers in that channel.

Example:

```

Move    #C, CR1          ; Let CR1 point out calculator channel
Move    CR1:0, Acc       ; Load universal A. Variable at SwNo C0
Add     CR1:1            ; Add universal B. Variable at SwNo C1
Move    Acc, CR1:2       ; Store to universal C. Variable at SwNo C2

```

Index registers are used to hold an index (or offset) to address into a complex variable at a SoftWire number. Normally a index register is loaded with a value and then used to access a complex variable.

Example:

```

Move    #D, CR1          ; Let CR1 point out pulse processor channel
Move    #0, IR1          ; First element
Loop:   Move    CR1:#A[IR1], Acc; Load pulse processor registers[elementNo]
Add     100
Move    Acc, CR1:#A[IR1]; Store back incremented value
Inc     IR1              ; Next element
Move    IR1, Acc         ; Load IR1 into Acc
Comp    Acc > 15        ; Is last element treated?
Jump.False Loop         ; No: then repeat loop
Finish: .                ; Yes: then ...
.
.

```

The bit index register (BIR) is used to address a separate bit in a byte or word variable. In contrast to channel registers and index registers the bit index register is not necessary, but is an alternative to an immediate value. The following instruction

```

Move    CR1:7 :5, BitAcc ; Load bit 5 into BitAcc

```

has the same result as:

```

Move    5, BIR           ; Load BIR with 5
Move    CR1:7 :BIR, BitAcc ; Load bit 5 into BitAcc

```

The operations on the channel registers, the index registers and the bit index register, are Move, Inc, Dec.

3.2 Calculator channel.

Variables in the Calculator Channel varies depending on the actual type of P-NET module, please refer to the documentation for the module in question for further information.

3.3 Assembler program.

In the following sections the syntax for assembler programs is described and formally defined by syntax diagrams. All syntax diagrams are also listed together in appendix B. In the syntax diagrams a round box is a reserved word or special symbol, and a square box is another syntax diagram. A carriage return (end of line) is written as CR in the syntax diagram.

The source text for a calculator program have one instruction per line. The maximal line length is 128 characters.

A Calculator program consists of a sequence of Define statements or instructions; one instruction per line.

To indicate end of program an 'END' keyword must be placed on a line after the last instruction.

The last instruction in the program must be a jump Instruction. This means that the calculator will some how loop forever when it is started once. It is legal to let the last instruction jump to it self:

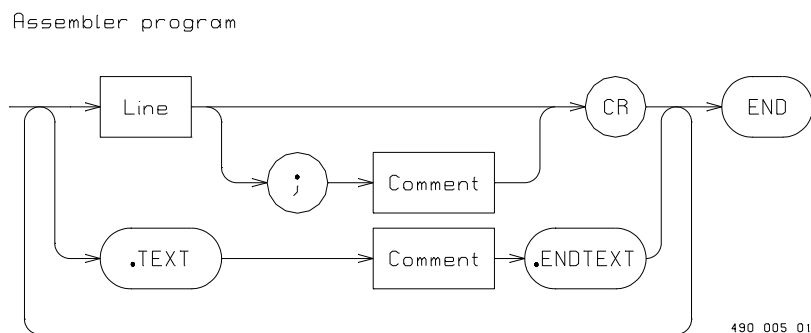
```

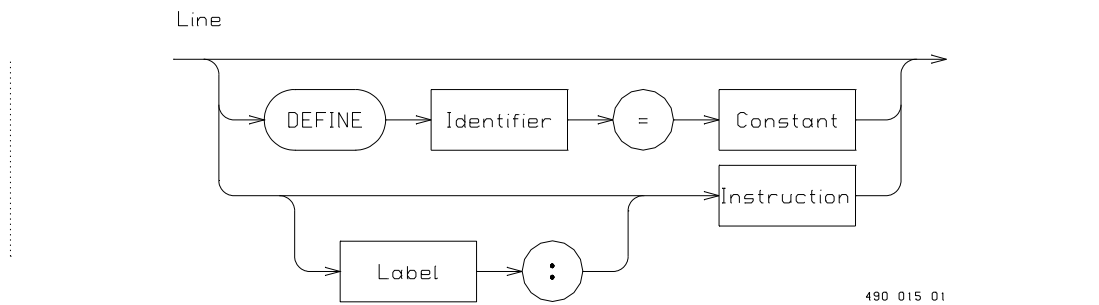
Forever:   Jump Forever   ; The Calculator does nothing
END                ; End of Program

```

It is also possible to clear the RunEnable bit in the calculator channel.

Calculator Assembler is not case sensitive, so both uppercase and lowercase letters can be used for identifiers and reserved words.





Two kinds of comments are available:

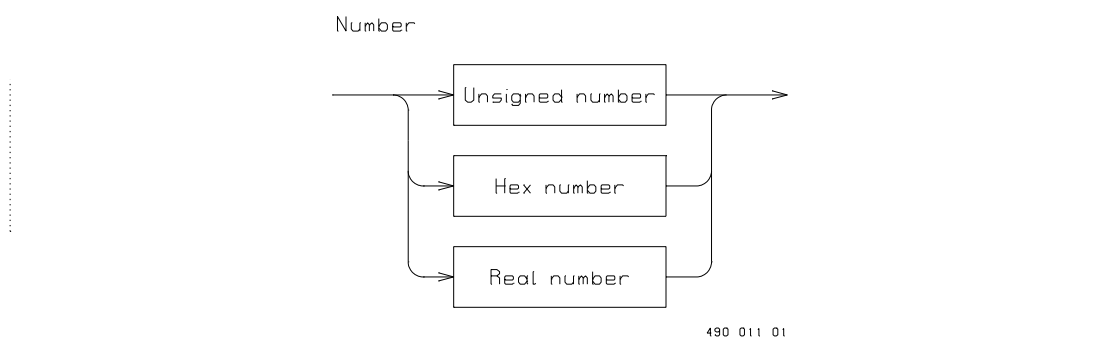
A ';' indicates that the rest of the line is a comment. This is useful e.g. when comments is given for a separate instruction or a single line comment.

A '.Text' indicate that a comment is given until the next occurrence of '.EndText'. This is useful when making larger comments covering several lines.

3.4 Number constants.

Constants are used as immediate operands in arithmetic instructions and in move instructions.

Number can be either decimal or hexadecimal. Hexadecimal numbers starts with a '#'. Real numbers is written as in Process-Pascal.



The value range for the different number types is also as in Process-Pascal, as given in the following table:

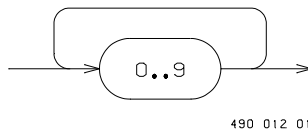
Type	Value ranges	Size
Byte	: 0..255	1 byte
Integer	: -32768..+32767	2 bytes
Longint	: -2147483647..+2147483647	4 bytes
Real	: +/-1e-38..+/-1e38	4 bytes

False and True can be used as names for 0 and 1.

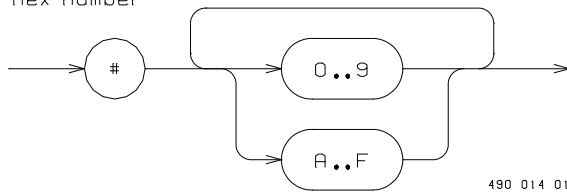
Example:

215 ; Byte
 -2017 ; Integer
 3.1415 ; Real
 -27.9e6 ; Real
 #1F ; Byte
 #1ACF ; Integer

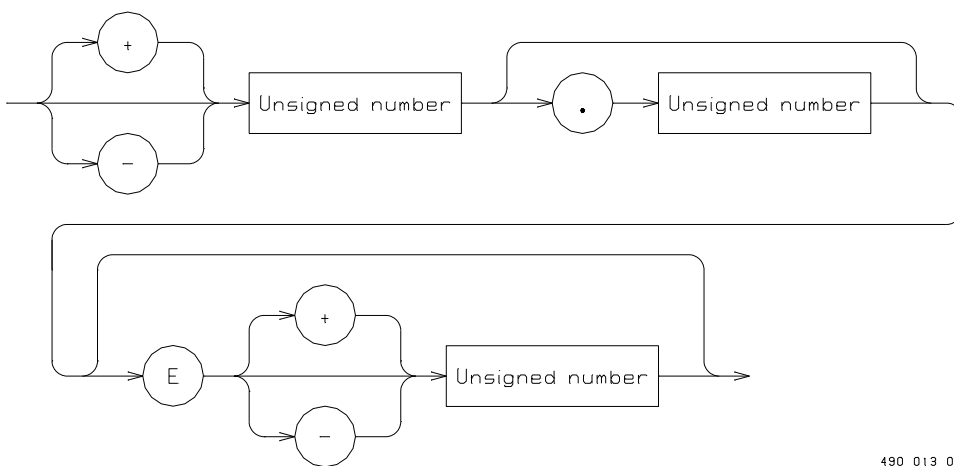
Unsigned number



Hex number



Real number



3.5 Identifiers.

Identifiers denotes labels and constants. Identifiers are strings starting with a letter, and eventually followed by some letters or figures.

Example:

Start, Counter, Loop1, L2

Identifiers of any length are allowed only the first 80 characters will however be used to separate identifiers.

Identifiers must be unique and different from any reserved word.

Reserved words

Acc, Add, And, AutoSave, BitAcc, BIR, Clr, Comp, CR1, CR2, Dec, Define, Div, End, False, Inc, IR1, IR2, Jump, Lookup, Move, Mul, Nop, Not, Or, Protect, Set, SetError, Sub, True, XOr.

Special symbols

+, -, :, [,], <>, <=, >=, <, =, >, ', ', !, !, #.

Carriage return (end of line) is also treated as a special symbol; written as CR.

Define statement

Define statement are used to associate a name with a constant value or another name. If another name is used it must be defined in a previous define statement. It is not possible to redefine a name to be associated with a new value. False and true are predefined names for 0 and 1.

Example:

```
Define PI = 3.1415
Define MaxValue = 27.967
Define Ok = true
```

3.6 Label.

Lines can be marked with an identifier, for symbolical reference to the instruction at the line. Labels are declared at the start of the line as an identifier followed by a colon. Labels are used in jump instruction as reference to the jump destination. In jump instructions only the identifier, and not the colon is written.

Example:

```

LoopStart: .           ; Instructions to be repeated
           .
           .
           Jump LoopStart

```

Only one label can exist with a given identifier.

3.7 Addressing modes.

The following addressing modes are available:

Immediate values

Use of a number constant or a named constant identifier in instructions, will code the value of the number into each instruction.

The same value or named constant can be coded into different types, depending on the instruction it is used in.

Calculator Registers

Temporary values and results of operations are stored in calculator registers. When registers are operands in an instruction, the register is coded into the instruction.

Variables

The values to be accessed over the P-NET are placed in the variables in the channels in the modules.

3.8 Variables.

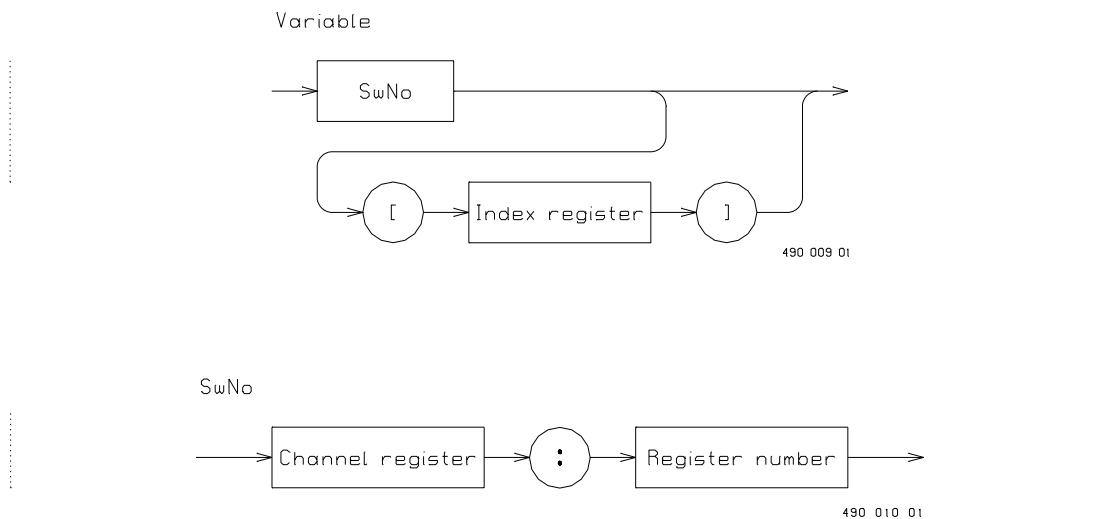
The variables are the different variables in the channels (SoftWire numbers) of the P-NET module, and these variables are predefined and of fixed type. It is not possible to declare variable storage as part of a calculator program.

Simple variable

A variable address (SoftWire number) is formed of a channel register and a register number separated with a colon. E.g. CR1 : 7. The channel register contain the number of the channel to address.

Complex variable

For complex variables an index register is used to address into the variable. The index is initially stored in an index register, either IR1 or IR2. The index register is then specified in a pair of squared parentheses, e.g. CR1:7 [IR1]



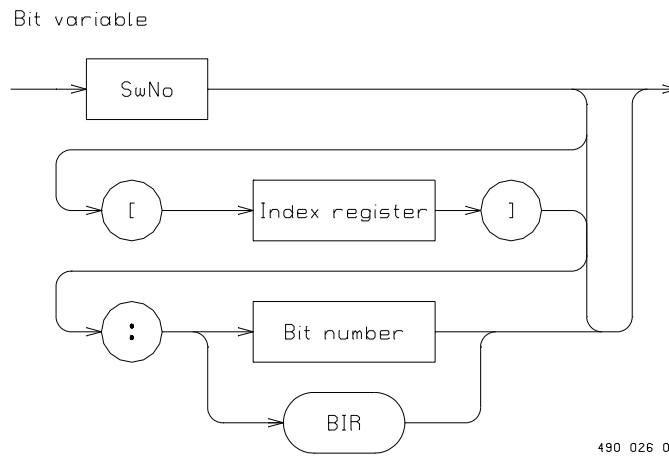
Bit variable

To address a single bit, a bit number can be specified after the SoftWire number. The SoftWire number and the bit number must be separated with a colon. E.G. CR1:7 :15. Instead of the bit number, a bit index register (BIR) can be used to hold the bit number. E.G. CR1:7 :BIR. Only one bit index register is available.

Complex bit variable

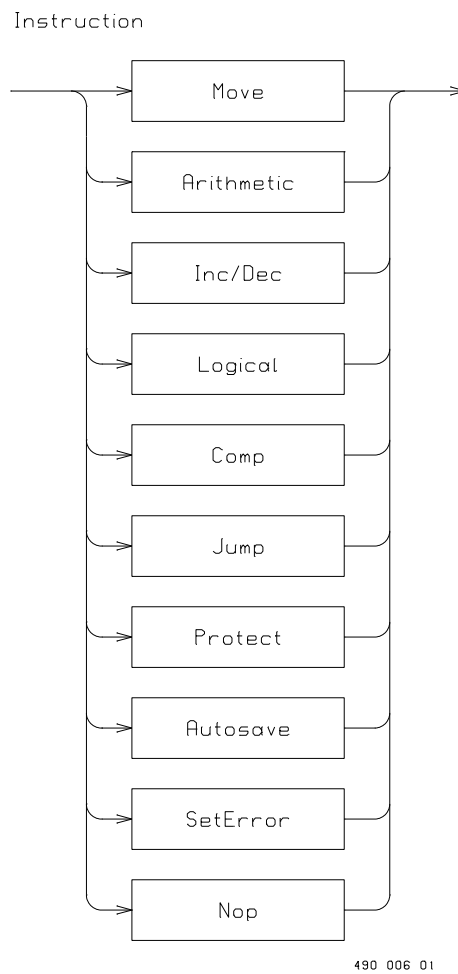
It is also possible to specify both an index and a bit index in the same instruction.

Example: CR1 : 7 [IR1] : 15



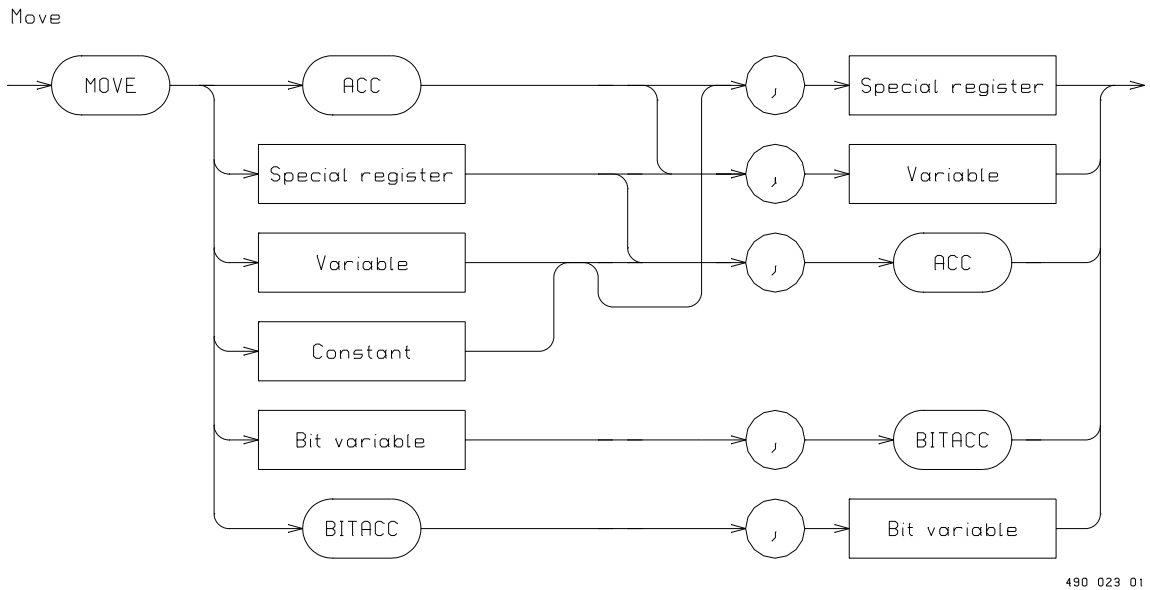
3.9 Instructions.

The available assembler instructions are listed in the following syntax diagram.



Move instruction

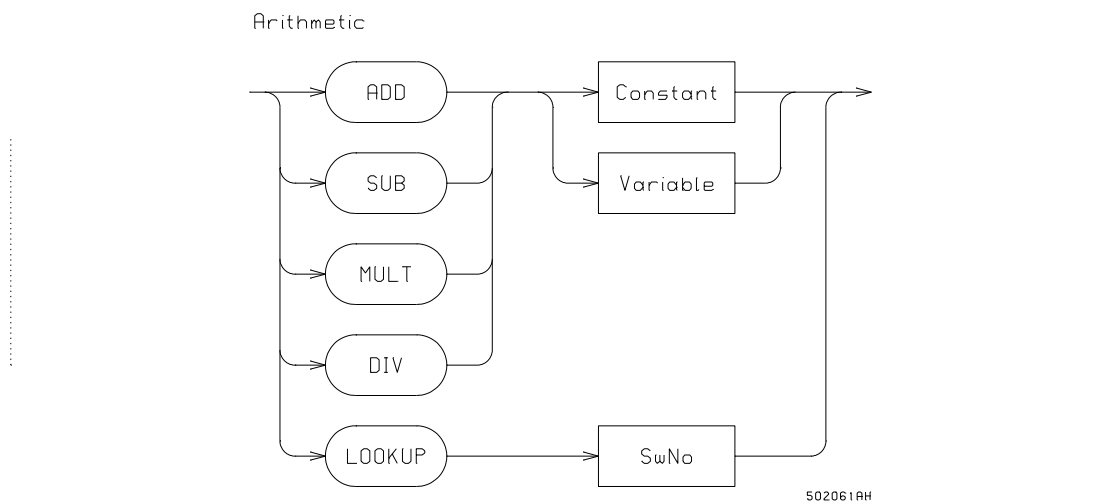
The move instruction has the form: Move <Source>, <Destination>. The move instruction moves a copy of the source to the destination. The possible type of operands are the accumulator, the bit accumulator, channel registers, index registers, bit index registers, variable addresses and constants.



Arithmetic instructions

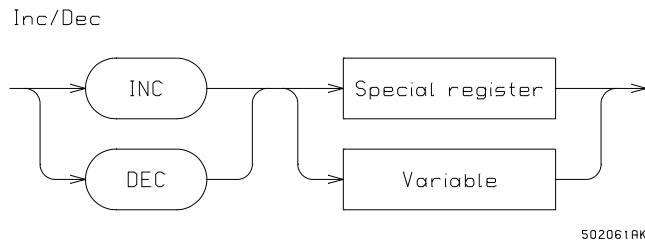
The arithmetic instructions are Add, Sub, Mul, Div and lookUp. These instructions use the accumulator as implicit operand. Add, Sub, Mul and Div also have a single explicit parameter as second operand. The second operand is either a variable or a constant. The result of the operation is placed in the accumulator.

LookUp has a variable of type lookup table as explicit parameter. The accumulator is used as input into the table and the resulting lookup value is returned into the accumulator.



Inc/Dec instructions

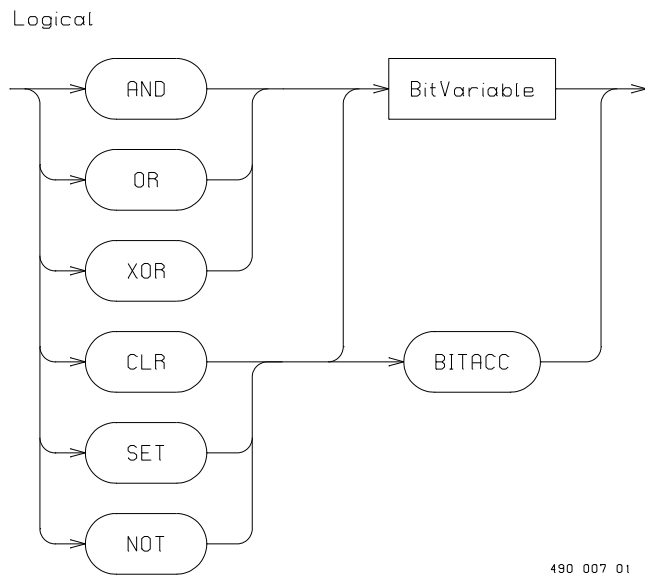
Increment/Decrement instructions add 1/subtracts 1 to a special register or a variable of type byte, integer and longint, but do not work on the accumulator or variables of type real. As the assembler is not able to determinate the type of the addressed variable, no error is shown during the assembling. If the variable is of type real it is not incremented/decremented.



Logical instructions

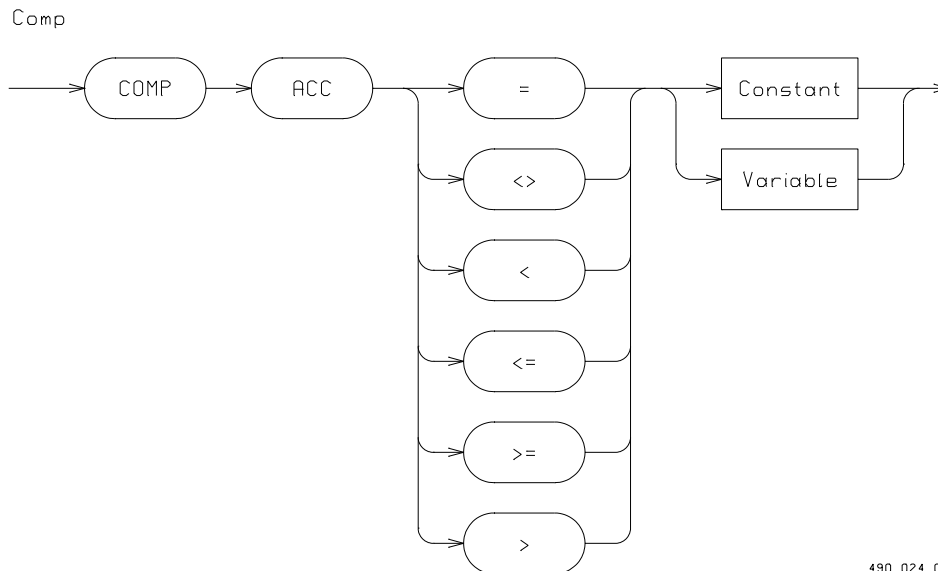
The logical instructions are Not, And, Or, Xor, Set, Clr. The And, Or and Xor instructions use the bit accumulator as implicit operand and a variable (of type boolean) as explicit operand/parameter. The result is stored in the BitAcc. The Set, Clr and Not instructions use one explicit operand, either the bit accumulator or a variable (of type boolean).

Example: AND CR1:#7 means that the variable at CR1:#7 is and'ed with the BitAcc. The result is stored in the BitAcc.



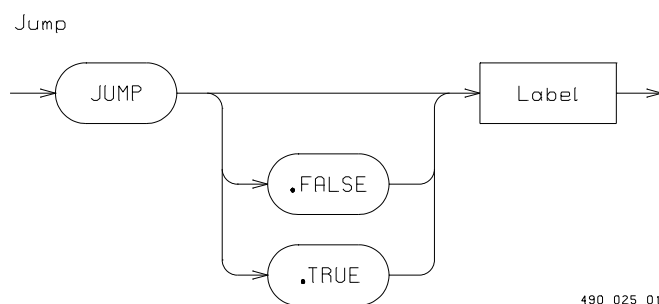
Compare instruction

The Comp instruction compares the accumulator with either a constant or a variable, and sets the bit accumulator according to the compare relation. If the relation is true, the bit accumulator is set to 1 otherwise to 0.



Jump instructions

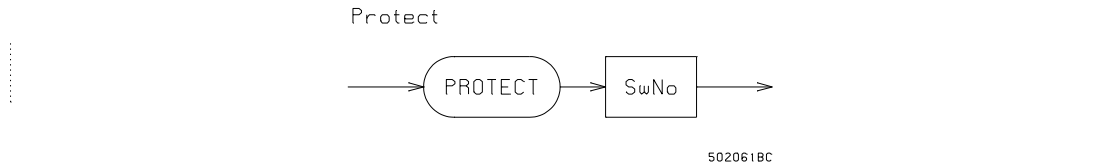
The Jump instructions can, eventually conditionally, jump to a labelled instruction. The Jump instruction make an unconditionally jump to the label. Jump.True makes a jump only if the BitAcc is true, and Jump.False jumps only if the BitAcc is false. The specified label must exist.



Protect

The Protect instruction protects a variable (SoftWire number) against overwriting from outside the module. Reading of the variable is still possible. Protection of a variable is removed by protecting another variable. By protecting a ReadOnly variable such as the NumberOfSWNo in channel 0, no variables are protected.

The protect instruction assures that a controller attempting to write a value to the protected variable, receives a 'busy' reply over the P-NET. When the variable later is unprotected, the controller is allowed to write its own value to the variable.



A typical application could be the following:

```

Move      #0,CR2      ; Channel 0
Move      #D,CR1      ; Calculator channel
Protect   CR1:5        ; Start protection
Move      CR1:5, Acc   ; Load value
Add       123          ; Update
Move      Acc, CR1:5   ; Store value
Protect   CR2:0        ; No protection of CR1:5

```

AutoSave

The AutoSave instruction saves all variables with memory type RAM AutoSave to EEPROM. This can be used to save configuration parameters or data, so that it will have these values after a reset or power of.

N.B. remember that writing to EEPROM is limited to a certain number of cycles.

Example:

```

Move      #C, CR1      ; Work on calculator channel
Move      1, Acc
Move      Acc, CR1:0   ; Set UniversalA (memory type RAM AutoSave)
                          ; Value is still just stored in RAM
AutoSave                          ; Store to EEPROM
                          ; Value is now stored in EEPROM

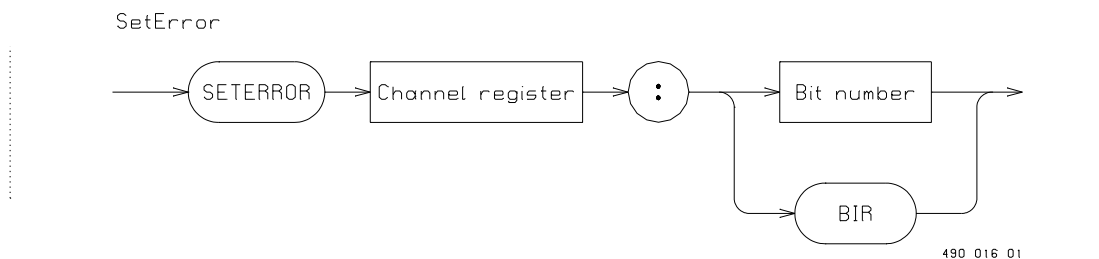
```

SetError

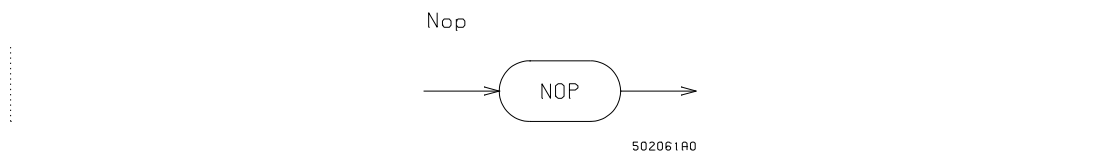
The SetError sets a specified bit in the calculator channel's ChError register. This is also reflected in Channel 0's ChError. A channel register containing the calculator channel's number, e.g. #0C for the PD 3221 UPI, must be specified. Both the Act and His bits are set.

Example:

```
SetError    #C, #7;
```

**Nop**

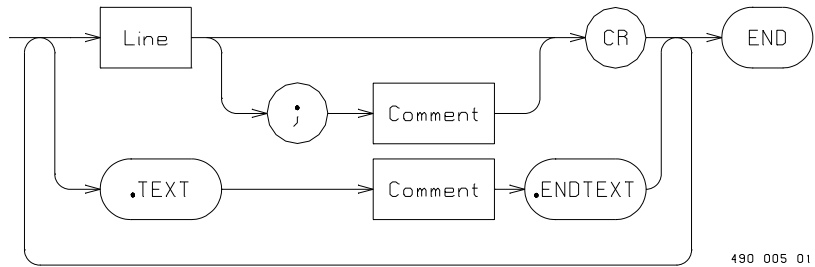
No operation.

**Instruction execution times**

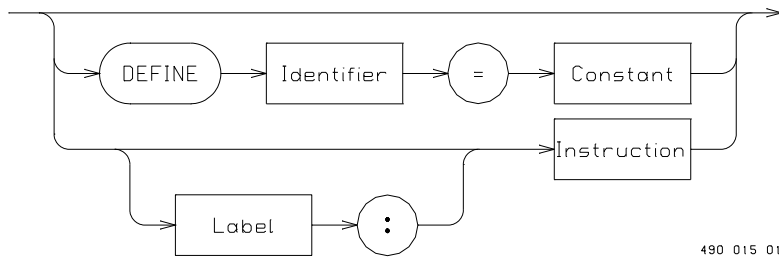
The execution times for the different instructions vary with the total load of the module, because the calculator has a lower priority than the module's other tasks. The execution time also vary for different module types. The execution time for a specific modules is given in the module's manual.

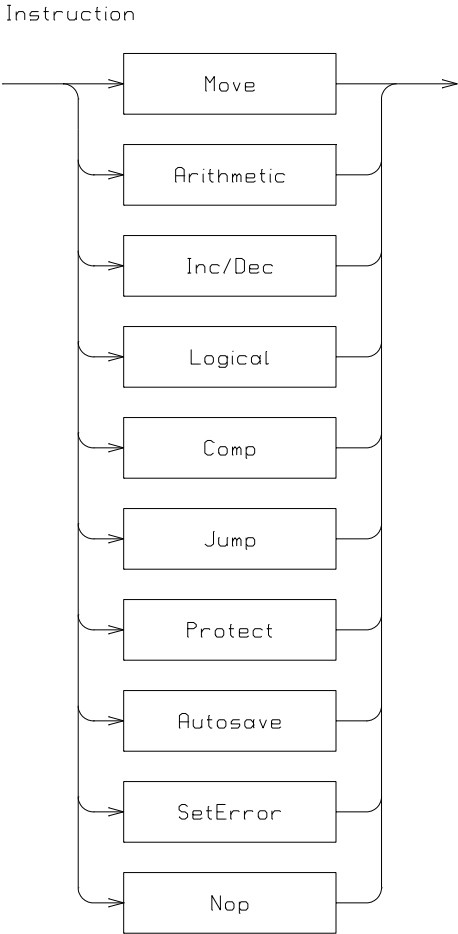
Appendix A. Syntax diagrams.

Assembler program

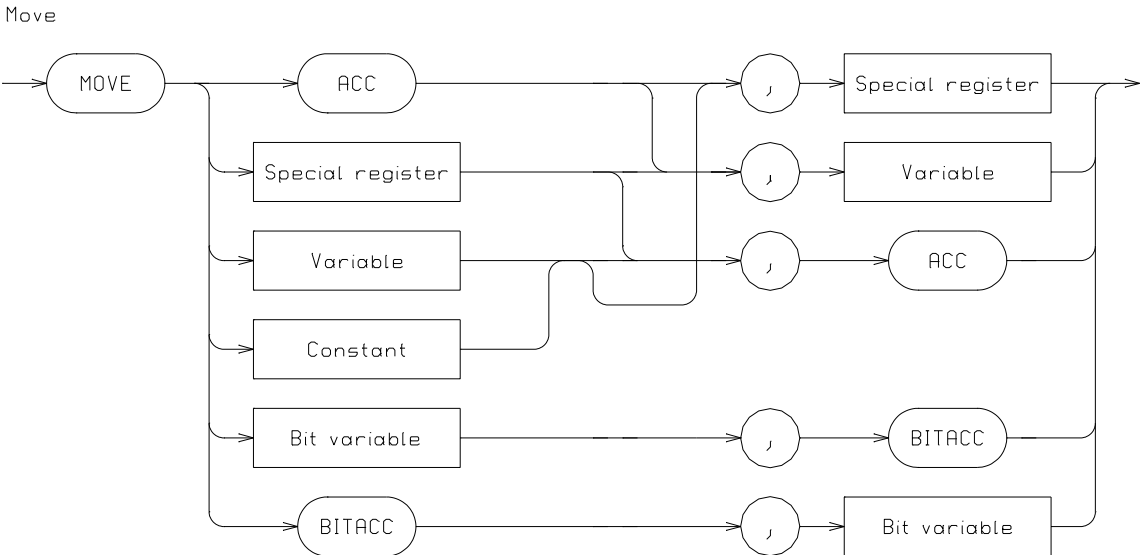


Line



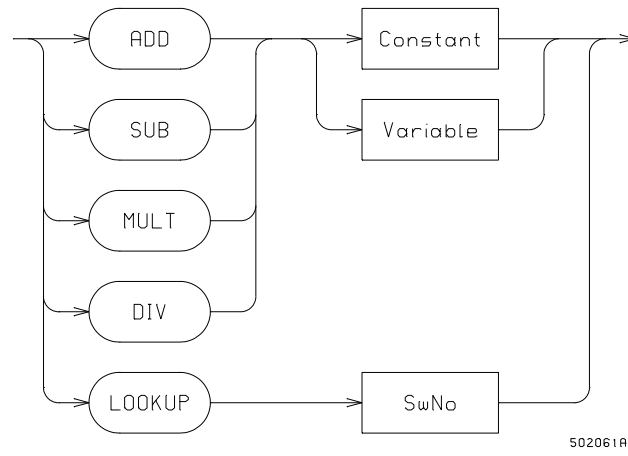


490 006 01

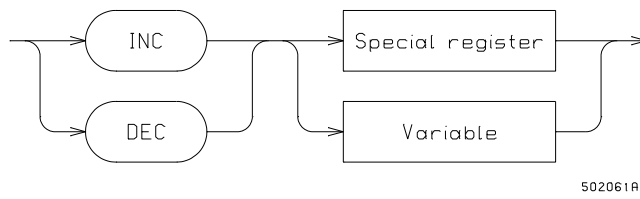


490 023 01

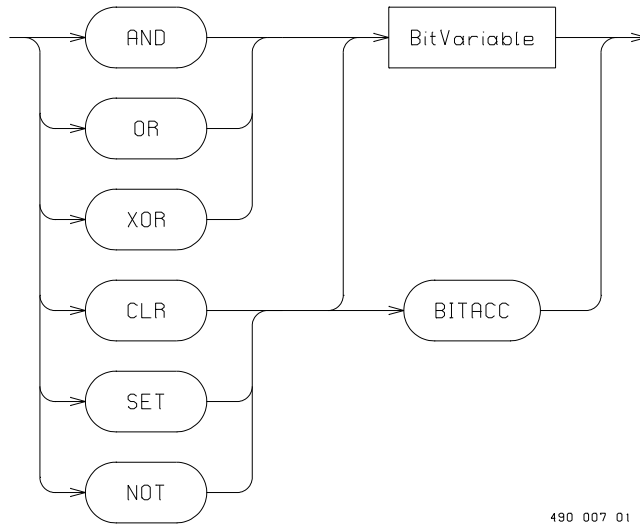
Arithmetic

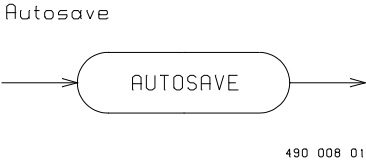
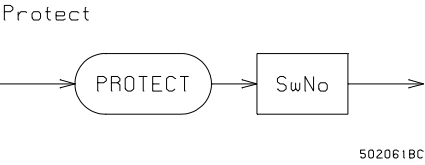
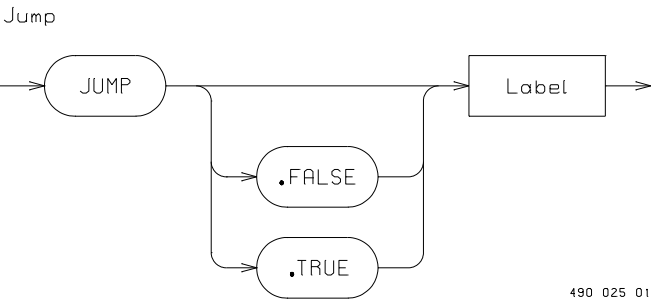
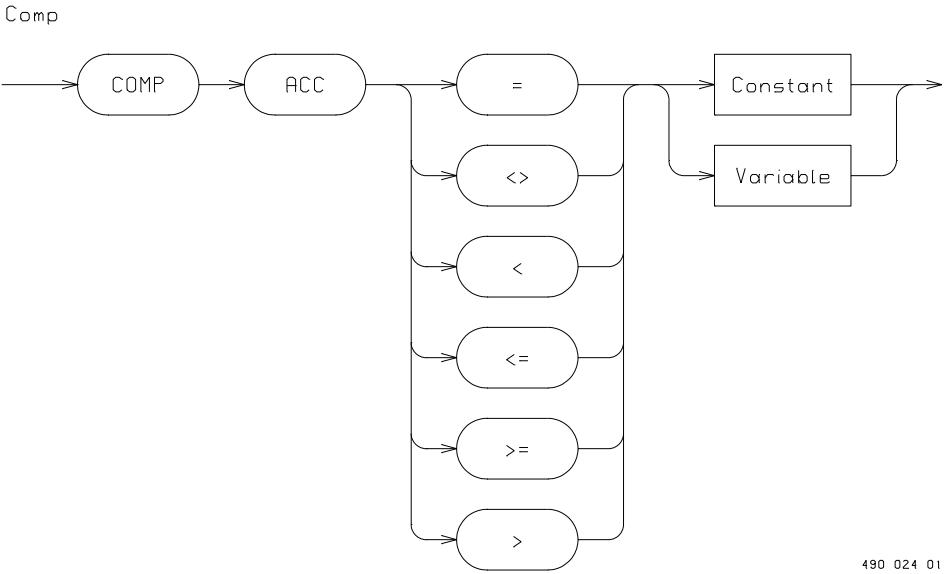


Inc/Dec

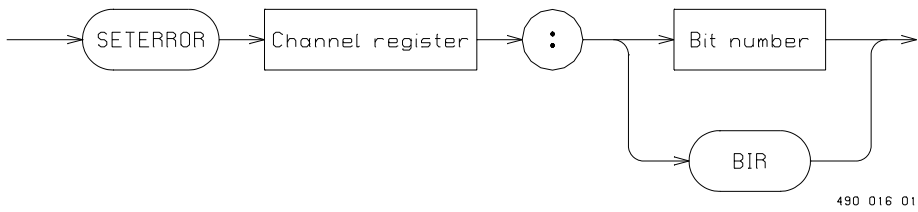


Logical

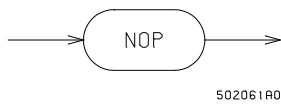




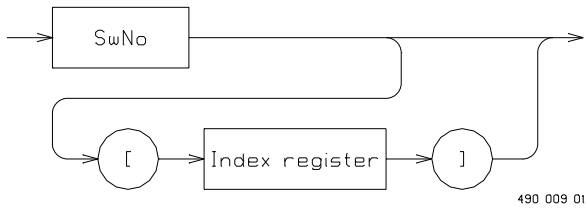
SetError



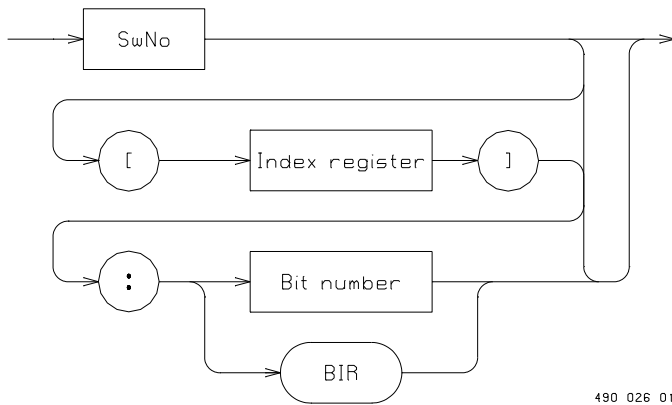
Nop



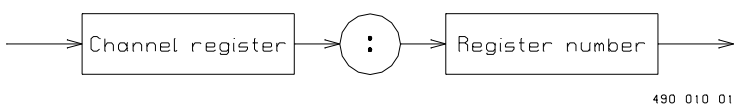
Variable



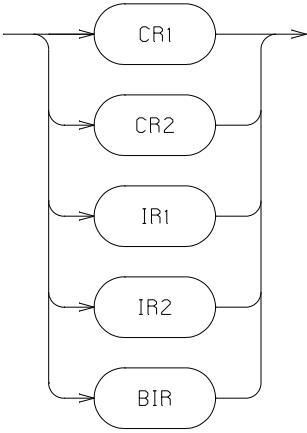
Bit variable



SwNo

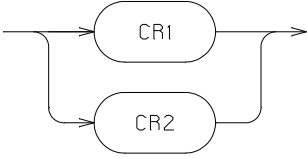


Special register



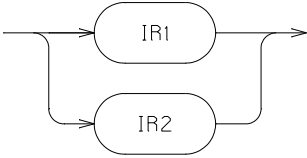
502061AU

Channel register



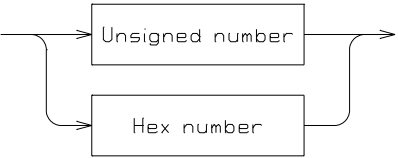
502061AV

Index register



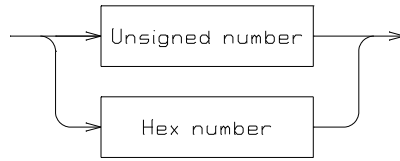
502061AX

Register number



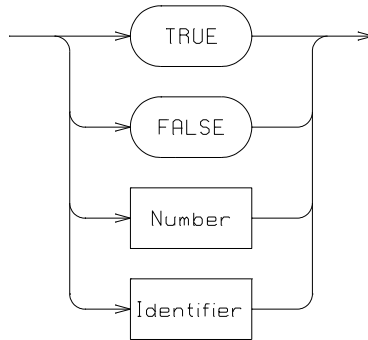
502061AY

Bit number



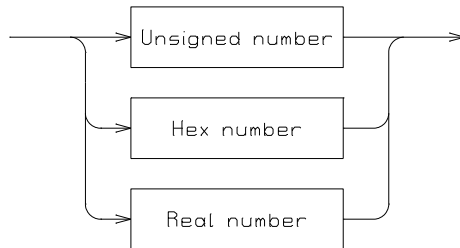
502061A2

Constant



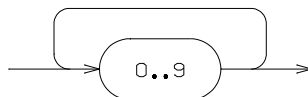
502061A0

Number



490 011 01

Unsigned number



490 012 01

