POSITIONING AND CONTOURING CONTROL SYSTEM

APCI-8001 AND APCI-8008

PROGRAMMING AND REFERENCE MANUAL / PM (PART 2)

Last updated: 31/03/2021, from disk V2.53VP
Rev. 15/052022

www.addi-data.com

5 The rw_SymPas programming language for stand-alone application programming

5.1 Introduction

rw_SymPas is a programming language for creating autonomously running CNC programs (stand-alone application programs) for the APCI-800x positioning control system. The lexical and semantic grammar of *rw_SymPas* is very similar to that of the *Pascal* programming language.

5.2 Lexical grammar

This chapter contains a formal definition of the lexical grammar used in *rw_SymPas*. This deals with the word-like units of a language, referred to as »symbols« or »tokens«. The semantic grammar determines the rules by which symbols can be combined to form expressions, statements or other units.
In *rw_SymPas,* the symbols are obtained as a result of the operations performed by the NCC compiler with the user program. An *rw_SymPas* program is a sequence of ASCII characters representing the source code and written with a text editor (e.g. CNC-Edit). The basic program unit in *rw_SymPas* is the file, which corresponds to a named DOS file in the memory or on the disk and has the extension ".SRC".

5.2.1 White space

In the lexical analytical phase of compiling, the source code file is parsed (broken down) into symbols and »white space«. White space is the collective term for characters categorized as separators: blanks, tabs, line breaks and comments. White space is used for marking the beginning and end of a symbol; but apart from this, white space is ignored.

5.2.2 Comments

Comments are text lines containing explanations on the program. They are removed from the source text prior to parsing.
An *rw_SymPas* comment is a character string located after the character "{". The comment ends at the first occurrence of the "}" character following the start symbol "{". Comments cannot be nested.
There is also an option for creating a one-line comment with two slashes "//". The comment can begin at any point and extends up to the next line.

5.2.3 Symbols

rw_SymPas recognizes the following kinds of symbol

Symbol:
- *Keyword*
- *Designator*
- *Qualified designator*
- *Labels*
- *Constant*
- *Operator*
- *Punctuation character (including separators)*

5.2.3.1 Keywords

Keywords are words reserved for special purposes, which may not be used as normal designator names. The table below lists all the *rw_SymPas* keywords.

Table 21: All *rw_SymPas* keywords

and	begin	boolean	const
do	double	downto	else
end	for	forward	function
goto	if	integer	label
mod	module	not	or
procedure	repeat	shl	shr
single	then	timer	to
until	var	while	xor

5.2.3.2 Designators

Designators can consist of the following elements:

Designator
    Non-figure
    Designator non-figure
    Designator figure

Non-figure: one of the following characters
a b c d e f g h i j k l m n o p q r s t u v w x y z _
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Figure: one of the following characters
0 1 2 3 4 5 6 7 8 9

Examples:

A, AA, AB, A1, A2, _A *// valid*
1A, ?B *// invalid*

5.2.3.2.1 Name and length restrictions

Designators can be any names of any length for variables, procedures, functions, label names, etc. Designators may contain the letters A to Z, a to z, the underscore and the figures 0 to 9. However, the following restrictions apply:

- The first character must be a letter or an underscore.
- Only the first 32 characters are significant. If the designator contains more than 32 characters, the remaining characters are ignored. In the case of large *rw_SymPas* programs, you should keep to short names, so as not to overload the PC's main memory.

5.2.3.2.2 Designator upper and lower case

rw_SymPas distinguishes between upper and lower-case letters, so that *Position*, *position* and *positioN* are different designators.

5.2.3.2.3 Unambiguity and validity of designators

Designators can be any names which conform to the applicable rules. Errors may, however, occur if the same name is used inside the same range of application for several different designators having the same name range. Identical names are permissible for different name ranges, irrespective of the range of application involved. The definition of a designator range of application is explained in chapter 5.3.2.2.

5.2.3.3 Standard designators

rw_SymPas already has a series of predefined designators, which are accordingly referred to as "standard designators". All *rw_SymPas* standard designators are listed in the table below.

Table 22: All standard designators predefined *rw_SymPas*

abort	Cl	contcnct	disev
enev	Ja	jaw	jhi
jhiw	Jhl	jhlw	jhr
jhrw	Jr	jrw	js
jsw	Mca	mcaw	mcr
mcrw	Mha	mhaw	mhr
mhrw	Mla	mlaw	mlr
mlrw	Ms	msw	ol
ra	Rs	shp	smca
smcr	Smha	smhr	smla
smlr	Ssms	ssmsw	startcnct
stop	Stopcnct	uf	wt
utrovr			

5.2.3.4 Axis designators

Each axis channel is referenced using a symbolic name. This name can be freely chosen by the user, with up to 8 characters. These axis designators are likewise incorporated in the standard designator list by *rw_SymPas*.

Note: Automatic declaration of the axis designators deviates from Standard Pascal.

5.2.3.5 Qualified designators

Referencing to designators of the same name which have been declared for different axis systems (by *rw_SymPas*) is handled in a qualifying routine by prefixing the axis designator.
Examples:

```
A1.digo := 0;                          // Reset all outputs of APCI-800x
A2.digo := $FFFFFFFF;     // Set all outputs of APCI-800x
```

Note: Variable referencing to qualified designators deviates from Standard Pascal.

5.2.3.6 Labels

The same rules apply for the structure of a label as for the designators. Labels are used solely in connection with the *goto* statement.

5.2.3.7 Constants

Constants are symbols which stand for fixed numerical values. *rw_SymPas* knows two classes of constants: floating-point and integer. A constant's data type is derived by the *NCC* compiler on the basis of its numerical value and its format in the source text. Table 23 shows the formal definition of a constant

Table 23: Formal definition of a constant.

Constant: Floating-point constant Integer constant
Floating-point constant: Fractional constant<exponent> Digit string exponent
Fractional constant: <Digit string>.Digit string Digit string.
Exponent: e<sign>Digit string E<sign>Digit string
Sign: one of the following characters + -
Digit string: Digit Digit string digit
Integer constant: <sign>Decimal constant Hexadecimal constant
Decimal constant: Digit Decimal constant digit
Hexadecimal constant: $ Hex digit Hexadecimal constant hex digit
Digit: 0 1 2 3 4 5 6 7 8 9
Hex digit: 0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

5.2.3.7.1 Integer constants

Integer constants can be decimal (base 10) or hexadecimal (base 16) numbers. Remember that different rules apply for decimal and non-decimal constants.

5.2.3.7.1.1 Decimal constants

Decimal constants of -2147483648 to 2147483647 are permitted. Constants outside this range will automatically be limited to the appropriate minimum or maximum value.

5.2.3.7.1.2 Hexadecimal constants

All constants which begin with the dollar sign ($) are interpreted as hexadecimal constants. Hexadecimal constants of $80000000 to $7FFFFFFF are permitted. Constants outside this range will be limited to the appropriate minimum or maximum value.

5.2.3.7.2 Floating-point constants

A floating-point constant is made up of 4 constituents:

- Places before the decimal point
- Decimal point
- Decimal places
- e or E and a signed integer exponent (optional))

You can omit either the places before the point or after it (but not both). The decimal point or the letter e (E) can be omitted (but not both). These rules enable you to use both the conventional and the scientific notation (with exponents).

5.2.3.7.2.1 The type of floating-point constants

Floating-point constants are always handled as double values. They are filed in a double word (8 bytes) in accordance with IEE. The range is $1.7*10^{-308}$ to $1.7*10^{308}$.

5.2.3.7.2.2 Declaration of constants

A constant declaration agrees a designator, which inside the block concerned stands for a constant value. An example of a constant declaration is:

```
const one = 1;
```

Signed constants stand for an integer or floating-point value. Computation of constants is not possible.

5.2.3.7.3 Punctuation characters

Punctuation characters (also referred to as separators) are defined in *rw_SymPas* as follows:

Punctuation characters: one of the following symbols
() , ; : =

5.2.3.7.3.1 Parentheses

Parentheses () group expressions together, isolate conditional expressions and represent procedure calls and procedure parameters:

```
d := c * (a + b);        // Alter the normal sequence
if (d = z) then ...      // Required with a conditional
                                 // statement
proc()                   // Procedure call without arguments
```

5.2.3.7.3.2 Comma

The comma (,) separates the elements in a procedure argument list:

```
mlr (A1, A2);
```

5.2.3.7.3.3 Semi-colon

The semi-colon (;) is used as the end criterion for a statement. Every valid *rw_SymPas* expression (including an empty expression) with a semi-colon at its end will be interpreted as a statement (expression statement).

5.2.3.7.3.4 Equals sign

The equals sign (=) separates constant declarations from the initialization values:

```
const one = 1.0;
```

5.3 Semantic grammar

This chapter will explain the formal definition of the *rw_SymPas* language structure. This semantic grammar determines the rules by which symbols can be combined to form expressions, statements or other meaningful units.

5.3.1 Declarations

The following section provides a brief summary of subjects involving declarations: objects, types, blocks, locality and range of application. Locality and range of application define those parts of the program from which the object linked to the designator can permissibly be accessed.

5.3.1.1 Objects

An object is an identifiable memory area in which a fixed or variable value (or a quantity of values) is located. Each object has a name and a type (referred to as the "data type"). An object is accessed over its name. This name can be a simple designator or a complex expression which unambiguously indicates an object. The type is used in order to:

- specify the correct memory reservation required at the beginning
- check the types so as to ensure that correct assignments are made

The predefined types of *rw_SymPas* include the Boolean data type, integer numbers with sign and floating-point numbers with differing accuracy.
Declarations establish the link between designators and objects. Each declaration links a designator to a data type. In addition, most declarations (referred to as the "definition declarations") also determine the generation of the object (where and when) and handle assignment of the memory location.

5.3.1.2 Types

Every declaration of a variable has to specify the type of this variable. The type specifies the value range of the variable concerned and determines the operations which can be performed with it. Thus a type definition agrees a designator, which in turn stands for a particular type.

Type declaration:
    Designator = Type;

Type:
    Boolean type
    Integer type
    Floating-point type

5.3.1.2.1 Boolean type

The *Boolean* data type can assume only one of the predefined values *FALSE* or *TRUE*. Note that the following relations apply:

- *FALSE* < *TRUE*
- Ordinal number of *FALSE* = 0
- Ordinal number of *TRUE* = 1

5.3.1.2.2 Integer type

rw_SymPas provides the integer types *Integer* and *Timer*.

Table 24: The integer type and its value range

Type	**Range**	Format
Integer	-2147483648 .. 2147483647	32 bits with sign
Timer	0 .. 4294967295	32 bits without sign

5.3.1.2.3 Floating-point types (real types)

rw_SymPas knows two different kinds of floating-point types: *Single* and *Double*. These types differ from each other both in their value ranges and in the accuracy of operations performed with them.

Note: Occasionally the term "real type" is also used for "floating-point type".

Table 25: The floating-point types and their accuracy

Type	**Range**	Format
Single	$-1.2e^{-38}$.. $3.4e^{38}$	7 to 8 places
Double	$-2.2e^{-308}$ .. $1.8e^{308}$	15 to 16 places

5.3.1.2.4 Assignment compatibility of types

Assignment compatibility is essential if a value is to be assigned. The value of a type T_2 can be assigned to a value T_1 (i.e. T_1:=T_2), if one of the following conditions is satisfied:

- T_1 and T_2 are of the same type.
- T_1 has the type double, T_2 the value integer or single.
- T_1 has the type single, T_2 the value integer.

If none of these conditions is satisfied, but assignment compatibility is required, the *NCC* compiler will report an error.

5.3.1.3 Variables

5.3.1.3.1 Automatic type conversion

rw_SymPas executes an automatic type conversion function if there are different types in one expression. Conversion is performed as follows: integer to single or integer and single to double. For example:

```
...
Var
        i : Integer;
        s : Single;
        d : Double;
...

d := s * i;                    // s and i are automatically converted to double

s := i;   // i is automatically converted to single
```

5.3.2 Blocks, locality and range of application

A block consists of declarations and statements arranged at will. Each block is part of a procedure declaration or of a program. All designators and labels in the block's declaration section are restricted in their effect to this block - they are local to this block.

5.3.2.1 Syntax

The syntactic structure of each block can be represented as follows:

Block:
  Declaration section
  Command section

5.3.2.1.1 Declaration section

Declaration section:
    Label declaration section
    Constant declaration section
    Variable declaration section
    Declaration section label declaration section
    Declaration section constant declaration section
    Declaration section variable declaration section

5.3.2.1.1.1 Label declaration section

In the *Label declaration section,* all labels are agreed which are to represent *goto* jump destinations in the command section of the block involved. Each label may be defined once only inside the command section (i.e. each *goto* must have an unambiguous destination).

Structure of the label declaration section:

label Labels;

Labels:
    LabelName
    Labels, LabelName

5.3.2.1.1.2 Constant declaration section

The declaration section for constants contains all agreements for constants which are local to the block involved.

Structure of the constant declaration section:

const constant declarations

Constant declarations:
    Constant declaration
    Constant declarations constant declaration

5.3.2.1.1.3 Variable declaration section

The declaration section for variables contains all variable declarations which are local to the block involved.

Structure of the variable declaration section:

var variable declarations

Variable declarations:
    Variable declaration
    Variable declarations variable declaration

5.3.2.1.2 Command section

It is in the command section that all operations are defined which are executed at block activation.

Command section:
Compound statement

Permissible compound statements are explained in chapter 5.3.5.5.

The command section of the main program block is structured as follows:

```
begin
        Statement list;
end.
```

5.3.2.2 Range of application

Each designator and each label of a declaration agrees precisely one object or jump destination. This is why a designator, like a label, must always be in its declaration's range of application when it appears in the program. The range of application for designators and labels lies between the actual declaration as such and the end of the block involved, with all those blocks being included which this block encloses. There are, however, a few exceptions to this, which are explained in the paragraphs below.

5.3.2.2.1 Redeclaration in a subordinate block

With the assumption that a block »outside« encloses a block i.e. is of a higher order, every redeclaration of a designator from »outside« in the block »inside« restricts this designator's range of application to the »inside« block. Or to put it another way: if a variable *x* is declared »outside« and a variable of the same name is declared »inside«, then statements in the block »inside« cannot access the variable *x* declared »outside«.

5.3.2.2.2 The location of a declaration in a block

Designators and labels must be declared before they can be used in a block. The *NCC* compiler will react to access attempts before such declaration with Error Number 3.

5.3.2.2.3 Redeclarations inside a block

Designators and labels can each be declared only once on the topmost level of a block, unless they are redeclared inside a subordinate block.

5.3.2.2.4 Standard designators

rw_SymPas offers a whole series of predefined constants, types and procedures, which work as if they had been declared inside a block covering the whole program. Consequently their range of application also covers the entire program.

5.3.3 Variables

5.3.3.1 The declaration of variables

The variable declaration contains a list of designators, which in their turn stand for new variables and their types.

Variable declaration:
    Designator list: type;

Designator list:
    Variable names
    Variable names, variable name

Type:
    BOOLEAN
    INTEGER
    SINGLE
    TIMER
    DOUBLE

Examples of valid variable declarations are:

```
var
    on, off:        BOOLEAN;
    one:            INTEGER;
    dvalue:         DOUBLE;
    ticks:          TIMER;
```

When a designator is located in the designator list of a declaration section, it applies inside the entire block for which it has been declared. Reference can be made to this variable throughout the block, provided the same designator is not being used for a different variable in a subordinate block ("redeclaration"). A redeclared variable uses the name of an already-existing designator, but otherwise represents an autonomous unit. The value of the original variable is not affected by the redeclaration. Variables or functions declared outside procedures are referred to as *global*. Variables declared inside procedures or functions are *local*.

5.3.3.1.1 Axis-type declaration

You can define variable axes with the AXIS type declaration. It is particularly useful for the design of sub-programmes (procedures/functions) used several times and in which recurrent actions are to be executed for several axes.

Example:

```
var
    VA : AXIS;                          // variable axis with name VA
    VA.an := 0;                         // Assign axis number 0 to  VA (important!)
    ol(VA);                             // open loop of axis 0
    for VA.an := 0 to 5 do  cl(VA);     // close loop axes 0 .. 5
```

Note: The axes numbers of the predefined axes specifiers (A1 .. An) that are defined by the system parameters can also be assigned anew in this way. This can yet cause a confused and incorrect programming. Should you operate with variable axes, the use of the corresponding variables with corresponding symbol character is recommended.

Notice: Currently, variable axes may be declared only in the main program, i.e. not in procedures or functions!

5.3.3.1.2 Timer declaration

An entry with the aid of the predefined system variable *CLOCK* supplies a value of the *timer* type, which represents the time. This value is supplied by the control's internal clock, which alters its value at regular intervals. This value continues cyclically, i.e. after the largest positive value the next value supplied is the smallest negative value. The time interval in which this internal clock is incremented is 64 µs.
CLOCK can be used at any time to assign the counter reading of this clock to an *integer* or *timer* variable. If different times have to be compared with each other, this comparison should be carried out only by means of *timer* variables, since here a timer overflow will automatically be taken into account at the comparison operator >. Likewise, addition and subtraction with *timer* variables is performed in modulo technique, i.e. without signs. With *integer* variables, conversely, there may be an overflow or underflow, which in turn will cause the internal error flag to be set and in certain situations will trigger an abort of the *rw_MOS* operating system.
A practical timer application might look like this:

```
Const
        s := 15625;             // 15,625 ticks = 1s
Var
        t: timer

t := CLOCK + 5*s;               // Compute time-delay of 5 s from now
...
repeat
        ...
until CLOCK > t;                // wait until 5 s have passed
...
```

In this example, you can see that the addition of *CLOCK* and the time-delay results in an overflow at large values for *CLOCK*. We therefore recommend using the *timer* instead of the *integer* type for declaration of the variable *t*. Another reason is the interrogation of whether the computed time-delay has been reached. In the event of an overflow in computing the time-delay, you see, the content of *t* is smaller than *CLOCK*. This circumstance is likewise handled properly by the declaration as a *timer* variable.
The value range of a timer variable lies between 0 and 4294967295. Time-delays of up to 38h can be implemented.

5.3.3.2 Conversion of variable types

The reference to a variable of a particular type can be converted into a reference to a variable of a different type.

Type conversion:
  Type designator (variable reference)

Type designator:
  BOOLEAN
  INTEGER
  SINGLE
  DOUBLE

A few examples for the conversion of variable types:

```
var
        B : BOOLEAN;
        I : INTEGER;
        D : DOUBLE;

        B := BOOLEAN (I);
        B := BOOLEAN (D);
        D := B;
        I := INTEGER (D);
        I := B;
```

5.3.4 Expressions

Expressions consist of operators and operands. Most operators of *rw_SymPas* link two operands and are therefore referred to as binary. The remaining operators work with only one operand and are therefore referred to as unary. Binary operators utilize the conventional algebraic form like *a+b*. A unary operator is always positioned immediately before its operand, as with *-b*. In the case of extensive expressions, the order of *precedence* shown in Table 26 governs the sequence of computation. Three basic rules apply:

- An operand between two operators of different precedence rankings is always linked to the higher-ranking operator.
- An operand between equal-ranking operators is always linked to the operator located to the left of it.
- Expressions in brackets are regarded as a single operand and always evaluated first.

Table 26: Operator precedence

Operators	Precedence	Category
-, +, not	1 (highest)	unary
*, /, mod, shl, shr and	2	multiplying
+, -, or, xor	3	adding
=, <>, <, >, <=, >=	4	relational

Operations of the same precedence ranking are normally performed from left to right.

5.3.4.1 Syntax of expressions

The order of precedence for operators follows the syntax for expressions composed of factors, terms and simple expressions. Factors can be represented by the following syntax:

Factor:
variable reference
unsigned constant
(expression)
not factor
type conversion (values)

unsigned constant:
unsigned numerical value
character string
constant designator

The following particulars represent valid factors:

Dummy	variable reference
15	unsigned constant

5.3.4.2 Operators

We distinguish between four groups of operators: arithmetical, logic, boolean and relational operators.

5.3.4.3 Arithmetical operators

The tables below show the types of operand and result involved in binary and unary arithmetical operations.

Table 27: Binary arithmetical operators

Operator	Operation	Operand type	Result type
+	Addition	Integer, Real	Integer, Real
-	Subtraction	Integer, Real	Integer, Real
*	Multiplication	Integer, Real	Integer, Real
/	Division	Integer, Real	Integer, Real
mod	Modulo	Integer	Integer

Note: If one of the operands is of the *Timer* type, addition and subtraction are performed using the modulo technique. No overflow check is made, since the *Timer* values are cyclical. You will find more details on the *Timer* type in Chapter 5.3.3.1(b)

Table 28: Unary arithmetical operators

Operator	Operation	Operand type	Result type
+	Identity	Integer, Real	Integer, Real
-	Negation	Integer, Real	Integer, Real

If both of an operator's operands +, -, *, /, or *mod* have an integer type, the result will likewise be of the integer type. If one of an operator's operands is +, -, *, or / is of the *Real* type, then the result will likewise be of the *Real* type.
The *mod* operator returns the rest of the division of its operands as follows:

*i mod j = i - (i/j)*j;*

5.3.4.4 Logic operators

Table 29 shows the types of operand involved and the results of logic operations.

Table 29: Logic operations

Operator	Operation	Operand type	Result type
Not	bitwise negation	Integer	Integer
And	bitwise AND	Integer	Integer
Or	bitwise OR	Integer	Integer
Xor	bitwise exclusive OR	Integer	Integer
Shl	Shift left	Integer	Integer
Shr	Shift right	Integer	Integer

Note: *not* is a unary operator.
The i *shl* j and i *shr* j operations shift the value of *i* by *j* bit positions to the left or the right and thus correspond to a multiplication or division by 2^j.

5.3.4.5 Boolean operators

Table 30 shows the types of operand involved and the results of Boolean operations.

Table 30: Boolean operators

Operator	**Operation**	**Operand type**	**Result type**
not	logic negation	Boolean	Boolean
and	logic AND	Boolean	Boolean
or	logic OR	Boolean	Boolean
xor	logic exclusive OR	Boolean	Boolean

Note: the operator **not** is unary here as well.
In the case of operands of the *Boolean* type, normal Boolean logic determines the result of these operations. For example, *a and b* will only give *TRUE* when *a* and *b* are both true

5.3.4.6 Relational operators

Table 31 shows the operand types involved and the results of relational operations.

Table 31: Relational operators

Operator	Operation	Operand type	Result type
=	equal	Integer, Real	Boolean
<>	unequal	Integer, Real	Boolean
<	smaller than	Integer, Real	Boolean
>	greater than	Integer, Real	Boolean
<=	smaller than/equal	Integer, Real	Boolean
>=	greater than/equal	Integer, Real	Boolean

Note: If one of the operands is of the *Timer* type, the *greater than* (>) operation is performed using the modulo technique. No overflow check is made, since the *Timer* values are cyclical. You will find more details on the *Timer* type in chapter 5.3.3.1(b).

5.3.5 Statements

This term “statement” stands for all constructs which agree an action which can be executed by the APCI-800x board. In this manual, the term »statement« is used as a generic term for statements (like *begin*, *end* or *for*) and *commands* (like *goto*, assignments, procedure calls, etc.).
Each statement (i.e. each agreement for an executable action) can be preceded by a label, which in its turn can be referenced with *goto*: a *goto* this label causes a direct jump to this statement and its execution.

Structure of a statement:

Label: statement

Statement:
Assignment
Procedure statement
Goto statement

5.3.5.1 Assignments

Assignments replace the instantaneous value of a variable with a new value, which is specified by mean of an expression.

Structure of an assignment:

Variable reference **:=** expression

5.3.5.2 Procedure or function calls

A procedure is called by specifying a procedure designator with which the procedure concerned has been declared. Parameter transfer to the procedure is supported only from mcfg.exe V2.5.3.97 (ncc.exe/dll V2.5.3.73). By indicating a function designator, a function that has been declared with this designator is called. The use of user-defined functions is supported only from mcfg.exe V2.5.3.97 (ncc.exe/dll V2.5.3.73) and RWMOS.ELF from V2.5.3.126.

5.3.5.3 The goto statement

executes a jump to the label specified: the program is continued at a point immediately following the label concerned. The syntax of *goto* is:

goto Label

When *goto* is used, the following rules must be observed:

- The label to which *goto* is referenced must be located in the same block as the *goto* statement itself. It is not possible to jump back and forth at will between procedures/functions with *goto*.
- Referencing to a structured statement block from a program section outside this block (i.e. a jump to a deeper nesting level) may have unforeseeable consequences. *rw_SymPas* cannot detect errors of this sort.

5.3.5.4 Structured instructions

consist of several interested levels, which in their turn contain statements. They are executed either in the order of their appearance (compound statements), conditionally (conditional statements) or repeatedly (repeat statements or loops).

Structured statement:
block command
conditional statement
repeat statement

5.3.5.5 Compound statements

Compound statements specify that the individual components they contain are to be executed in the order in which they appear in the source text concerned. All statements contained in the compound are handled as one single block and thus satisfy the requirements at points where the syntax of *rw_SymPas* permits only a single statement. Beginning and end of a compound are indicated by *begin* and *end*, with the individual components separated from each other by semi-colons.

A compound statement can be represented as follows:

begin statement list **end;**

Statement list:
  statement;
  statement list statement;

Example:

```
// ...
var
          i: Integer;
          j: Integer;
          temp: Integer;
// ...
begin
          if (i > 0) then i := 0;
          else begin
                    // interchange j and i
                    temp := i;
                    i := j;
                    j := temp;
          end;
end.
```

5.3.5.6 Conditional statements

Conditional statements offer one or more options and select one of their components (or none) for an instruction.

5.3.5.6.1 The if statement

can be represented as follows:

if (conditional expression) **then** w-statement <**else** f-statement>

The brackets around *Conditional expression* are not absolutely necessary. The result of *Conditional expression* must be of the standard *Boolean* type. If *Conditional expression* is TRUE, then w-statement will be executed; otherwise w-statement will be ignored.
If the optional *else f-statement* is present and *Conditional expression* is true, then *w-statement* will be executed; otherwise *w-statement* will be ignored and *f-statement* executed.
The statements *f-statement* and *w-statement* may themselves be *if-statements*, thus enabling a nested conditional test to be implemented in almost any depth you want. You have to be very cautious in using nested *if. else* constructs - make absolutely sure that the correct statements are chosen. *Else* ambiguities are resolved by assigning an *else* to the last *if*-without-*else* occurring on the same nesting depth. Compound statements are also permissible for *w-statement* and *f-statement*.

5.3.5.7 Loops

Loops (or repeat statements) specify the repeated execution of defined program sections.

Loop:
  while statement
  repeat statement
  for statement

5.3.5.7.1 The while statement

The format for a **while** statement is:

while (conditional expression) **do** w-statement;

The brackets around *Conditional expression* are not absolutely necessary. The loop statement *w-statement* will be executed as long as the *Conditional expression* gives the value FALSE. The *Conditional expression* is evaluated and tested beforehand. If the value obtained is TRUE, then *w-statement* will be executed. If the program does not encounter any jump statements, causing it to leave the loop, the *Conditional expression* will be evaluated anew. This operation is repeated until *Conditional expression* gives the value FALSE. If there are no jump statements, then *w-statement* must influence the value of *Conditional expression*, or *Conditional expression* itself must alter during evaluation, so as to avoid endless loops. Compound statements are also permissible for *w-statement*.

5.3.5.7.2 The repeat statement

The format for a *repeat* statement reads:

repeat r-statement **until** (conditional expression);

The brackets around *Conditional expression are not absolutely necessary*. The *r-statement* is executed as long as *Conditional expression has the value* FALSE. In contrast to the *while* statement, *Conditional expression* is tested not before, but after every execution of the loop statement. *r-statement* will accordingly be executed at least once.
Compound statements are also permissible for *r-statement*.

5.3.5.7.3 The for statement

The format for a *for* statement reads:

for controlled variable := Start value **to/downto** final value **do** f-statement;

The controlled variable must be the designator of an integer-type variable, which has been declared either inside the same block locally like the *for* statement, or globally for the entire program. The definition of a loop with *for* includes the specification of a *start* and *final value* as well. Both these values must likewise be of the integer type, which is assignment-compatible to that of the controlled variable.
When the loop is started, the controlled variable is set to the *start value* and increased or reduced by one each time the loop is run - until the *final value* is reached. In each run, the *f-statement* or compound statement contained in the rump of the loop is executed once. If the final condition of the loop is already given before the first run (i.e. *final value* < *start value* or *final value* > *start value when downto* is being used), then the loop and its rump will be skipped completely.

5.3.6 Procedures and functions

In formal terms, procedures and functions represent additional levels inside the main program block, i.e. a nesting feature. A procedure is activated by a procedure call (i.e. specification of a designator) and does not return a direct value. A function is activated during the computation of an expression in which its designator appears and normally has a result which can for this call be equated with the function designator.

5.3.6.1 Procedure declarations

A declaration initiated with the reserved word *procedure* links a designator and a block of statements for a procedure. Procedures declared in this manner can be activated (i.e. called) by specifying their designator. A procedure declaration has the following formal structure:

Procedure header; procedure block;

The procedure header names the procedure (i.e. assigns a designator to it). From mcfg.exe V2.5.3.97 (ncc.exe/dll V2.5.3.73), the declaration of parameters is possible at this point. Data can be exchanged between the main program and a procedure via global variables or these parameters. A procedure is activated by specifying its designator: The actions defined in the command section of the corresponding procedure declaration are executed.
A procedure which contains its own statement as part of its command section is executed recursively, i.e. it calls itself repeatedly. In this context, a suitable criterion must be found for aborting the recursion before the internal CNC task stack overflows.
Nesting procedures in *rw_SymPas* is only possible if the procedure header with optional parameters has been declared as *forward*. For this, the procedure header must be followed by the keyword *forward* and another semicolon. A forward declaration is only required if the procedure is to be used before the completion of the actual declaration (with procedure block). Between the procedure header and the procedure block, local variables and labels can be declared. Also the forward declaration is only possible from mcfg.exe V2.5.3.97 (ncc.exe/dll V2.5.3.73).

Examples:

```
procedure ProcA; forward;
...
procedure ProcA;
begin
        (procedure block)
end;

procedure ProcB (ParamA, ParamB : integer; ParamC : double); forward;
...
procedure ProcB (ParamA, ParamB : integer; ParamC : double);
begin
        (procedure block using ParamA, ParamB and ParamC)
end;

procedure ProcC (ParamA : integer);
var locVarA, locVarB : integer;
    locVarC : double;
begin
        (procedure block)
end;
```

5.3.6.2 Function declarations

Note: In *rw_SymPas*, the function declaration implemented according to *Pascal* standard is possible from V2.5.3.97 (ncc.exe/dll V2.5.3.73). For its execution, rwmos.elf from V2.5.3.126 is required; otherwise, an rw_SymPas program is quit with runtime error 4 when returning from the function. Moreover, there are various predefined system functions.

The functions (system functions and user-defined functions) are activated during the computation of expressions in which their designator appears and stand there for the value they return. A function designator can be inserted anywhere in an expression in place of an operand, provided the type of the function result concerned is compatible with that of the operand replaced.

Assignments to a function designator are not permitted.
A function is called by specifying its designator, followed by a list of current parameters, which in type and sequence must conform to the formal parameters of the correspondingly predefined function. The return value of a function must be used; otherwise, compilation error 91 is displayed.

The declaration of a user-defined function initiated with the reserved word *function* combines a designator, type declarations and a block of statements to form a function. Functions declared in this manner can be activated (i.e. called) by specifying their designator. A function declaration has the following formal structure:

Function header; function block;

The function header names the function (i.e. assigns a designator to it) and defines parameters and the function type. Data can be exchanged between the main program and a function via global variables or via parameters and the return value. A function is activated by specifying its designator: The actions defined in the command section of the corresponding function declaration are executed.
A function which contains its own statement as part of its command section is executed recursively, i.e. it calls itself repeatedly. In this context, a suitable criterion must be found for aborting the recursion before the internal CNC task stack overflows.
Nesting functions in *rw_SymPas* is only possible if the function header with optional parameters and function type has been declared as *forward*. For this, the function header must be followed by the keyword *forward* and another semicolon. A forward declaration is only required if the function is to be used before the completion of the actual declaration (with function block). In the function block, the return value of a variable must be assigned with the designation of the function name. This variable for the return value can be used within the function as a local variable. If the function is to be called recursively, the function name must be used with a pair of parentheses, which contain the parameters. For functions without parameters, the function call in this case must be marked by a blank pair of parentheses.
Between the function header and the function block, local variables and labels can be declared (see example for the procedures).

Examples:

```
function FuncA  : integer; forward;
...
function FuncA  : integer;
begin
        (funktion block)
        FuncA := function return value;
end;
```

```
function FuncB (ParamA, ParamB : integer; ParamC : double) : double; forward;
...
function FuncB (ParamA, ParamB : integer; ParamC : double) : double; forward;
begin
        (function block using ParamA, ParamB and ParamC)
        FuncB := function return value;
end;
```

In the examples above, *function return value* may be any expression.

5.3.7 The syntax of an *rw_SymPas* program

An *rw_SymPas* program is similar in form to a procedure declaration. The differences are merely in the program descriptor.

rw_SymPas program:

Program descriptor; program block

5.3.7.1 The program descriptor

The program descriptor specifies the name of a program, but has no special significance of its own.

Program descriptor:
program designator

Example:

```
program Test;
```

5.3.7.2 The program block

Program block:
Implementation section
Procedure/function command section
Initialization section

Implementation section:
Constant declaration
Variable declaration
Implementation section constant declaration
Implementation section variable declaration

Initialization section:
begin
Command section
end

The initialization section is the final constituent part of an *rw_SymPas* program and represents the main program. It consists of a block initiated with **begin**, which contains statements and is concluded by a terminating **end**. The entire program block is concluded with the (**.**) character.

6 Stand-alone application programming

6.1 Introduction

The *rw_SymPas* programming language incorporates a comprehensive set of commands, which you can use for flexible, efficient program creation. The procedure calls are performed in accordance with *Pascal c*onvention, apart from a few exceptions.
Since the procedure names and also the functioning of the individual procedures are identical for the two programming methods involved - stand-alone application programming [SAP] and PC application programming [PCAP], a detailed description is provided here only for the commands involved in PCAP programming.
The individual commands are listed in alphabetical order.

6.2 rw_SymPas example programs

The *rw_SymPas* example programs included in the APCI-800x TOOLSET software show how simple it is to use the functions described below. The source texts for the example programs incorporate comments to make them self-explanatory. So there is no need to go into a detailed description of these example programs here. They all have the file extension .SRC and can be found in the SAP subdirectory of the APCI-800x TOOLSET software floppy.

6.3 Abbreviations, system parameters, axis specifiers and axis qualifiers

For the SAP function reference list printed below, we will start off by explaining the various abbreviations and types involved, some of which are used as parameters for the different functions in question.

6.3.1 System parameters

The system parameters predefined by the *rw_SymPas* programming language are listed in tabular form, with an explanation of how they function. Remember that the *NCC* compiler distinguishes between upper and lower case for these parameters.

Table 32: *rw_SymPas* predefined system parameters

Name	Type	Abbr. meaning	Function
BOARD TYPE	integer	Board-Typ	Hardware version of the control type (see chapter 4.4.18)
CI0..CI99	integer	Common Integer 0..999	100 predefined integer variables for data exchange or for synchronization with a PC application program running in parallel. Further information at the PCAP commands rdci() and wrci().
CD0.. CD99	double	Common Double 0..999	100 predefined double variables. Otherwise as for CI0..CI999. Further information at the PCAP commands rdcd() and wrcd().
CFLAG	integer	ControllerFlags	Access to the ControllerFlags register (see Chapter 6.3.1.4)
DTCA1	double	Distance-to-Center A1	Indication of medium point for helical profiles and 3 D circles for the X circle axis
DTCA2	double	Distance-to-Center A2	Indication of medium point for helical profile and 3 D circles for the Y circle axis
DTCA3	double	Distance-to-Center A3	Indication of medium point for 3 D circles for tze Z circle axis
ERROR REG	integer	error register	Bit coded error register in which internal error states of RWMOS.ELF are indicated.
IRQPC	boolean	Interrupt Request PC	PC interrupt request, active when TRUE
LEDGN	boolean	Led green	Green LED on APCI-800x, switched on when TRUE
LEDRD	boolean	Led red	Red LED, otherwise as for LEDGN
LEDYL	boolean	Led yellow	Yellow LED, otherwise as for LEDGN
LET	double	Latch End Time	Time for the recording of the graphical system analysis in seconds from the moment LST. See the commands LPR and LPRS. Basically 1000 values are always recorded. The value entered in LET is always rounded up in integral multiples of 1000 * T_A. T_A is set to 1.28ms as a standard.
LST	double	Latch Start Time	Moment for the begining of the recording the graphical system analysis in seconds from the moment of the calling up. See the commands LPR and LPRS.
MODEREG	integer	Mode Register	Bit coded register to control the operating system functionality (see chapter 6.3.1.5)
NFRAX	integer	No-Feed-Rate-Axis	In this variable the axes ca be defined as bit coded. They are not utilized for the calculation of the trajectory velocity by interpolation movements.
NOA	integer	Number of Axis	This system variable includes the number of the axes actually available in the system and cannot be written.
OSVERSION	integer	Operating system – version information	The predefined system parameter OSVERSION (Type) returns the current operating system version number of the rwmo.elf file while the SAP program is running. The version number contains a primary number and secondary number. Yet the primary number is incremented in 1000 steps and the secondary in 1 steps. The version number 253042 e.g. means that the primary number is 2.5.3 and the secondary is 042.
PHI	double		Traverse angle for circular and helical profiles
DTCA1	double	Distance-to-Center A1	Center point for helical profile with target point for the X axis

Name	Type	Abbr. meaning	Function
DTCA2	double	Distance-to-Center A2	Center point for helical profile with target point for the Y axis
NFRAX	integer	No-Feed-Rate-Axis	Axes can be coded in bits, which are not utilized by interpolation for the calculation of the trajectory velocity.
PN1	double	Plane-Normal	Surface normals for MCA3D command. Additional Information at the command MCA3D.
PN2	double	Plane-Normal	Surface normals for MCA3D command. Additional Information at the command MCA3D.
PN3	double	Plane-Normal	Surface normals for MCA3D command. Additional Information at the command MCA3D.
PU	integer	Position Unit	Index for position unit (Table 33)
SSFP	integer	Spool-Special-Function-Parameter	Function parameter for specific functions in the spool operating mode. Additional information at the SAP command *SSF*
TRAC	double	Trajectory Acceleration	Trajectory acceleration for linear, circular and helical profiles. The unit of this parameter is defined in TU und PU.
TROVR	double	Trajectory Override	Trajectory velocity correction value
TROVRST	double	Trajectory Override Settling Time	Time for the settling of the trajectory velocity correction value
TRTVL	double	Trajectory Target Velocity	Trajectory target velocity for linear, circular and helical profiles. The unit of this parameter is defined in TU und PU.
TRVL	double	Trajectory Velocity	Trajectory velocity for linear, circular and helical profiles. The unit of this parameter is defined in TU und PU.
TU	integer	Time Unit	Index for time unit (Table 34)

6.3.1.1 PC interrupt generation

You can use the *IRQPC* system parameter to trigger a hardware interrupt on the PC. This option offers an efficient approach for using the two programming methods: PC application and stand-alone application programming. A stand-alone program can be used for largely autonomous process sequence, which needs to interrupt the parallel-running PC program only if necessary, or in the event of an error. The program is then interrupted with the aid of this interrupt generation feature. After the PC program has detected the hardware interrupt, the common variables listed above can be used for exchanging data between the two parallel-running programs.

Note: The hardware configuration for PC interrupt generation (PCI-Interrupt) is automatically given with the aid of the Plug & Play properties integrated on the APCI-800x board and manage through the system driver mcug3.dll. The user has only to define a PCAP user routine with predefined structure and to inform the driver. Once the APCI-800x board has generated a hardware interrupt, the corresponding user routine is called up automatically. The mcug3.dll driver is structured in such a way that other PCI interrupts, which use the same interrupt sources can be called up as well.
The driver software that is contained in the scope of delivery, supplies functions in order to easily install an interrupt service routine and, if required, to uninstall it (see chapters 4.4.33 and 4.4.34).

6.3.1.2 System parameters for unit processing

All *move commands* of the *rw_SymPas* programming language require specification of the acceleration *(TRAC)*, velocity *(TRTVL, TRVL)* and position parameters, each in selected distance and time units. You can use the two system parameters listed below to switch over the path unit (PU) and time unit (TU) parameters any time you want.

Table 33: System parameter PU

Value	Unit	Abbr. meaning
0	Mm	Millimeter
1	Inch	Inch
2	M	Meter
3	Rev	Revolution
4	Deg	Degree
5	Rad	Radiant
6	Counts	Counts
7	Steps	Steps

Table 34: System-Parameter TU

Value	Unit	Abbr. meaning
0	Sec	Seconds
1	Min	Minutes
2	Tsample	Sampling time

Note: The default values for TU and PU are specified in the [Setup][Set CNC-specific parameters] menu in the CNC Editor environment.
The units selected are used only for interpolation commands (all *move* commands)! If the commands concerned are axis-specific motion ones (all *jog* commands), the axis units specified in *mcfg.exe* are taken into account. There is no option here for switching over during the run time.

6.3.1.3 ERRORREG

In this bit-coded register, runtime errors of the RWMOS operating system software are specified. The bit assignment can be found in Table 15 in Chapter 4.4.63.1.

The ERRORREG register is only reset if it is written on with 0 or by a system boot.

6.3.1.4 ControllerFlags

This is an axis-specific bit-coded register by means of which special options in the position controller of the motion control boards can be activated. There are options for the behaviour of the position controller during traversing and standstill. This register is only effective with servo-axes having PID filter characteristics; with stepper axes, it has no effect. The register can be accessed using the dll functions wrControllerFlags (Chapter 4.4.137) and rdControllerFlags (Chapter 4.4.51). From the rw_SymPas programming, the register can be accessed using the axis qualifier CFLAGS.

Table 35: Description of the ControllerFlags register

Bit # / Hex	Name	Description
0 / 0001H	***MoveControl***	Activate all Move... flags
1 / 0002H	***MoveZero***	Delete the integral component of the axis controller once the axis is traversed.
2 / 0004H	***MoveDeadBand***	Limit the integral component of the axis controller once the axis is traversed and the control difference of the axis is outside the deadband. In the ControllerParams field [8][0], the deadband is specified in digits (see also Chapter 4.4.114).
3 / 0008H	***MoveFix***	Retain the integral component of the axis controller and do not integrate it further once the axis is traversed.
4 / 0010H		not used
5 / 0020H		not used
6 / 0040H		not used
7 / 0080H		not used
8 / 0100H	***StopControl***	Activate all Stop... flags
9 / 0200H	***StopZero***	Delete the integral component of the axis controller once the axis stands still.
10 / 0400H	***StopDeadBand***	Limit the integral component of the axis controller once the axis stands still and the control difference of the axis is outside the deadband. In the ControllerParams field [8][0], the deadband is specified in digits (see also Chapter 4.4.114).
11 / 0800H	***StopFix***	Retain the integral component of the axis controller and do not integrate it further once the axis stands still.
12 - 31		not used

Move... means that the relevant flags take effect once the corresponding axis is traversed. Stop... means that the relevant flags take effect once the corresponding axis is position-controlled. If StopControl and MoveControl are set to FALSE, all other bits are ineffective. To display these flags and set them for test purposes, the utility program McuControlInit.exe, which is described in the Operating Manual (OM), is availabe.

6.3.1.5 MODEREG

With the bit-coded register different options of the operating system software RWMOS.ELF can be set. As a standard all bits are set to 0.

Note: When a bit is to be set or reset in this register, you must pay attention that the other bits are not modified. For this, it is necessary to read MODEREG before writing, to process the content with boolean operations and then to write again. Bits are set with boolean Or connection, and reset with And connection. Bits which are currently not assigned must not be used as they are reserved for future extensions.

Table 36: Bit-coding MODEREG

Bit # / Hex	Name	Description
0 / 0001H	***LookAhead***	With this bit, the look-ahead functionality of the RWMOS operating system software is activated. The given target velocity of interpolation profiles is limited so that the maximum axis-specific velocity jump MDVEL is not exceeded with any axis and that all axes are at rest by the end of the interpolation travel. This mode is only valid for the commands SMLA, SMLR, SMCA, SMCR, SMHA, SMHR. In addition, in this mode, the trajectory velocity in circle commands is limited in such a way that the axis-specific maximum acceleration (MaxAcc) is not exceeded with any axis involved. The maximum velocity is calculated as follows: √ (circular radius * >MaxAcc). For this, see also Chapters 4.4.160, 4.4.85 and 6.3.3.
1 / 0002H	***S-Profil***	By setting this bit, the acceleration and braking ramps are run with S-fom velocity rise/drop. This option can be parameterised with the axis qualifier JERKREL.
2 / 0004H		free for future use
3 / 0008H	***WkzRadKorr***	Tool radius correction (only for TC option) The tool radius correction is described in a separate manual. Ask for it if you need it.
4 / 0010H		free for future use
5 / 0020H	***AutoSpool***	When travel profiles are spooled in SAP programs, the spooler is checked by the activated option. Once the spooler is full, the spooler processing is automatically started by the axes selected in the current axes. Further profiles are only entered when memory is available. This option can be used for the following travel commands: SMLA, SMLR, SMCR, SMCA, SMHA, SMHR and G01 by DIN66025
6 / 0040H	***NoTriangle***	Triangular profiles in look-ahead mode are disabled. In case a sub-profile cannot reach the programmed trajectory velocity, the current start velocity remains. This enable a correcter running for short travel profile parts without continuous acceleration and braking phases.
7 / 0080H	***ChkMaxVel***	In this mode, the trajectory velocity and the trajectory acceleration of all axes are limited by spooled linear interpolation commands so that no axis exceeds the maximum values set in MAXVEL and MAXACC. No-Feedrate axes are also included in this monitoring (see Table 32– NFRAX).
8 / 0100H	***ExactTargetPos***	Usually at the end of an travel profile which has a target velocity of 0, the setpoint position is rounded up in integral value of the system resolution. It is for example by stepper motor systems a step or by encoder systems an encoder counting pulse. It can cause an error by connecting relative profiles with each other. This rounding up can be switched off by setting this bit.
9 / 0200H	***ShortestRotatoric Distance***	When this bit is set by rotary axes, Jog absolute commands (JA) are run in the direction in which the shortest traverse distance is required.
10 / 0400H	***RotatoricUnit***	When this bit is set, the target position / the traverse distance are given in the axis-specific rotatory unit for rotatory axes which must be travelled with translatory axes per interpolation command.
11 / 0800H	***ForbidTargetVel***	If this bit is set, a system reset (rs) is executed, if the traverse profiles are terminated with a target speed <> 0. In this case in the system variable ErrorReg Bit 1 is set.
12 / 1000H	***CenterAlwaysRel***	Circle centers at G-Codes G02 and G03 are to be interpeted always a relative coordinates.
13 / 2000H	***NoLsmCheck***	By setting this flag, the automatic spooler monitoring at G01 of the G-

Bit # / Hex	Name	Description
		Code interpreter (McuWIN) can be switched off.
14 / 4000H		Free for future use
15 / 8000H	***MS_DECEL***	With the MotionStop (ms) command, use the value from TRAC as braking acceleration considering the active interpolation units.
16 / 1 0000H to 23/80 0000H		Free for future use
24 / 0100 0000H	***SimulationMode***	With this bit, the control can be set into the simulation mode. In this mode no position sizes are given to the dirve systems, the profile of the actual position is simulated. **Caution:** A drift of the axes must be avoided from the user, as in this mode, the bearing controller is disabled.
25 / 0200 0000H	***OverMode***	When this bit is set, the Jog-Override of the selected axes will not be influenced when calling the commando ctru.
26 / 0400 0000H	***StopAtWriteln***	When this bit is set, the SAP command writeln causes to stop the corresponding task. In this case in the register “running” of the data structure CNCTS (Section 4.3.2.10) additionally Bit 2 is set. This mode can be used for the complete processing of output strings in an overlapping program.
27 / 0800 0000H	***ClearZeroPosition***	When this bit is set, the zero offset set with szpa / szpr is deleted. However, the current position values remain the same. When the bit is set, the reference position can thus be moved at any time, for example.
28 / 1000 0000H	***JSatSAF***	JOG Stop at Spooler-Asynchronous Flag: If this bit is set, all axes are stopped with the programmed Stop deceleration when an SAF flag appears in the AXST register. In case of error, also bit 19 is set in the ErrorReg.
29 / 2000 0000H	***InhibitProfile Refuse***	Usually, interpolation positioning profiles without traverse distance or with velocity/acceleration = 0 are automatically rejected and an error message in the fwsetup monitor screen is generated. Using this bit, the output of an error message during the rejection of positioning profiles can be disabled.
Following bits		Currently not assigned, reserved for future use

6.3.2 Axis specifiers

The various axis channels are referenced with a symbolic name. You can choose these names quite freely in the *mcfg.exe* program. In the *rw_SymPas* programming language, these names are predefined automatically and serve in the user program as parameters for various commands. Remember that the *NCC* compiler distinguishes between upper and lower case for the axis specifiers.

6.3.3 Axis qualifiers

The system parameters listed below are used as axis qualifiers and are therefore available for all the axis channels in the system and thus for all axis specifiers. You can use these parameters to interrogate or set various axis-specific data. Remember that the *NCC* compiler distinguishes between upper and lower case for these parameters. An axis qualifier is referenced by stating an axis specifier, the character "." and the axis qualifier. The example below illustrates this.

```
...
var
        input: integer;
...
input := A2.digi;               // Read in digital inputs from
                                // axis channel 2
...
```

Table 37: Axis qualifiers

Name	Type	Abbr. meaning	Function
an	integer	axis number	The axis qualifier an contains the axis number of the axis designator involved. The qualifier can be used in relation with „variable“ axis name. [Chapter 5.3.3.1]
aux	double	Auxiliary Register	The content of this register is dependent from the option. If the system includes the optionEV (Encoder Verification), the encoder count by stepper motor systems can be accessed through this variable. In this case the unit of the register is Counts.
axst	integer	axis status	Error, state and profile flags (wordwise)
digi	integer	digital inputs	Digital inputs of the APCI-800x (wordwise) Various flags of this register can be erased by assigning any desired value to this register [chapter 4.4.51.1].
digo	integer	digital outputs	Digital outputs of the APCI-800x (wordwise)
dp	double	desired position	Setpoint position of the axis channel
dpoffset	double	desired position offset	In this register, a position offset for the position controller can be entered in the axis-specific user unit. This register can be used for a cascade control e.g. by steppers with encoder verification. This register is available for stepper systems from the version 2.5.2.23 of RWMOS, for servo systems from the version 2.5.2.29 of RWMOS.
dv	double	desired velocity	Setpoint velocity of the axis channel
dvoffset	double	desired velocity offset	In this register, a velocity offset of the position offset (dpoffset) for the position controller can be entered in the axis-specific user unit.
effradius	double	Effektiv Radius	When rotary axes are involved in translatory interpolation travels: axis-specific parameter for conversion of rotatory values in tanslatory ones, (Surface area processing) [Chapter 2.3.4]
epc	integer	EEPROM programming cycles	Number of programming cycles
gcr	integer	gear configuration register	With this register, the Gear functionality of the APCI-8001 can be controlled. This register is also described in the manual “Resource Interface”.

Name	Type	Abbr. meaning	Function
gf	double	gear factor	The axis-specific gear factor can be accessed using this variable. An assignment to this value may only be made in special cases.
ifs	integer	interface status	Interface status flags of the APCI-800x (wordwise) Various flags of this register can be erased by assigning any desired value to this register [see chapter 4.4.69.1].
hac	double	home acceleration	Acceleration for *home* commands
hvl	double	home velocity	Velocity for *home* commands
ipw	double	In position window	Position-dependent target window
jac	double	jog acceleration	Acceleration for *jog* commands
jerkrel	double	Jerk Relativ	Parameter for S velocity profile
jovr	double	jog override	Velocity factor
jtvl	double	jog target velocity	Target velocity for *jog* commands
jvl	double	jog velocity	Velocity for *jog* commands
kd	double		PIDF filter coefficient for differentiation
kfca	double		PIDF filter coefficient for forward compensation for acceleration
kfcv	double		PIDF filter coefficient for forward compensation for velocity
ki	double		PIDF filter coefficient for integration
kp	double		PIDF filter coefficient for amplification
kpl	double		PIDF filter coefficient for add. phase lead
lp	double	latched position	latched position value
lpndx	double	latched position index	latched position value with index signal (zero track)
lsm	integer	left spool memory	free spool area [Bytes]
maxacc	double	maximum acceleration	Axis-specific maximum acceleration in ChkMaxVel mode
maxvel	double	maximum velocity	Axis-specific maximum velocity in ChkMaxVel mode
mcis	integer	Move Commands in Spooler	This register shows how many traverse commands are currently included in the spooler. Thus the processing state of the spooler is checked. This information can be used, when the current process must be continued after interruption (see also PCAP command rdMCiS).
mcp	integer	Motor Command Port	Servomotors: Setpoint value for anologue port stepper motors: Stepper signal for stepper motor perfomance end levels. Additional description of the commands: wrmcp (chapter 4.4.152) and rdmcp (chapter 4.4.87).
mdvel	double	maximum velocity skip	Axis-specific maximum velocity jump in Look-ahead mode
mpe	double	maximum position error	Maximum permitted position error
poserr	double	position error	The current axis-specific position error in the user unit is shown in this register. This is the value dp – rp calculated in real-time.
pprev	double	Pulses per Revolution	The number of encoder pulses per revolution (drive side) can be read in this register. For stepper and linear axes or if the denominator unit of slsp is a linear unit, 0 is returned here. Compared to slsp, a possible pulse quadruplication is taken into account here.
rp	double	real position	Actual position of the axis channel
rv	double	real velocity	Actual velocity of the axis channel [Chapter 4.4.97], can only be read not assigned
sdec	double	stop deceleration	Stop deceleration of the axis channel
sf	integer	special function	Application-specific register.

Name	Type	Abbr. meaning	Function
sll	double	Software limit left	Left software limit
slr	double	Software limit right	Right software limit
slsp	double	Slits or stepper Pulses	In this register, the number of encoder slits per turn (drive side) or the number of steps per turn at stepper motors can be read or set. Quadruplication and units correspond with the values set in mcfg.
tp	double	target position	Target position of axis channel
zerooffset	double	Zero-Offset	Lately set zero point switch

The function of these qualifiers can be found at the relevant *rdxxxx()* and *wrxxxx()* commands in the function reference list for PCAP programming. The significance of the qualifier *digo*, for example, is explained under the *wrdigo()* command.

Exception: The PIDF filter coefficients become operative together with the SAP command *UF()*. These coefficients are read and written on PCAP level using the *rdf()* and *uf()* commands.

6.3.4 Structured axis qualifiers

The system parameters listed below are used as structured axis qualifiers and are therefore available for all the axis channels in the system and thus for all axis specifiers. You can use these parameters for bitwise interrogation and setting of various axis-specific data. Remember that the *NCC* compiler distinguishes between upper and lower case for these parameters. Referencing to a structured axis qualifier is illustrated by the example below:

```
...
const
        enable = 1;
var
        input: boolean;
...
input := A2.digib.enable; // read digital input 1 of axis
                                             // channel 2 (I1)
A1.digob.7 := TRUE;       // Set digital output 7 (O7)
...
```

Table 38: Structured axis qualifiers

Name	Type	Abbr. meaning	Function
digib	boolean	digital-input-bit	Digital inputs of the APCI-800x (bitwise)
digob	boolean	digital-output-bit	Digital outputs of the APCI-800x (bitwise)
ifsb	boolean	interface-status-bit	Status flags of the APCI-800x (bitwise)
axstb	boolean	axis status-bit	Error, status and profile flags (bitwise)

The function of these qualifiers can be found at the relevant *rdxxxxb()* and *wrxxxxb()* commands in the function reference list for PCAP programming. The significance of the qualifier *digib*, for example, is explained in the *rddigib()* command.

Note: Bit counting for the structured axis qualifiers begins at 1!

6.3.5 Abbreviations

Some of the abbreviations used in the function reference list will be explained to start with:

Table 39: Abbreviations

Name	Description
A1	Symbolic name for the first axis channel. This name can be freely selected in mcfg.exe. Is mainly used for examples
A2	Symbolic name for the second axis channel. Otherwise as for A1.
Spec	Axis specifier, such as A1 or A2
Qual	Axis qualifier, such as digi, digib, digo, digob, axst etc.
Pos	Position setpoint value (data type: double)
Event	Procedure with function as event handler

6.4 Reserved procedure names with event function

rw_SymPas incorporates a series of predefined procedure names with event function. If there are procedure definitions with these procedure names in the user program, the CNC task can be made by means of an enable command to call these procedures automatically if a procedure-specific event occurs. These procedures are accordingly also referred to as "event handlers".

Note: The events are checked after every execution of an rw_SymPas statement. Here it must be observed that the respective events are not checke anymore, if a task is terminated. If a continuous event monitoring is necessary, the respective task must stay in a an endless loop.

6.4.1 Event procedure EVEO

The EVEO event procedure is processed automatically after the definition of the procedure EVEO and the release of the corresponding event. The EO (Emergency Out) event occurs when a digital input planned with EO function is activated (see MCFG / Chapter 1.7.2.5). If the system includes more than one EO inputs, the *axst* status register can be used to check which EO input is causing the error concerned. A simple example program for implementing an EO-handler is listed below:

```
...
procedure EVEO;     // predefined name for
                    // Timeout EVENT hHandler
begin
     CI0 := 999;    // Common Variable
                    // signals program abort
     abort;         // Abort application program
end;

...
begin
     ...
     CI0 := 0;      // Delete common
                    // Variable
     enev(EVEO);    // Enable timeout handler
     ...
end.
```

6.4.2 Event procedure EVDNR

The EVDNR event procedure also operates like EVEO, except that this procedure is processed automatically when the Drive Not Ready event occurs. The DNR event occurs when a digital input planned with DR function becomes inactive (MCFG / Chapter 1.7.2.5).

6.4.3 Event procedure EVLSH

The EVLSH event procedure also operates like EO, except that this procedure is processed automatically when the Limit Switch Hardware event occurs. The LSH event occurs when a digital input planned with LSL_SMD, LSL_TOM, LSL_SMA, LSL_SMD, LSR_TOM, LSR_SMA or LSR_SMD function is activated (MCFG / Chapter 1.7.2.5).

6.4.4 Event procedure EVLSS

The EVLSS event procedure also operates like EVEO, except that this procedure is processed automatically when the Limit Switch Software (software limit) event occurs. The LSS event occurs when the current position of an axis system exceeds a limit value specified in the TOOLSET program *mcfg.exe* and the limit value concerned has been planned with the TOM, SMA or SMD function (MCFG / Chapter 1.7.2.5).

6.4.5 Event procedure EVMPE

The EVMPE event procedure also operates like EVEO, except that this procedure is processed automatically when the Maximum Position Error event occurs. The MPE event occurs when the control loop is closed and the difference between setpoint and actual positions of an axis system exceeds the limit value specified in the TOOLSET program *mcfg.exe* (MCFG / Chapter 1.7.2.1.9)

6.4.6 Event procedure EVUI

The EVUI event procedure also operates like EVEO, except that this procedure is processed automatically when the User Input event occurs. The UI event occurs when a digital input planned with UI is activated (MCFG / Chapter 1.7.2.5). You have an option for building up user-specific special functions with UI-planned digital inputs in the SAP program. Alternative cyclical polling can be dispensed with.

6.4.7 Priority and processing sequence for the event procedures

It is possible that different events will occur at the same point in time. In this case, the following priorities apply:

Procedure name	Priority
EVEO	highest priority
EVDNR	↓
EVLSH	
EVLSS	
EVMPE	
EVUI	lowest priority

If one event procedure (Event 1) is currently being processed, the occurrence of another event (Event 2) with lower or higher priority will be ignored; this event will not be executed until the current event handler (Event 1) has been processed. But Event 2 must still be active then!

Note: After the *STOP* and *ABORT* SAP commands and during execution of the *WT()* SAP command, no event handlers will be processed!

6.5 SAP block commands

The command reference list provided below contains a series of commands which can be used to achieve a block-oriented program structure. All these commands have names which end with the character "W". Examples include the SAP commands *MLAW()*, *JAW()* or *SSMSW().* These commands automatically wait for the profile end of all axes involved, i.e. the next statement will not be processed until the target positions of the selected axes have been reached. For this purpose, the CNC task polls the profile end flags of these axes and continues the program at the next statement when appropriate. This check routine takes the above-enabled EVENT handlers into account and processes them automatically when and as required.

Note: Another option for profile end checking is to evaluate the *axst* axis qualifier.

6.6 *rw_SymPas* SAP command reference list

6.6.1 Structure of the reference list

The reference list is structured as follows:

ABBREVIATION MEANING, DESCRIPTION	This is the name which is used to call the function subsequently described. Here you will find a detailed description of the function name concerned.
FUNCTION PARAMETERS:	If the function demands a parameter transfer, these are listed here.
SYSTEM PARAMETERS:	Various functions are executed by taking various system parameters into account. These are listed here.
SIMULTANEOUS FUNCTION:	With various functions, it is permitted to specify one or more axes for which the function concerned is to be executed.
REFERENCES:	Refers to other functions and chapters.
DECLARATION:	The formal declaration of predefined system functions; user-defined elements are shown in italics.
RESULT TYPE:	The type of the value returned (with system functions only).
DESCRIPTION:	Plaintext description of the command concerned.
NOTE:	Recurrent notes and explanations here indicate the chapters you should consult.
EXAMPLE:	An example of the function involved.

6.6.2 ABORT, abort

DESCRIPTION:	This command causes a running SAP program to be aborted. In contrast to the STOP statement, the program cannot be continued with the PCAP command *contcnct()* or the SAP command *CONTCNCT()*. This is possible only with the PCAP command *startcnct()* or the PCAP command *STARTCNCT()*.
NOTE:	After the command has been executed, the enabled EVENT handler procedures will no longer be processed.
EXAMPLE:	*ABORT;*

6.6.3 ABS, absolute function

DESCRIPTION:	The function returns the absolute value of *value*.
DECLARATION:	abs(*value*:double)
RESULT TYPE:	double

EXAMPLE:

```
...
var
            d1, d2: double;
...
d1 := -5.0;
d2 := ABS(d1);              // d2 := 5.0
```

6.6.4 ACOS, arc cosine function

DESCRIPTION:	The function returns the arc cosine of *value*. The argument *Value* must lie within the range [-1..+1]. The return value has the unit rad and lies within the limits [0..pi].
DECLARATION:	acos(*value*:double)
RESULT TYPE:	double

6.6.5 ASIN, arc sine function

DESCRIPTION:	The function returns the arc sine of *value*. The argument *Value* must lie within the range [-1..+1]. The return value has the unit rad and lies within the limits [-pi/2..+pi/2].
DECLARATION:	asin(*value*:double)
RESULT TYPE:	Double

6.6.6 ATAN, arc tangent function

DESCRIPTION:	The function returns the arc tangent of *value*. The return value has the unit rad and lies within the limits [-pi/2..+pi/2].
DECLARATION:	atan(*value*:double)
RESULT TYPE:	Double

6.6.7 AZO, activate zero offsets

DESCRIPTION:	PCAP command *azo()*
FUNCTION PARAMETERS:	Integer constant in the value range of 0..4

EXAMPLE:

```
const Offsets1 = 1;

azo(Offsets1);   // Activate zero offsets Set 1
```

6.6.8 CL, close loop

DESCRIPTION:	PCAP command *cl()* [chapter 4.4.6]
FUNCTION PARAMETERS:	*Spec*
SIMULTANEOUS FUNCTION:	Yes
EXAMPLE:	*CL(A1, A2); // Bring Axis Channels 1 and 2 into position control*

6.6.9 CLV

DESCRIPTION:	PCAP command *clv()* [Kapitel 4.4.9]
FUNCTION PARAMETERS:	*Spec*
SIMULTANEOUS FUNCTION:	Ja

EXAMPLE:	*clv (A1, A2); // Bring axis channels 1 and 2 into position contro*
	js (A1, A2); // then stop the axes immediately

6.6.10 CONTCNCT, continue CNC-Task

DESCRIPTION:	This command continues the CNC task transferred in the parameter.
FUNCTION PARAMETERS:	Integer constant in the range of 0..3
NOTE:	The command can be used to continue a stopped SAP program. An SAP program which has been stopped with the SAP command *ABORT* can only be restarted (i.e. not continued) with the SAP command *STARTCNCT()* or the PCAP command *startcnct()*. Automatic continuation of stopped tasks is not possible either.
EXAMPLE:	

```
...
const
        TASK0 = 0;
...
CONTCNCT(TASK0);            // continue Task 0
CONTCNCT(1);                // continue Task 1
```

6.6.11 COS, cosine function

DESCRIPTION:	The function returns the cosine of *value*. The argument *Value* is interpreted as an angle in the unit rad (0..2Pi = 0..360) degrees.
DECLARATION:	cos(*value*:double)
RESULT TYPE:	Double
NOTE:	*Sin()*, *Tan()*-function
EXAMPLE:	

```
...
var
        d1, d2: double;
...
d1 := 3.1415;
d2 := COS(d1); // d2 := -1.0 (rounded)
```

6.6.12 COSH, hyperbolic cosine function

DESCRIPTION:	The function returns the hyperbolic cosine of *value*.
DECLARATION:	cos(*value*:double)
RESULT TYPE:	Double

6.6.13 DISEV, disable event

DESCRIPTION:	disables the event handler specified
FUNCTION PARAMETERS:	*Event*
REFERENCES:	Chapter 0 and SAP command *ENEV()*
EXAMPLE:	*DISEV(EVEO); // ignore emergency out handler*

6.6.14 ENEV, enable event

DESCRIPTION:	enables the event handler specified.
FUNCTION PARAMETERS:	*Event*
REFERENCES:	Chapter 0 and SAP command *DISEV()*
EXAMPLE:	*ENEV(EVEO); // enable emergency out handler*
NOTE:	The released event-handler is not active anymore if the task is terminated or stopped.

6.6.15 EXP, exponential function

DESCRIPTION:	The function returns the value e^{value}, where *e* is the base of the natural logarithm (2.718281...).
DECLARATION:	exp(*value*:double)
RESULT TYPE:	Double
NOTE:	Function *Ln()*

6.6.16 JA, jog absolute

DESCRIPTION:	The axis channel(s) selected is/are moved absolutely to the position setpoints specified. For this purpose, the motor is accelerated with the axis-specific acceleration *jac* to the velocity *jvl and* moved to the specified target position *Pos*. In addition, you can use the *jtl* parameter to specify a target velocity. The trajectory parameters are specified in the axis-specific units.
FUNCTION PARAMETERS:	*Spec* and *Pos*
SYSTEM PARAMETERS:	Qualifier: *jac, jvl* and *jtvl*
SIMULTANEOUS FUNCTION:	Yes
REFERENCES:	PCAP command *ja()*, SAP command *JAW()*
NOTE:	PCAP command *ja()*

EXAMPLE:	*JA(A1:=100.0); // Move Axis 1 absolutely to position 100*
	JA(A1:=100.0, A2:=100.0);

6.6.17 JAW, jog absolute waiting

DESCRIPTION:	This command is identical to the SAP command *JA()* and PCAP command *ja(),* except that the system also waits for the profile end of all axes involved. The use of this command gives the SAP program a block-like form of the kind found in commercially available CNC controls.
FUNCTION PARAMETERS:	*Spec, Po*s
SYSTEM PARAMETERS:	Qualifiers: *jac, jvl* and *jtvl*
SIMULTANEOUS FUNCTION:	Yes
REFERENCES:	*JA*
NOTE:	You should use EVENT handlers to ensure that the drive is operated properly even in exceptional situations, since the CNC program dwells concomitantly long on this command, particularly when very time-consuming positioning operations are involved.

EXAMPLE:	*JAW(A2:=-1000.0); // Move Axis 1 absolutely to Position -1000.0 and // wait until the profile end is reached*
	JAW(A1:=1e3, A2 := 1.3e4);

6.6.18 JHI, jog home index

DESCRIPTION:	The reference search run for the zero track (index) of the rotary transducer or the linear scale for all selected axis channels is started. The search run will be aborted if the traverse distance or angle specified in *Pos* is exceeded.
FUNCTION PARAMETERS:	*Spec, Pos*
SYSTEM PARAMETERS:	Qualifiers: *hac* and *hvl*
SIMULTANEOUS FUNCTION:	Yes
REFERENCES:	PCAP command *jhi()*, SAP command *JHIW()*
EXAMPLE:	*JHI(A1 := 1.0, A2 := 1.5); // Start reference search run for axes 1 and 2.*

6.6.19 JHIW, jog home index waiting

DESCRIPTION:	This command is identical to PCAP command *jhi()* and SAP command *JHI()*. In addition, the system waits for the profile end of the axes involved.
FUNCTION PARAMETERS:	*Spec*
SYSTEM PARAMETERS:	Qualifiers: *hac* and *hvl*
SIMULTANEOUS FUNCTION:	Yes
NOTE:	SAP command *JA()*
EXAMPLE:	*JHIW(A1 := 5.0);*

6.6.20 JHL, jog home left

DESCRIPTION:	The reference search run on a digital input planned with REF for all selected axis channels is started towards the left traversing direction.
FUNCTION PARAMETERS:	*Spec*
SYSTEM PARAMETERS:	Qualifiers: *hac* and *hvl*
SIMULTANEOUS FUNCTION:	Yes
REFERENCES:	PCAP command *jhl()*, SAP command *JHLW()*
EXAMPLE:	*JHL(A1);*

6.6.21 JHLW, jog home left waiting

DESCRIPTION:	This command is identical to the PCAP command *jhl()* and SAP command *JHL()*. In addition, the system waits for the profile end of the axes involved.
FUNCTION PARAMETERS:	*Spec*
SYSTEM PARAMETERS:	Qualifiers: *hac* and *hvl*
SIMULTANEOUS FUNCTION:	Yes
REFERENCES:	PCAP command *jhl()*, SAP command *JHL()*
NOTE:	SAP command *JA()*
EXAMPLE:	*JHLW(A2);*

6.6.22 JHR, jog home right

DESCRIPTION:	The reference search run on a digital input planned with REF for all selected axis channels is started towards the right traversing direction.
FUNCTION PARAMETERS:	*Spec*
SYSTEM PARAMETERS:	Qualifiers: *hac* and *hvl*
SIMULTANEOUS FUNCTION:	Yes
REFERENCES:	PCAP command *jhr()*, SAP command *JHRW()*
EXAMPLE:	*JHR(A2);*

6.6.23 JHRW, jog home right waiting

DESCRIPTION:	This command is identical to the PCAP command *jhr()* and SAP command *JHR()*. In addition, the system waits for the profile end of the axes involved.
FUNCTION PARAMETERS:	*Spec*
SYSTEM PARAMETERS:	Qualifiers: *hac* and *hvl*
SIMULTANEOUS FUNCTION:	Yes
NOTE:	SAP command *JA()*
EXAMPLE:	*JHRW(A1);*

6.6.24 JR, jog relative

DESCRIPTION:	For description, please consult PCAP command *jr()*.
FUNCTION PARAMETERS:	*Spec, Pos*
SYSTEM PARAMETERS:	Qualifiers: *jac, jvl* and *jtvl*
SIMULTANEOUS FUNCTION:	Yes
EXAMPLE:	*JR(A1 := 100);*

6.6.25 JRW, jog relative waiting

DESCRIPTION:	This command is identical to the PCAP command *jr()* and the SAP command *JR()*. In addition, the system waits for the profile end of the axes involved.
FUNCTION PARAMETERS:	*Spec, Po*s
SYSTEM PARAMETERS:	Qualifiers: *jac, jvl* and *jtvl*
SIMULTANEOUS FUNCTION:	Yes
REFERENCES:	JR

6.6.26 JS, jog stop

DESCRIPTION:	For description, please consult PCAP command *js()*.
FUNCTION PARAMETERS:	*Spec*
SYSTEM PARAMETERS:	Qualifiers: *sdec*
SIMULTANEOUS FUNCTION:	Yes
EXAMPLE:	*JS(A1);*

6.6.27 JSW, jog stop waiting

DESCRIPTION:	This command is identical to the PCAP command *js()* and the SAP command *JS()*. In addition, the system waits for the profile end of the axes involved.
FUNCTION PARAMETERS:	*Spec*
SYSTEM PARAMETERS:	Qualifiers: *sdec*
SIMULTANEOUS FUNCTION:	Yes
EXAMPLE:	*JSW(A1);*

6.6.28 LN, natural logarithm function

DESCRIPTION:	The function returns the natural logarithm of *value*, i.e. the power by which the constant 2.71828... must be raised to obtain *value*.
DECLARATION:	ln(*value*:double)
RESULT TYPE:	Double
NOTE:	Values smaller than/equal to 0.0 for *value* are not defined mathematically. In this case the function has no valid return value. Function *Exp()*

6.6.29 LPR, latch position registers

DESCRIPTION:	Start the data recording of an motion process for one axis (see graphical system analysis in mcfg).
FUNCTION PARAMETERS:	*Spec*
SYSTEMPARAMETER:	PU, TU, LST, LET
SIMULTANEOUS FUNCTION:	No
EXAMPLE:	LPR (A1);

6.6.30 LPRS, latch position registers synchronous

DESCRIPTION:	Start the synchronous data recording of an motion process for several axes (see graphical system analysis in mcfg).
FUNCTION PARAMETERS:	*Spec*
SYSTEMPARAMETER:	PU, TU, LST, LET
SIMULTANEOUS FUNCTION:	Yes
EXAMPLE:	LPRS (A1, A2, A3);

6.6.31 MCA, move circular absolute - SMCA, spool motion circular absolute

DESCRIPTION:	PCAP command *mca()*, *smca()*
FUNCTION PARAMETERS:	*Spec*, *Pos*
SYSTEM PARAMETERS:	*TRAC, TRVL, TRTVL, PHI*

EXAMPLE:	*MCA(A1 := 50.0, A2 := 0.0, PHI := 720.0);*
	SMCA(A1 := 0.0, A2 := 10.0, PHI := 0.1);

6.6.32 MCAW, move circular absolute waiting

DESCRIPTION:	This command is identical to the SAP command *MCA()*, except that here the system also waits for the profile end of the two axes involved.
FUNCTION PARAMETERS:	*Spec*, *Pos*
SYSTEM PARAMETERS:	*TRAC, TRVL, TRTVL, PHI*
REFERENCES:	PCAP command *mca()*

6.6.33 MCA3D, move circular absolute three-dimensional SMCA3D, spool move circular absolute three-dimensional

DESCRIPTION:	PCAP command *mca3d()*, *smca3d()*
FUNCTION PARAMETERS:	*Spec*, *Pos*
SYSTEM PARAMETERS:	*TRAC, TRVL, TRTVL, PHI, PN1, PN2, PN3*
EXAMPLE:	*MCA3D(A1 := 50.0, A2 := 0.0, A3 := 0.0, PN1 = 1.0, PN2 =0. 0, PN3 = 1.0, PHI := 720.0); // Circle rotated by 45 degrees around A2*

6.6.34 MCA3DW, move circular absolute three-dimensional waiting

DESCRIPTION:	This command is identical to the SAP command *MCA3D()*,except that here the system also waits for the profile end of the two axes involved.
FUNCTION PARAMETERS:	*Spec*, *Pos*
SYSTEM PARAMETERS:	*TRAC, TRVL, TRTVL, PHI, PN1, PN2, PN3*
REFERENCES:	SAP command *mca3d()*

6.6.35 MCR3D, move circular relative three-dimensional SMCR3D, spool move circular relative three-dimensional

DESCRIPTION:	PCAP command *mcr3d()*, *smcr3d()*
FUNCTION PARAMETERS:	*Spec*, *Pos*
SYSTEM PARAMETERS:	*TRAC, TRVL, TRTVL, PHI, PN1, PN2, PN3*
EXAMPLE:	*MCR3D(A1 := 50.0, A2 := 0.0, A3 := 0.0, PN1 = 1.0, PN2 =0. 0, PN3 = 1.0, PHI := 720.0); // Circle rotated by 45 degrees around A2*

6.6.36 MCR3DW, move circular relative three-dimensional waiting

DESCRIPTION:	This command is identical to the SAP command *MCR3D()*,except that here the system also waits for the profile end of the two axes involved.
FUNCTION PARAMETERS:	*Spec*, *Pos*
SYSTEM PARAMETERS:	*TRAC, TRVL, TRTVL, PHI, PN1, PN2, PN3*
REFERENCES:	SAP command *mcr3d()*

6.6.37 MCR, move circular relative - SMCR, spool motion circular relative

DESCRIPTION:	PCAP command *mcr()*, *smcr()*
FUNCTION PARAMETERS:	*Spec*, *Pos*
SYSTEM PARAMETERS:	*TRAC, TRVL, TRTVL, PHI*

EXAMPLE:	*MCR(A1 := 50.0, A2 := 0.0, PHI := 360.0);*
	SMCR(A1 := 0.0, A2 := 10.0, PHI := 45.0);

6.6.38 MCRW, move circular relative waiting

DESCRIPTION:	This command is identical to the SAP command *MCR()*, except that here the system also waits for the profile end of the two axes involved.
FUNCTION PARAMETERS:	*Spec*, *Pos*
SYSTEM PARAMETERS:	*TRAC, TRVL, TRTVL, PHI*

6.6.39 MHA, move helical absolute - SMHA, spool motion helical absolute

DESCRIPTION:	PCAP command *mha()*, *smha()* *If the circle is to be defined by the target point, the target coordinates are to be allocated to the axis specifiers and the center point coordinates to the system parameters DTCA1 and DTCA2. If the circle is specified by the traverse angle, the center point coordinates are allocated to the axis specifiers of the circle axes.*
FUNCTION PARAMETERS:	*Spec*, *Pos*
SYSTEM PARAMETERS:	*TRAC, TRVL, TRTVL, PHI, (ggf. DTCA1, DTCA2)*

EXAMPLE:	*MHA(A1 := 50.0, A2 := 0.0, PHI := 720.0, A3 := 10);*
	SMHA(A1 := 0.0, A2 := 10.0, PHI := 0.1, A3 := 10);
	// Circle in anticlockwise direction with Radius 10
	MHR(A1 := 0.0, A2 := 0.0, PHI := 0.0, A3 := 10, DTCA1 :=-10, DTCA2 := 0);
	// Semi-circle in clockwise direction with Radius 10
	SMHR(A1 := 20.0, A2 := 0.0, PHI := -1e-100, A3 := 10, DTCA1 := 10, DTCA2 := 0);

6.6.40 MHAW, move helical absolute waiting

DESCRIPTION:	This command is identical to the SAP command *MHA()*, except that here the system also waits for the profile end of all axes involved.
FUNCTION PARAMETERS:	*Spec*, *Pos*
SYSTEM PARAMETERS:	*TRAC, TRVL, TRTVL, PHI, (ggf. DTCA1, DTCA2)*

6.6.41 MHR, move helical relative - SMHR, spool motion helical relative

DESCRIPTION:	PCAP command *mhr()*, *smhr()* Here the target point cannot be specified to run the circle.
FUNCTION PARAMETERS:	*Spec*, *Pos*
SYSTEM PARAMETERS:	*TRAC, TRVL, TRTVL, PHI, (ggf. DTCA1, DTCA2)*

6.6.42 MHRW, move helical relative waiting

DESCRIPTION:	This command is identical to the SAP command SAP command *MHR()*, except that here the system also waits for the profile end of all axes involved.
FUNCTION PARAMETERS:	*Spec*, *Pos*
SYSTEM PARAMETERS:	*TRAC, TRVL, TRTVL, PHI, (if necessary DTCA1, DTCA2)*

6.6.43 MLA, move linear absolute - SMLA, spool motion linear absolute

DESCRIPTION:	description is provided at the PCAP commands *mla()* or *smla()*.
FUNCTION PARAMETERS:	*Spec*, *Pos*
SYSTEM PARAMETERS:	*TRAC, TRVL, TRTVL*
SIMULTANEOUS FUNCTION:	Yes

EXAMPLE:	*MLA(A1:=1000.0, A2:=3.2e2);*
	SMLA(A1:=100.0, A2:=-335.0);

6.6.44 MLAW, move linear absolute waiting

DESCRIPTION:	This command is identical to the SAP command *MLA()*, except that here the system also waits for the profile end of the axes involved.
FUNCTION PARAMETERS:	*Spec*, *Pos*
SYSTEM PARAMETERS:	*TRAC, TRVL, TRTVL*
SIMULTANEOUS FUNCTION:	Yes
EXAMPLE:	*MLAW(A1:=-0.3e3, A2:=100.4);*

6.6.45 MLR, move linear relative - SMLR, spool motion linear relative

DESCRIPTION:	The description is provided at the PCAP commands *mlr()* or *smlr()*.
FUNCTION PARAMETERS:	*Spec*, *Pos*
SYSTEM PARAMETERS:	*TRAC, TRVL, TRTVL*
SIMULTANEOUS FUNCTION:	Yes

EXAMPLE:	*MLR(A1:=2000.0, A2:=3.2e2);*
	SMLR(A1:=300.0, A2:=-35.3);

6.6.46 MLRW, move linear relative waiting

DESCRIPTION:	This command is identical to the SAP command *MLRW()*, except that here the system also waits for the profile end of the axes involved.
FUNCTION PARAMETERS:	*Spec*, *Pos*
SYSTEM PARAMETERS:	*TRAC, TRVL, TRTVL*
SIMULTANEOUS FUNCTION:	Yes
EXAMPLE:	*MLRW(A1:=-3.45e3, A2:=100.4e-1);*

6.6.47 MS, motion stop

DESCRIPTION:	The description is provided at the PCAP command *ms()* [Chapter 4.4.39].
FUNCTION PARAMETERS:	*Spec*
SYSTEM PARAMETERS:	None
SIMULTANEOUS FUNCTION:	Yes
EXAMPLE:	*MS(A1, A2);*

6.6.48 MSW, motion stop waiting

DESCRIPTION:	This command is identical to the PCAP command *ms()* and SAP command *MS()*,except that here the system also waits for the profile end of the axes involved.
FUNCTION PARAMETERS:	*Spec*
SYSTEM PARAMETERS:	None
SIMULTANEOUS FUNCTION:	Yes
EXAMPLE:	*MSW(A1, A2);*

6.6.49 OL, open loop

DESCRIPTION:	PCAP command *ol()* [Chapter 4.4.41]
FUNCTION PARAMETERS:	*Spec*
SIMULTANEOUS FUNCTION:	Yes
EXAMPLE:	*OL(A1, A2); // Open position control loop of A1 and A2*

6.6.50 POWER

DESCRIPTION:	The function returns the value of *base* to the *exponent*.
DECLARATION:	sqrt(*base, exponent* : double)
RESULT TYPE:	double
NOTE:	Function is available from RWMOS.ELF V2.5.3.93

EXAMPLE:

```
...
var
        d1, d2: double;
...
d1 := 2.0;
d2 := 3.0;
d2 := POWER(d1, d2);  // d2 := 8.0
```

6.6.51 RA, reset axis

DESCRIPTION:	PCAP command *ra()*
FUNCTION PARAMETERS:	*Spec*
SIMULTANEOUS FUNCTION:	Yes
EXAMPLE:	*RA(A1, A2); // Reset axes A1 and A2*

6.6.52 RDCBD, read COMMON BUFFER double function

DESCRIPTION:	The function returns a floating-point value with double accuracy from the CNC-task-specific COMMON BUFFER. The *offset* parameter is a byte offset referenced to the first element (Element 0) of the COMMON BUFFER. The double data type occupies 8 bytes in the COMMON BUFFER. To enable the APCI-800x board CPU system to access this correctly, *offset* must always be word-oriented, i.e. have a value which is divisible by 8.
DECLARATION:	RDCBD(*offset*:integer)
RESULT TYPE:	double
NOTE:	The CNC-task-specific buffer size is <u>1,000 bytes</u>. PCAP commands *rdcbcnct() and wrcbcnct()*, SAP commands *RDCBx()* and *WRCBx()*
EXAMPLE:	*...* *var* *cbd: double;* *...* *cbd := RDCBD(500); // Read in double variable from offset 500*

6.6.53 RDCBI, read COMMON BUFFER integer function

DESCRIPTION:	The function returns an integer value from the CNC-task-specific COMMON BUFFER. The *offset* parameter is a byte offset referenced to the first element (Element 0) of the COMMON BUFFER. The integer data type occupies 4 bytes in the COMMON BUFFER. To enable the APCI-800x board CPU system to access this correctly, *offset* must always be word-oriented, i.e. have a value which is divisible by 4.
DECLARATION:	RDCBI(*offset*:integer)
RESULT TYPE:	integer
NOTE:	The CNC-task-specific buffer size is <u>1,000 bytes</u>. PCAP commands *rdcbcnct() and wrcbcnct()*, SAP commands *RDCBx()* and *WRCBx().*
EXAMPLE:	*...* *var* *cbi: integer;* *...* *cbi := RDCBI(500); // Read in integer variable from offset 500*

6.6.54 RDCBS, read COMMON BUFFER single function

DESCRIPTION:	The function returns a floating-point value with single accuracy from the CNC-task-specific COMMON BUFFER. The *offset* parameter is a byte offset referenced to the first element (Element 0) of the COMMON BUFFER. The single data type occupies 4 bytes in the COMMON BUFFER. To enable the APCI-800x board CPU system to access this correctly, *offset* must always be word-oriented, i.e. have a value which is divisible by 4.
DECLARATION:	RDCBS(*offset*:integer)
RESULT TYPE:	single
NOTE:	The CNC-task-specific buffer size is 1,000 bytes. PCAP commands *rdcbcnct() and wrcbcnct()*, SAP commands *RDCBx()* and *WRCBx()*
EXAMPLE:	*...* *var* *cbs: single;* *...* *cbs := RDCBS(500); // Read in single variable from offset 500*

6.6.55 RPTODP, Real-Position to Desired-Position

DESCRIPTION:	PCAP command *RPtoDP()*
NOTE:	The relevant axes must not be in a positioning profile, i.e. the profile end flag in the axis status register must be set.
EXAMPLE:	*RPTODP (X, Z);*

6.6.56 RS, reset system

DESCRIPTION:	PCAP command *rs()*
NOTE:	Once this command has been executed, no more monitoring can be performed by the stand-alone application program, since the CNC task is halted by this command.
EXAMPLE:	*RS*; *// reset complete axis system*

6.6.57 SHP, set home position

DESCRIPTION:	PCAP command *shp()*
FUNCTION PARAMETERS:	*Spec, Pos*
SIMULTANEOUS FUNCTION:	Yes
EXAMPLE:	*SHP(A2:=1000.0);*

6.6.58 SIN, sine function

DESCRIPTION:	The function returns the sine of *value*. The argument *Value* is interpreted as an angle in the unit rad (0..2Pi = 0..360) degrees.
DECLARATION:	sin(*value*:double)
RESULT TYPE:	double
NOTE:	*Cos()*, *Tan()* function
EXAMPLE:	

```
...
var
        d1, d2: double;
...
d1 := 3.1415;
d2 := SIN(d1);          // d2 := 0.0 (rounded)
```

6.6.59 SINH, hyperbolic sine function

DESCRIPTION:	The function returns the hyperbolic sine of *value*.
DECLARATION:	Cos(*value*:double)
RESULT TYPE:	double

6.6.60 SQR, square function

DESCRIPTION:	This function returns the square of value.
DEKLARATION:	sqr(*value*:double)
RESULT TYPE:	double
NOTE:	Available only in RWMOS and compiler versions from 05.10.2007
EXAMPLE:	

```
...
var
        d1, d2: double;
...
d1 := 9.0;
d2 := SQR(d1); // d2 := 81.0
```

6.6.61 SQRT, square root function

DESCRIPTION:	The function returns the square root of *value.*
DECLARATION:	sqrt(*value*:double)
RESULT TYPE:	double
NOTE:	Negative values of *value* are not defined mathematically. In this case, the function does not have a valid return value.
EXAMPLE:	

```
...
var
        d1, d2: double;
...
d1 := 9.0;
d2 := SQRT(d1);         // d2 := 3.0
```

6.6.62 SSF, Spool-Special-Function

DESCRIPTION:	This commands allows to enter other commands as traverse commands in the spooler. The command you want to execute is entered in the system variable *SSFP*. The value *Value* is entered as a parameter in the axis concerned.
FUNCTION PARAMETERS:	*Spec*, *Value*
SYSTEM PARAMETERS:	*SSFP*
SIMULTANEOUS FUNCTION:	Yes
COMMANDS:	See PCAP-command ssf in chapter 4.4.122

EXAMPLE:

```
SSF(A1:=999, SSFP = 1);                 // Write CI1 with  999
SSF(A1:=1, A4:=2, SSFP := 1001);        // Set O1 at axis 1 and O2 at axis 4
                                        //
SSF(A1:=0, A2:=0, A3:=0, SSFP:=1000); // Spooler halts at A1, A2 and A3
```

6.6.63 SSMS, start spooled motions synchronous

DESCRIPTION:	*Spool* commands can be used to transfer commands to the individual axis channels of the APCI-800x; they are entered in a queue. The *SSMS()* command causes a synchronized start for spooler command processing at all the axes specified in *AS.*
FUNCTION PARAMETERS:	*Spec*
SIMULTANEOUS FUNCTION:	Yes
REFERENCES:	PCAP command *ssms()*, SAP command *SSMSW()*

EXAMPLE:

```
...
SMLA(A1:=1000.0, A2:=1000.0);         // Spool traversing command
SMLR(A1:=200.0, A2:=500.0);           // Spool traversing command
...
SSMS(A1, A2);                         // Start spooler
```

6.6.64 SSMSW, start spooled motions synchronous waiting

DESCRIPTION:	Synchronized start of all axes selected and wait until all spooled motion profiles of these axes have been run completely and the profile end of all axes involved has been reached.
FUNCTION PARAMETERS:	*Spec*
SIMULTANEOUS FUNCTION:	Yes
REFERENCES:	SAP command *SSMS()*
NOTE:	SPOOL mode

EXAMPLE:

```
...
SMLR(A1:=1000.0, A2:=1000.0);         // Spool traversing command
SMLR(A1:=200.0; A2:=500.0);           // Spool traversing command
...
SSMSW(A1, A2);                        // Start spooler
```

6.6.65 STARTCNCT, start CNC-Task

DESCRIPTION:	This command starts the CNC task transferred in the parameter and executes the SAP program stored there from its beginning.
FUNCTION PARAMETERS:	Integer-constant in range 0..3
NOTE:	An SAP program can also start itself automatically from the beginning with this command.

EXAMPLE:

```
...
const
        Task1 = 1;
...
STARTCNCT(Task1);
```

6.6.66 STOP, stop

DESCRIPTION:	This command causes the currently running stand-alone application program to stop. In addition, the corresponding CNC task (Task 0, 1, 2, or 3) is put into idle state. The application program can be resumed by means of the *contcnct()*-PCAP command, the CONTCNCT()-SAP command or in the TOOLSET program *mcfg.exe*.
NOTE:	Any EVENT handling procedures enabled will no longer be processed after execution of the Stop command. The drive should therefore be put into a safe operating state before this command is executed.
EXAMPLE:	*STOP; // Stops the SAP program*

6.6.67 STEPCNCT, step CNC-Task

DESCRIPTION:	This command exectutes a programm line in the indicated CNC task.
FUNCTION PARAMETER:	Integer constant in the range 0..3
NOTE:	EVENT handling procedures that were possibly released, will not be processed anymore after executing the program line. Before the execution of the command, a valid program must be loaded. See also PCAP command *stepcnct* (chapter 4.4.125).

EXAMPLE:

```
...
const
        Task3 = 3;
...

STEPCNCT(Task3);
```

6.6.68 STOPCNCT, stop CNC-Task

DESCRIPTION:	This command halts the CNC task transferred in the parameter and thus halts the SAP program stored in it as well.
FUNCTION PARAMETERS:	Integer constant in the range 0..3
NOTE:	Any EVENT handling procedures enabled will no longer be processed by the correspondingly selected task after executing *STOPCNCT()*. See also chapter 6.6.66.
EXAMPLE:	*...* *const* *Task3 = 3;* *...* *STOPCNCT(Task3);*

6.6.69 STOPTOSS

DESCRIPTION:	This command transfers the CNC-task, which has been transferred in the parameter, from the stop-state to the step-state, however without executing a program line in the indicated task.
FUNCTION PARAMETERS:	Integer constant in the range 0..3
NOTE:	If the indicated task is not in the stop-mode, the command has no influence. This command is required especially for the single-step processing by using several SAP programming tasks.
EXAMPLE:	*...* *const* *Task3 = 3;* *...* *STOPTOSS(Task3);*

6.6.70 SZPA – Set Zero Position Absolut

DESCRIPTION:	Set a virtual zero position. The command is described at the PCAP command szpa (Chapter 4.4.127). You can use SZPA for stepper motor systems only from the version 2.5.2.32 of RWMOS.ELF.
FUNCTION PARAMETERS:	*Spec, Pos*
SIMULTANEOUS FUNCTION:	Yes
REFERENCES:	PCAP command *szpa()*, szpr(), SAP command *SZPR*
EXAMPLE:	SZPA (X := 100, Y := -20);

6.6.71 SZPR – Set Zero Position Relativ

DESCRIPTION:	Set the virtual zero position in a relative position. This command described at the PCAP command szpr (Chapter 4.4.128). You can use SZPR bei Schrittmotorsystemen for stepper motor systems only from the 2.5.2.32 of RWMOS.ELF.
FUNCTION PARAMETERS:	*Spec, Pos*
SIMULTANEOUS FUNCTION:	Yes
REFERENCES:	PCAP command *szpa()*, szpr(), SAP command *SZPA*
EXAMPLE:	SZPR (X := 100, Y := -20);

6.6.72 TAN, tangent function

DESCRIPTION:	The function returns the tangent of *value*. The argument *Value* is interpreted as an angle in the unit rad (0..2Pi = 0..360) degrees.
DECLARATION:	tan(*value*:double)
RESULT TYPE:	Double
NOTE:	*Sin()*, *Cos()* function
EXAMPLE:	

```
...
var
        d1, d2: double;
...
d1 := 0.5;
d2 := TAN(d1);  // d2 := 0.5463 (rounded)
```

6.6.73 TANH, hyperbolic tangent function

DESCRIPTION:	The function returns the hyperbolic tangent of *value*.
DECLARATION:	tan(*value*:double)
RESULT TYPE:	Double

6.6.74 UF, update filter

DESCRIPTION:	PCAP command *uf()*
FUNCTION PARAMETERS:	*Spec*
SYSTEM PARAMETERS:	Qualifiers: *kp, ki, kd, kpl, kfca, kfcv*
SIMULTANEOUS FUNCTION:	Yes
NOTE:	For updating the PIDF filter coefficients, all the qualifiers listed above must be initialized before executing the command.
EXAMPLE:	

```
...
A1.kp := 5.0;     // Alter proportional amplification
A1.ki := 0.0;
A1.kd := 0.0;
A1.kpl := 0.0;
A1.kfca := 0.0;
A1.kfcv := 0.0;
UF(A1);
...
```

6.6.75 UTROVR, update trajectory override

DESCRIPTION:	PCAP command *utrovr()*
FUNCTION PARAMETERS:	*Spec*
SYSTEM PARAMETERS:	*TROVR*
SIMULTANEOUS FUNCTION:	Yes

EXAMPLE:	*...*
	TROVR := 0.9; // Trajectory velocity override = -10%
	UTROVR(A1, A2); // Reduced trajectory velocity for axes A1 and A2
	...

6.6.76 WRCBI, write COMMON BUFFER integer procedure

DESCRIPTION:	The procedure describes a memory location of the integer type with the value of *value* in the CNC-task-specific COMMON BUFFER. The *offset* parameter is a byte offset referenced to the first element (Element 0) of the COMMON BUFFER. The integer data type occupies 4 bytes in the COMMON BUFFER. To enable the APCI-800x board CPU system to access this correctly, *offset* must always be word-oriented, i.e. have a value which is divisible by 4.
DECLARATION:	WRCBI(*offset*:integer; *value*:integer)
NOTE:	The CNC-task-specific buffer size is 1,000 bytes. PCAP commands *rdcbcnct() and wrcbcnct()*, SAP commands *RDCBx()* and *WRCBx()*
EXAMPLE:	*WRCBI(500, -1000); // Write integer variable from offset 500 // with value -1000*

6.6.77 WRCBS, write COMMON BUFFER single procedure

DESCRIPTION:	The procedure describes a memory location of the single type (floating-point number with single accuracy) with the value of *value* in the CNC-task-specific COMMON BUFFER. The *offset* parameter is a byte offset referenced to the first element (Element 0) of the COMMON BUFFER. The single data type occupies 4 bytes in the COMMON BUFFER. To enable the APCI-800x board CPU system to access this correctly, *offset* must always be word-oriented, i.e. have a value which is divisible by 4.
DECLARATION:	WRCBS(*offset*:integer; *value*:single)
NOTE:	The CNC-task-specific buffer size is 1,000 bytes. PCAP commands *rdcbcnct() and wrcbcnct()*, SAP commands *RDCBx()* and *WRCBx().*
EXAMPLE:	*WRCBS(500, 3.99); // Write single variable from offset 500 // with value 3.99*

6.6.78 WRCBD, write COMMON BUFFER double procedure

DESCRIPTION:	The procedure describes a memory location of the "double" type (floating-point number with double accuracy) with the value of *value* in the CNC-task-specific COMMON BUFFER. The *offset* parameter is a byte offset referenced to the first element (Element 0) of the COMMON BUFFER. The double data type occupies 8 bytes in the COMMON BUFFER. To enable the APCI-800x board CPU system to access this correctly, *offset* must always be word-oriented, i.e. have a value which is divisible by 8.
DECLARATION:	WRCBD(*offset*:integer; *value*:double)
NOTE:	The CNC-task-specific buffer size is 1,000 bytes. PCAP commands *rdcbcnct() and wrcbcnct()*, SAP commands *RDCBx()* and *WRCBx()*

EXAMPLE:	*WRCBD(500, 100.2e-128); // Write double variable from offset 500*
	// with value 100.2e-128

6.6.79 WRITE

DESCRIPTION:	Adding a partial string to the current task specific string output.
FUNCTION PARAMETER:	*Diverse*
NOTES:	The function can be called with an undefined number of parameters, which can be of the type string constant, integer, double or boolean. String constants are strings that are limited in rw_SymPas by superior commas and in the G-Code proramming by inverted commas. The single parameters are separated by commas. Numeric or boolean parameters can be also expressions. The call of this function sets Bit 0 in the system variable tskinfo. Information about the state of the string output see Chapter 4.4.13. The reading of the task specific output string is done with the PCAP function gettskstr(), see chapter 4.4.14.
EXAMPLE RW_SYMPAS:	write (’This is a string: ’, CI0); write (’Istposition: ’, A1.rp);
EXAMPLE G-CODES:	N0100 write “This a string“, CI0

6.6.80 WRITELN

DESCRIPTION:	Adding a partial string to the current task specific string output and concluding the output string.
FUNCTION PARAMETER:	*Diverse*
NOTE:	The function can be called with an undefined number of parameters, which can be of the type string constant, integer, double or boolean. String constants are strings that are limited in rw_SymPas by superior commas and in the G-Code proramming by inverted commas. The single parameters are separated by commas. Numeric or boolean parameters can be also expressions. If after this command write or writeln is called again, the previous output string will be overwritten. The call of this function sets Bit 1 in the system variable tskinfo. For information about the state of the string output, see chapter 4.4.13. The reading of the task specific output string is done with the PCAP function gettskstr(), see chapter 4.4.14. If Bit 26 is set in the MODEREG register (chapter 6.3.1.5), the respective CNC task will be stopped by this command.
EXAMPLE RW_SYMPAS:	writeln (ʼThis is a string: ʼ, CI0); writeln (ʼIstposition: ʼ, A1.rp);
EXAMPLE G-CODES:	N0100 writeln “This is a string“, CI0

6.6.81 WT, wait timer

DESCRIPTION:	Wait for the wait time transferred as a parameter before continuing the SAP program again. This command de-activates the CNC task and therefore does not need any CPU time. To reduce the workload on the master CPU system, this command may be used in queues, etc.
FUNCTION PARAMETERS:	Integer values with a unit of 64 µs
NOTE:	The EVENT handling procedures are not processed while this command is being executed. But if you want these to be monitored, this can, for example, be achieved by means of several *WT()* calls with shorter wait times (perhaps in a loop).

EXAMPLE:	*...*
	CONST sec = 15625;
	...
	*WT(5*sec); // Wait 5s*
	...

6.7 Compiler commands

As the name implies, a compiler command instructs the compiler, while it is compiling a source text, to execute (or not to execute) certain operations. In *rw_SymPas,* a compiler command is activated as follows:

Inside the SAP source text program, a special syntax is formulated inside a comment: The opening bracket ({) is followed directly by a dollar sign ($) and the name of the command, which consists of one or more letters. These "comments" can (apart from a few exceptions) appear at any position in the source text at which a normal comment would also be permissible.

6.7.1 Include file

DESCRIPTION:	This compiler command instructs the compiler to read in the file designated by *filename*. Basically, the compiler behaves as if the text read is in place of the {$I} command. *rw_SymPas* permits include files to be nested up to 15 levels. A file inserted by means of {$I} can thus itself insert further files, which in turn contain {$I} commands. **Note:** If in the *mcfg.exe* NCC editor environment an include file has already been opened in one of the three editor windows, the SAP source text of this editor will be incorporated and not the content of the file concerned.
SYNTAX:	{$I Filename }

6.7.2 Task selection

DESCRIPTION:	You can use this compiler command to specify the task (*TaskNr*, values 0..3) in which the SAP program involved is to be run. The information is stored in the autocode file *"filename.cnc"*. The PCAP command *txbf()* is used to transfer this file automatically into the right task.
SYNTAX:	{$TASK *TaskNr*}
NOTE:	If the SAP program concerned does not contain this statement, the task number currently selected will be utilized for compiling. But if the ($TASK) command is given, the correspondingly selected task number also becomes the default task number for all subsequent display, start and stop commands. Chapter 3.2.1

EXAMPLE:	*...*
	const
	Task1 = 1;
	...
	
	{$TASK Task1}; // or
	{$TASK 1}

6.7.3 Full system compiling

DESCRIPTION:	The command selects the compiler option FULLSYSTEM.
SYNTAX:	{$FULLSYSTEM}
NOTE:	If the SAP program concerned does not contain this statement, the option currently selected will be utilized for compiling. This statement is to be entered at the beginning of the source text file. This command is available for mcfg from the version V2.5.2.13 and for ncc from the version V2.5.2.9.

6.8 SAP runtime errors

When operating stand-alone programs, different errors may occur. In this case, the corresponding task is stopped. An error number and the number of the line where the error occurred are then entered in the data structure CNCTS (see chapter 4.3.2.10). The error numbers and possible causes of error are listed below.

Table 40: SAP runtime errors

Error #	Description
1 / 0001	The arithmetic operation is not correct for the used data type
2 / 0002	Incorrect data type
4 / 0004	Incorrect internal operation code. This can be caused by a compatibility problem between mcfg/ncc and RWMOS.ELF.
8 / 0008	Stack overflow. Program too big or internal problem for the RWMOS operating system software
16 / 0010	Stack underflow. This error can be an internal problem for the RWMOS operating system software.
32 / 0020	Unknown event handler. It can be be caused by a compatibility problem between mcfg/ncc and RWMOS.ELF.
64 / 0040	Incorrect NC command. This can be caused by a compatibility problem between mcfg/ncc and RWMOS.ELF.
128 / 0080	Address injury in NC program. This error can be caused by an internal problem by the RWMOS operating systemsoftware.
256 / 0100	Address injury in NC program through wrong parameter settings
512 / 0200	Error when using the AT interface
1024 / 0400	A common variable has been addressed outside the correct range.
2048 / 0800	Incorrect index in case of double access to the common buffer (cannot be divided by 8).
4096 / 1000	Incorrect SAP command. This error can be caused by a compatibility problem with controllers of another generation.
8192 / 2000	Error in cutting speed interpolation
16384 / 4000	Use of non-permitted axes in interpolation with G-codes
32768 / 8000	Invalid parameter in arithmetic operation, e.g. mod 0
10000 hex	Too many SSF commands in one spline line (a maximum of 8 are possible)
20000 hex	Recursion depth exceeded in G-code subroutines with program repeat (O-parameter)