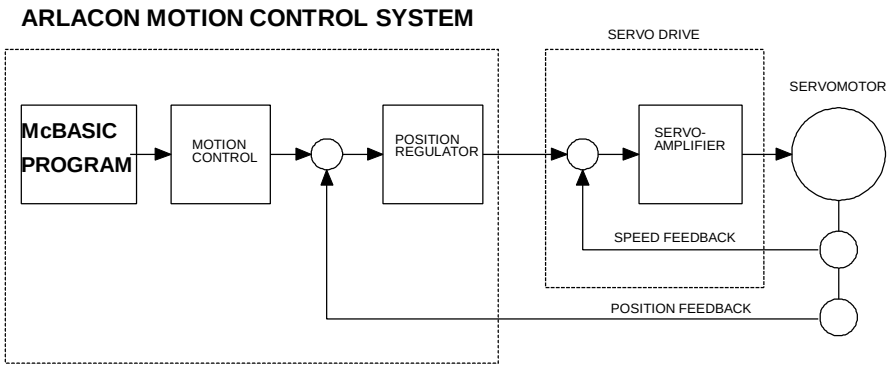
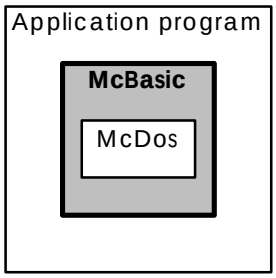
ARLACON McBasic 3.2 programming environment reference manual



Programming environment



Arlacon Motion Controls, Control Techniques SKS Oy
Martinkyläntie 50 01720 VANTAA
tel +358-9-852 678 fax +358-9-852 6740, email: arlacon@sksct.fi
23.1.2001, Ari Lindvall

Table of contents:

1. GENERAL

This manual describes the use of McBasic programming language version 3.2 supplied with ARLACON MC300 and MC400 control systems.

Since the programming environment is used under the McDos operating system we recommend first studying the McDos 2.2 operating system documentation.

In the beginning of this manual there is a short summary of McBasic commands and functions in alphabetic order, after which the startup and programming procedures are described.

Chapter CONTROL describes the commands used in the command mode of the programming environment. The next chapters explain the syntax and operation of program commands and functions for different areas of programming. Examples are provided to clarify operation and use.

In connection with each command and function there is a table describing the meaning of different syntax elements.

Command	Description of operation.
Syntax	The exact form in which the command is written.
elements	Description of the parameters and different forms of syntax.

Function	Description of operation.
Syntax	The exact form in which the function is written.
Type	The type of value returned by the function.
elements	Description of the arguments and different forms of syntax.
Value	Description of values the function can return.

The following notation conventions have been used in this manual.

- Examples have been indented and a typewriterlike text has been used in them.

```
10 REM example
```

- The syntax elements in the commands and functions, that must be replaced with their respective values, have been written in italics.

GOTO *linenumber* SIN(*expression*) MOVE*axes*(*expression*)

- The following notations are used when describing optional parts of syntax

[ ]	brackets, the syntax element is optional
...	ellipsis, the syntax element can be repeated
{ | }	braces, vertical line (or), alternative syntax elements

MOVE*axis*[*axis*[...[*axis*]]] (*expr1*[,*expr2*[,...[,*exprN*]]])

or shorter

MOVE*axes*...(*expr*..)

TRACE{ON|OFF|*n*}

In the examples the parts the user entries are written in normal text

```
PRINT "Text"
Text
```

The parts printed by the control system are written in bold text.

This programming manual has been inspected to be as accurate as possible when describing the details of the McBasic programming language. As McBasic is continuously developed to meet new demands of machine control, some functions may be added or changed in later versions of the language. However, most new developments are designed to be compatible with old McBasic programs to enable simple updating of equipment and software.

Because of the large number of commands and functions in the McBasic language it is probable, that the careful reader finds some parts in this programming manual, where the operation is not explained as accurately as possible.

The author is grateful for all notes and suggestions regarding this manual and will try to use them to enhance the usefulness of this manual in future editions.

Vantaa 23.1.2001

Control Techniques SKS Oy
Arlacon Motion Controls

Ari Lindvall

1.1 MCBASIC COMMANDS

name	description	details in chapter
' text	comment	COMMENTS
command : command	command delimiter	STRUCTURE
var = *expr*	set variable value	VARIABLES AND ARRAYS
label [var,*...,var*]	define jump or subroutine start address	STRUCTURE
ACCEL*axes*..=*expr*	set acceleration	MOTION CONTROL
ACCEL(*axnr,..,axnr*)=*expr*	set acceleration	MOTION CONTROL
ACOMP*axes*..=*expr*	set acceleration feedforward	MOTION CONTROL
ACOMP(*axnr,..,axnr*)=*expr*	set acceleration feedforward	MOTION CONTROL
ADDR *var*,....,*var*	declare address variables	VARIABLES AND TABLES
APPEND *string*	append to program	FILES
BIT *arrayname(dim,..,dim), ..*	declare bit array	VARIABLES AND TABLES
BREAK *addr*	insert breakpoint in program	CONTROL
BYTE(*#devicenr*)=*expr*	write byte	FILES
BYTE *arrayname(dim,..,dim), ..*	declare byte array	VARIABLES AND TABLES
CAPTTYPE*axes*..=*expr*	set position capture mode	MOTION CONTROL
CAPTTYPE(*axnr,...,axnr*)=*expr*	set position capture mode	MOTION CONTROL
CLOSE *#devicenr*	close file	FILES
CONT	continue program after stop	CONTROL
CREEP*axes*..(*expr.*.)	motion at constant speed	MOTION CONTROL
CREEP(*axnr* :*expr,..,axnr*:*expr*)	motion at constant speed	MOTION CONTROL
DATA *data,...,data*	data for READ command	OTHER COMMANDS
DATE$(*#devicenr*)=*datestring*	set date	FILES,CLOCK
DCOMP*axes*..=*expr*	set deceleration feedforward	MOTION CONTROL
DCOMP(*axnr,..,axnr*)=*expr*	set deceleration feedforward	MOTION CONTROL
DECEL*axes*..=*expr*	set deceleration	MOTION CONTROL
DECEL(*axnr,..,axnr*)=*expr*	set deceleration	MOTION CONTROL
DEF FN*fnname....*	define user function	OTHER COMMANDS
DELAY *expr*	delay	TIMING
DELETE *addr,addr*	remove program lines	CONTROL
DERV*axes*..=*expr*	set position control derivation	MOTION CONTROL
DERV(*axnr,..,axnr*)=*expr*	set position control derivation	MOTION CONTROL
DIGITS=*expr*	set printing accuracy	FILES
DIM *variablename*(*dim,..,dim*), ..	declare variables or arrays	VARIABLES AND TABLES
DO..UNTIL..LOOP	repeat loop	STRUCTURE
DOS	return to operating system	CONTROL
DRIVETYPE*axes*..=*type*	configure servo connections	MOTION CONTROL
DRIVETYPE(*axnr,..,axnr*)=*expr*	configure servo connections	MOTION CONTROL
ED *linenumber*|*label*	edit program	CONTROL
EDGE(*input*)={0|1}	input edge acknowledge	INPUT/OUTPUT
ELSE ...	alternate program part	STRUCTURE
ELSEIF ...	alternate conditional program part	STRUCTURE
ENCSIZE*axes*..=*width*	set encoder range	MOTION CONTROL
ENCSIZE(*axnr*,...,*axnr*)=*width*	set encoder range	MOTION CONTROL
END	end program or task	CONTROL
ENDIF	end of IF structure	STRUCTURE
ERROR *errornr*	cause error	ERRORS
EXEC(*string*)	execute command	STRINGS
FILTERSIZEaxes..=*expr*	set position reference filter	MOTION CONTROL
FILTERSIZE(*axnr*,...,*axnr*)=*expr*	set position reference filter	MOTION CONTROL
FLOAT(*#devicenr*)=..	write 4 byte floating point (real) number	FILES
FLOAT *variablename*(*dim,..,dim*), ..	declare 4 byte fp variables (arrays)	VARIABLES AND TABLES
FOLLOW*axis1axis2*(*n1*[*,n2*])	follow another axis position	MOTION CONTROL

FOLLOW(*axnr1,axnr2,n1*[*,n2*])	follow another axis position	MOTION CONTROL
FOR..TO..STEP	repeat loop	STRUCTURE
GAIN*axes..=expr*	set position control gain	MOTION CONTROL
GAIN(*axnr,..,axnr*)=*expr*	set position control gain	MOTION CONTROL
GOSUB [(expr*,..,expr)*] *address*	call subroutine	STRUCTURE
GOTO *address*	jump to program line	STRUCTURE
HELP [#*devicenr*]	display command list	CONTROL
HOME*axes..*	find axes home position	MOTION CONTROL
HOME(*axnr,...,axnr)*	find axes home position	MOTION CONTROL
IEEE32(#*devicenr*)=expression	write 4 byte IEEE unix format fp number	FILES AND COMMUNICATIONS
IEEE32I(#*devicenr*)=expression	write 4 byte IEEE pc format fp number	FILES AND COMMUNICATIONS
IEEE64(#*devicenr*)=expression	write 8 byte IEEE unix format fp number	FILES AND COMMUNICATIONS
IEEE64I(#*devicenr*)=expression	write 8 byte IEEE pc format fp number	FILES AND COMMUNICATIONS
IF..THEN..ELSE..	conditional commands	STRUCTURE
INPUT [#*devicenr,*]..	input value fo variable	FILES AND COMMUNICATIONS
INTEGER *variablename*(*dim,..,dim*), ..	declare 16bit integer variables or arrays	VARIABLES AND TABLES
INTG*axes..=expr*	set position control integration	MOTION CONTROL
INTG(*axnr,..,axnr*)=*expr*	set position control integration	MOTION CONTROL
JOYINP#*n1*[*,n2,n3*]	set joystick input	MOTION CONTROL
LET *variable=expression*	set value for variable	OTHER COMMANDS
LIMITTYPE*axes..=type*	configure axes limit switches	MOTION CONTROL
LIMITTYPE(*axnr,..,axnr*)=*type*	configure axes limit switches	MOTION CONTROL
LINE[(#*n*)]=expression	set output line length	FILES AND COMMUNICATIONS
LINK *#dev1,dev2,dev3*	link output to two other devices	FILES AND COMMUNICATIONS
LIST [#*n*,]*lnumber.. |label..*	list program	FILES AND COMMUNICATIONS
LOAD *string*	load program	FILES AND COMMUNICATIONS
LOCAL *variable, ...*	declare variables local	VARIABLES AND TABLES
LOG*axes..=n*	control motion control log operation	MOTION CONTROL
LOG(*axnr,...,axnr*)=*n*	control motion control log operation	MOTION CONTROL
LOGDATA*axes=expr*	set analog input to attach to axis log	MOTION CONTROL
LOGDATA(*axnr,..,axnr)=expr*	set analog input to attach to axis log	MOTION CONTROL
LOGSIZE*axes=expr*	set size of log array	MOTION CONTROL
LOGSIZE(*axnr,..,axnr)=expr*	set size of log array	MOTION CONTROL
LOOP	see DO .. UNTIL .. LOOP	STRUCTURE
MAXERR*axes..=expr*	set position error limit	MOTION CONTROL
MAXERR(*axnr,...,axnr*)=*expr*	set position error limit	MOTION CONTROL
MOVC*axes*..(*expr*..)	continuous absolute motion	MOTION CONTROL
MOVC(*axnr:expr,..,axnr:expr*)	continuous absolute motion	MOTION CONTROL
MOVCR*axes*..(*expr*..)	continuous relative motion	MOTION CONTROL
MOVCR(*axnr:expr,..,axnr:expr*)	continuous relative motion	MOTION CONTROL
MOVE*axes*..(*expr*..)	absolute motion	MOTION CONTROL
MOVE(*axnr:expr,..,axnr:expr*)	absolute motion	MOTION CONTROL
MOVER*axes*..(*expr*..)	relative motion	MOTION CONTROL
MOVER(*axnr:expr,..,axnr:expr*)	relative motion	MOTION CONTROL
MOVEPROF*axes*..(*axis*)	activate profile controlled motion	MOTION CONTROL
MOVEPROF(*axnr:axnr,.,axnr:axnr*)	activate profile controlled motion	MOTION CONTROL
NEW	clear program memory	CONTROL
NEXT *variable*	end of repeat loop	STRUCTURE
NOBREAK *address*	remove breakpoint	CONTROL
NOBREAKS	remove all breakpoints	CONTROL
NUMBER *lnum, lnum, addr, addr*	number unnumbered program lines	CONTROL
OFFSET*axes=expression*	set offset value or move current position	MOTION CONTROL
OFFSET(*axnr,...,axnr*)=*expression*	set offset value or move current position	MOTION CONTROL
OFREQ(outanr)=expr	set frequency output	ANALOG I/O
ON *var* GOSUB *addr,..,addr*	select subroutine	STRUCTURE
ON *var* GOTO *addr*,..,*addr*	select jump address	STRUCTURE
ON ERROR *addr*	set error trap	ERRORS
OPEN *#devicenr,string*	open file or port	FILES AND COMMUNICATIONS

OPWR*axes*..=*expression*	set servo axes reference output	MOTION CONTROL
OPWR(*axnr,..,axnr)=expression*	set servo axes reference output	MOTION CONTROL
OUT(*outputnr*)={0|1}	set binary output	INPUT/OUTPUT
OUTA(*outputnr*)=*expression*	set analog output	ANALOG I/O
PIDFREQ=expression	set position loop repeat rate	MOTION CONTROL
POS*axes=expression*	set current axes position	MOTION CONTROL
POS(*axnr,..,axnr)=expression*	set current axes position	MOTION CONTROL
PRINT [*#devicename*,]...	data output	FILES AND COMMUNICATIONS
PRIOR=*n*	set task priority	STRUCTURE
PROF*axes*..(*index*)=*expr*	write to profile table	MOTION CONTROL
PROF(*axnr,index*)=*expr*	write to profile table	MOTION CONTROL
PROFSIZE*axes=expr*	set profile table size for axes	MOTION CONTROL
PROFSIZE(*axnr,..,axnr)=expr*	set profile table size for axes	MOTION CONTROL
PTR(*#devicenr*)=*expression*	set file pointer	FILES AND COMMUNICATIONS
PWR*axes*..=*expression*	set servo axes reference limit	MOTION CONTROL
PWR(*axnr,..,axnr)=expression*	set servo axes reference limit	MOTION CONTROL
READ var,..,var	read data from DATA lines	OTHER COMMANDS
REAL *variablename*(*dim,..,dim*), ..	declare real variables or arrays	VARIABLES AND TABLES
REM *comment*	comment	COMMENTS
REN *lnum, lnum, addr, addr*	renumber program lines	CONTROL
RES*axes*..=*expression*	set axes position counter resolution	MOTION CONTROL
RES(*axnr,..,axnr)=expression*	set axes position counter resolution	MOTION CONTROL
RESTORE *address*	set data pointer for READ	OTHER COMMANDS (DATA)
RESUME	return from error trap routine	ERRORS
RESUME NEXT	return from error trap routine	ERRORS
RETURN	return from subroutine	STRUCTURE
RUN	start program execution	CONTROL
SAVE *string*	save program	FILES AND COMMUNICATIONS
SCOMP*axes*..=*expression*	set axes speed compensation	MOTION CONTROL
SCOMP(*axnr,..,axnr)=expression*	set axes speed compensation	MOTION CONTROL
SHORT INTEGER *varname*(*dim,*...), .	declare 8bit integer variables or arrays	VARIABLES AND TABLES
SIZE(*#devicenr*)=*size*	set file size	FILES
SPEED*axes*..=*expression*	set axes speed	MOTION CONTROL
SPEED(*axnr,..,axnr*)=*expression*	set axes speed	MOTION CONTROL
STOP	stop program execution	CONTROL
STOPMOVE*axes*..	stop motion	MOTION CONTROL
STOPMOVE(*axnr,..,axnr*)	stop motion	MOTION CONTROL
STRING[(*length*)] *varname*$(dim, ..)..	declare string variables	STRINGS
SYMBOLS	display symbols in use	CONTROL
SYSTEM	exit to McDos	CONTROL
SYSTEM(*string*)	execute McDos command	CONTROL
TASK *address*	create task	STRUCTURE
TASKMAX=*n*	set max. task number	STRUCTURE
TIMER[(*n*)]=..	set timer	TIMING
TRACE {ON|OFF|*tasknr*}	trace program execution	CONTROL
UNTIL *condition*	see DO .. UNTIL .. LOOP	STRUCTURE
WATCHDOG(outputaddr)=freq	set watchdog operation	INPUT/OUTPUT
WAYMOD$(*n,m*)=	set McWay i/o configuration	I/O CONFIGURATION
WORD(*#devicenr*)=..	write word	FILES AND COMMUNICATIONS
WORD *varname*(*dim,*...), ...	declare 16bit word variables or arrays	VARIABLES AND TABLES

1.2 MCBASIC FUNCTIONS

name	description	details in chapter
ABS(*expression*)	absolute value	MATHEMATICS
ACCEL*axis*	read axis acceleration	MOTION CONTROL
ACCEL(*axnr*)	read axis acceleration	MOTION CONTROL
ACOMP*axis*	read acceleration feedforward	MOTION CONTROL
ACOMP(*axnr*)	read acceleration feedforward	MOTION CONTROL
ADDR$(*address*)	convert address to string	STRINGS
AND	boolean or binary AND function	MATHEMATICS
ASC(*string*)	ASCII character to number	STRINGS
ATAN(*expression*)	arcus tangent	MATHEMATICS
BIN$(*expression*)	expression to binary string	STRINGS
BYTE(*#devicenr*)	read byte	FILES AND COMMUNICATIONS
CAPTPOS*axis*	read captured position	MOTION CONTROL
CAPTPOS(*axnr*)	read captured position	MOTION CONTROL
CAPTTYPE*axis*	read position capture status	MOTION CONTROL
CAPTTYPE(*axnr*)	read position capture status	MOTION CONTROL
CLOCK	system timer	TIMING
CHR$(*expression*)	number to ASCII character	STRINGS
COLOR$(*colornumber*)	set display terminal color	FILES AND COMMUNICATIONS
COS(*expression*)	cosine	MATHEMATICS
CURS$(*xcoord,ykoord*)	set cursor position	FILES AND COMMUNICATIONS
DATAPTR@	read data pointer	OTHER COMMANDS
DATE$(*#devicenr*)	read date	FILES AND COMMUNICATIONS
DATE$(*expression*)	convert number to long date string	FILES AND COMMUNICATIONS
DATE(*datestring*)	convert date to number	FILES AND COMMUNICATIONS
DEC$(*expression*)	expression to decimal string	STRINGS
DECEL*axis*	read axis deceleration	MOTION CONTROL
DECEL(*axnr*)	read axis deceleration	MOTION CONTROL
DERVaxis	read position control derivation	MOTION CONTROL
DERV(*axnr*)	read position control derivation	MOTION CONTROL
DIR$(*#devicenr,entry*)	read directory item	FILES AND COMMUNICATIONS
DRIVETYPE*axis*	read axis type	MOTION CONTROL
DRIVETYPE(*axnr*)	read axis type	MOTION CONTROL
EDGE(*input*)	read edge detector state	INPUT/OUTPUT
ENCERR*axis*	read encoder error counter	MOTION CONTROL
ENCERR(*axnr*)	read encoder error counter	MOTION CONTROL
ENCSIZE*axis*	read encoder type	MOTION CONTROL
ENCSIZE(*axnr*)	read encoder type	MOTION CONTROL
ERL	read error line number	ERRORS
ERL$	read error line	ERRORS
ERR	read error number	ERRORS
ERR@	read error line address	ERRORS
ERR$(*errornumber*)	error message	ERRORS
EXP(*expression*)	power of e	MATHEMATICS
FILL$	draw filled area	FILES AND COMMUNICATIONS
FILTERSIZE*axis*	read axis position reference filter setting	MOTION CONTROL
FILTERSIZE(*axnr*)	read axis position reference filter setting	MOTION CONTROL
FLOAT(*#devicenr*)	read real number	FILES AND COMMUNICATIONS
FN*name*(*argument,..*)	user defined numerical function	OTHER COMMANDS
FN*name*$(*argument,..*)	user defined string function	OTHER COMMANDS
FPOS*axis*	read position set value after filter	MOTION CONTROL
FPOS(*axnr*)	read position set value after filter	MOTION CONTROL
FREE(*n*)	user memory status	CONTROL

FSPEED*axis*	read position set value speed after filter	MOTION CONTROL
FSPEED(*axnr*)	read position set value speed after filter	MOTION CONTROL
GAIN*axis*	read position control gain	MOTION CONTROL
GAIN(*axnr*)	read position control gain	MOTION CONTROL
IEEE32(*#devicenr*)	read 4 byte IEEE unix format fp number	FILES AND COMMUNICATIONS
IEEE32I(*#devicenr*)	read 4 byte IEEE pc format fp number	FILES AND COMMUNICATIONS
IEEE64(*#devicenr*)	read 8 byte IEEE unix format fp number	FILES AND COMMUNICATIONS
IEEE32I(*#devicenr*)	read 8 byte IEEE pc format fp number	FILES AND COMMUNICATIONS
HEX$(*expression*)	expression to hexadecimal string	STRINGS
IFREQ(a*input*)	read frequency input	ANALOG I/O
INP(*input*)	read binary input	INPUT/OUTPUT
INPA(*input*)	read analog input	ANALOG I/O
INSTR(*expr,string,substring*)	locate substring	STRINGS
INT(*expression*)	integer part of number	MATHEMATICS
INTG*axis*	read position control integration	MOTION CONTROL
INTG(*axnr*)	read position control integration	MOTION CONTROL
JOY(*n*)	joystick position	MOTION CONTROL
JOYX	joystick position = JOY(0)	MOTION CONTROL
JOYY	joystick position = JOY(1)	MOTION CONTROL
LEFT$(*string*,*expression*)	part of string (from end)	STRINGS
LEN(*string*)	length of string	STRINGS
LIMITTYPE*axes*	read axis limit switch configuration	MOTION CONTROL
LIMITTYPE(*axnr*)	read axis limit switch configuration	MOTION CONTROL
LINE$	draw line	FILES AND COMMUNICATIONS
LOG(*expression*)	natural logarithm	MATHEMATICS
LOGDATA*axis*(*sample*,*data*)	read position control log data	MOTION CONTROL
LOGDATA(*axnr,sample*,*data*)	read position control log data	MOTION CONTROL
LOGSIZEaxis	read log array size	MOTION CONTROL
LOGSIZE(*axnr*)	read log array size	MOTION CONTROL
MAX(*expr1*,*expr2*)	greater of two numbers	MATHEMATICS
MAXERR*axis*	read current position error limit	MOTION CONTROL
MAXERR(*axnr*)	read current position error limit	MOTION CONTROL
MID$(*string,expr1,expr2*)	part of string	STRINGS
MIN(*expr1,expr2*)	smaller of two numbers	MATHEMATICS
MOVEBUFFER*axes*..	read motion buffer memory status	MOTION CONTROL
MOVEBUFFER(*axnr,..,axnr*)	read motion buffer memory status	MOTION CONTROL
MOVEREADY*axes*..	read axis status	MOTION CONTROL
MOVEREADY(*axnr,..,axnr*)	read axis status	MOTION CONTROL
NOT *expression*	logical negation	MATHEMATICS
OFF	constant 0	MATHEMATICS
OFFSET*axis*	read axis offset value	MOTION CONTROL
OFFSET(*axnr*)	read axis offset value	MOTION CONTROL
OFREQ(aoutput)	read output frequency	ANALOG I/O
ON	constant 1	MATHEMATICS
ONERR@	read current error trap address	ERRORS
OPWR*axis*	read axis reference output	MOTION CONTROL
OPWR(*axnr*)	read axis reference output	MOTION CONTROL
OR	boolean or binary OR function	MATHEMATICS
OUT(*output*)	read binary output status	INPUT/OUTPUT
OUTA(*output*)	read analog output status	ANALOG I/O
PIDFREQ	read position loop repeat rate	MOTION CONTROL
PII	constant pi (3.1415926..)	MATHEMATICS
PNT$(*xcoord,ycoord*)	graphics point	FILES AND COMMUNICATIONS
POS*axis*	read axis position	MOTION CONTROL
POS(*axnr*)	read axis position	MOTION CONTROL
POSERR*axis*	read axis position error	MOTION CONTROL
POSERR(*axnr*)	read axis position error	MOTION CONTROL
PRIOR	read current task priority	STRUCTURE

PROF*axis*(*sample,data*)	read from profile table	MOTION CONTROL
PROF(*axnr,sample,data*)	read from profile table	MOTION CONTROL
PROFSIZE*axis*	read axis profile table size	MOTION CONTROL
PROFSIZE(*axnr*)	read axis profile table size	MOTION CONTROL
PTR(*#devicenr*)	read file data pointer	FILES AND COMMUNICATIONS
PWR*axis*	read position control output limit	MOTION CONTROL
PWR(*axnr*)	read position control output limit	MOTION CONTROL
REAL(*#devicenr*)	read 8 byte fp (real) number	FILES AND COMMUNICATIONS
RES*axis*	read axis resolution	MOTION CONTROL
RES(*axnr*)	read axis resolution	MOTION CONTROL
RIGHT$(*string*,*expression*)	part of string (from beginning)	STRINGS
RND(*expression*)	random number	MATHEMATICS
RPOS*axis*	read position set value before filter	MOTION CONTROL
RPOS(*axnr*)	read position set value before filter	MOTION CONTROL
RSPEED*axis*	read position set value speed before filter	MOTION CONTROL
RSPEED(*axnr*)	read position set value speed before filter	MOTION CONTROL
SCOMPaxis	read axis speed compensation	MOTION CONTROL
SCOMP(*axnr*)	read axis speed compensation	MOTION CONTROL
SGN(*expression*)	sign of number	MATHEMATICS
SIN(*expression*)	sine	MATHEMATICS
SIZE(*#devicenumber*)	read file size or output buffer free space	FILES AND COMMUNICATIONS
SPEED*axis*	read axis set speed	MOTION CONTROL
SPEED(*axnr*)	read axis set speed	MOTION CONTROL
SQR(*expression*)	square root	MATHEMATICS
STR$(*expression*)	number to string	STRINGS
TAB(*expression*)	set cursor on line	FILES AND COMMUNICATIONS
TAN(*expression*)	tangent	MATHEMATICS
TASK	current task number	STRUCTURE
TASKMAX	maximum task number	STRUCTURE
TEXT$	return to text mode (end of graphics)	FILES AND COMMUNICATIONS
UCASE$(*string*)	convert string to uppercase	STRINGS
TIMER[(*n*)]	read timer value	TIMING
VAL(*string*)	string to number	STRINGS
WAYERR(*waynr*)	read McWay error counter	I/O CONFIGURATION
WAYMOD$(*waynr,modnr*)	read McWay configuration	I/O CONFIGURATION
WORD(*#devicenr*)	read word	FILES AND COMMUNICATIONS
XOR	boolean or binary XOR function	MATHEMATICS

2. GETTING STARTED

McBasic is a control system oriented programming environment with efficient commands and functions for utilising the hardware and connections of ARLACON machine control systems.

The commands and functions for standard data processing, calculations and output are essentially compatible with many popular Basic language interpreters and compilers.

2.1 MCBASIC VERSIONS

Since McBasic is a control system oriented environment, different ARLACON control system models are equipped with system specific versions of the system software.

Some properties of the environment, such as axis number and names and available i/o, are dependent of the software and hardware configurations used.

The manufacturer configures the McBasic environment software for ARLACON control systems at factory when installing the software in the control system.

Current versions of McBasic for McDOS include:

BASIC.CK	plain McBasic for system utility
BAS0.CK	McBasic without axes with CMOS memory features
BAS16.CK	16 logical axes version for MC400 (MC400 standard version)
BAS16X.CK	As BAS16.CK but for use with McWay loop W0: connected to CPU5 module port S1:
BAS100.CK	100 logical axes version for MC400
BAS300.CK	16 logical axes version for MC300 (MC300 standard version)

2.2 STARTING THE SYSTEM

At power up, the MC control system starts the McDOS operating system as described in the McDOS user's manual. After this McDOS seeks for the WAKEUP.EX batch file from D1: ... D8: root directories and, if found, executes the commands in the file. This way the startup procedure can be defined to suit the application.

The McBasic environment is started from the McDOS operating system by typing the name of the McBasic version used as command name:

BAS16 [*progname1*, ..., *prognamen*]

where *progname..* are names of programfiles to be loaded and run when starting McBasic.

If no *progname* is specified, McBasic starts the execution of the program (if any) currently in its work memory. If McBasic detects that a program previously in memory has been destroyed, it searches and loads a backup copy from a file WAKEUP.BA as described in chapter FILES.

If there is no program in the memory, McBasic stays in the command mode.

2.3 WRITING PROGRAMS

McBasic programs can be written with many different tools.

One way to write a program for the system is to use the command mode of the McBasic environment. In this mode program lines can be entered from the console serial communications (:CN) using a terminal, PC or ARLACON display terminal. Among other communications programs, the Terminal program of the McBench programming environment for PC shipped with MC300 and MC400 controllers is well suited for this. Program lines should be entered in the order of their execution, if they are not numbered.

Other ways to work with programs are editing the program files with the TX text editor contained in McDos as a command file or editing program files in a PC using the McBench program editor or another editor suitable for editing ASCII files. Programs can then be loaded to the control system either using a diskette, if applicable, or serial comms. (see McBench documentation).

Program lines can be marked with numbers, labels and be written without them. An empty line is also interpreted as a program line. A line marked with a label should begin from the first position of the line. Lines without labels or numbers must have at least one space character in the beginning of the line. Lines can be numbered from 1 to 65535.

```
  10 PRINT "first line"
  30 PRINT "third line"
  20 PRINT "second line"
Lab1 PRINT "fourth line"
     PRINT "fifth line"
```

In McBasic command mode program lines with line numbers can be entered by just typing them and ending the line with a <return>. They can also be replaced by typing a new line with the same number. Any type of program lines can be entered and edited with the ED command described in chapter 3. Editing a program causes current values of variables to be cleared.

By exiting McBasic to McDos operating system programs can be written and edited as text files using the McDos TX screen editor. Programs can also be edited in other systems such as a PC computer and loaded to the MC control system either using a diskette, if available, or via serial communications with ASCII transfer.

2.4 COMMAND AND VARIABLE NAMES

When typing command and function names, both uppercase and lowercase characters can be used, whereas variable names are case sensitive.

```
40 print sin(3.14);COS(1.65)
```

Variable names can be either “short” or “long”. A “short” numerical variable name consists of one letter and optionally one number. A “short” string name consists of a letter followed by $.

Short numerical variable names:

```
A a A3 a9 z8 z Z b B0 B9
```

Short string names:

```
A$ a$ z$ Z$
```

Variables with “short” names can be used as global variables without declaring them, whereas “long” variable names must be declared, as described in chapters 6 and 7, before using them in the program.

2.5 VARIABLE TYPES

String and address variables are the only McBasic variables that McBasic does not automatically convert according to use. Thus, for example, using a string variable when a command or a function requires a numerical value, causes McBasic to display an error message.

All McBasic numerical expressions can normally be used for commands and functions requiring numerical values. However, care must be taken not to exceed the minimum and maximum value allowed.

Some considerations:
- Any non-zero value is considered to be true. Only 0 is false.
- Using a non-integer value in a command or function requiring an integer value causes normally McBasic to round the value to the nearest integer value.
- When using too high values in some integer operations McBasic normally uses the value defined by the least significant bits.

When using long variable names or large arrays It is necessary to declare the variables before using them. Declaration can also be used to limit the scope of the variable as described in chapter 7.

Examples of variable declarations:

```
STRING Abc$,C$(5),D$(8,8)          'string of 80 characters
STRING(5) SymFirst$,SymArray(3)    'string of an arbitrary length in the range 1...255,
                                    here 5 characters.
REAL A,Varia(4),B(3,3)             'floating point numbers with a length of 8 bytes
ADDR A@(5),Jump@                   'address arrays and variables
```

Following types are used for arrays only:

```
FLOAT N(20),Dx(10,100)             '4 byte floating point numbers
WORD a(5),c(5,5)                   '16 bit word, unsigned ( 0...65535)
INTEGER Abc(3), Cba(6,6)           '16 bit word, signed (-32768...32767)
BYTE Abc(3), Cba(6,6)              '8 bit byte, unsigned (0...255)
SHORT INTERGER  Cba(6,6)           '8 bit byte, signed (-128...127)
BIT Abc(3), Cba(6,6)               '1 bit (0..1)
```

When numerical data is copied to an array of different type the values are rounded to nearest integer if necessary. An error message is displayed if the value is not within valid limits.

2.6 LABELS

McBasic uses labels or line numbers to mark program lines to be referred to from applicable commands. A label is a string of characters of arbitrary length starting with a letter. Upper and lowercase characters, numbers (0...9) and characters: !#$%&@ can be used in a label.

A label must start at the first column and can be followed by a command.

For example

```
10 REAL Abc    'some numbered lines
15 Abc=2
20 PRINT Abc   'Print is a command

Label1 Abc=Abc+10    'Label1 is a label
   PRINT Abc   'Print is command. Note spaces before PRINT
```

```
 2
12
```

Following error messages can be generated when misusing labels:
- error #44, when the command addresses a nondefined label
- error #45, when an attempt is made to define a label twice

Both labels and line numbers may be used in the following commands to determine the destination of operation

GOTO, GOSUB, ON ERROR, ON... GOTO, ON...GOSUB, LIST, DELETE, ED, RESTORE, REN

In most cases also mixed usage of labels and line numbers is possible in these commands.

For example

```
     ON I GOTO 100, Lab2, 300, Lab4,500,600
Lab2 GOSUB 1000
Lab4 GOSUB CalSum

     DELETE 100,Lab1  'deletes lines from 100 to label Lab1
```

Additionally, address variables and expressions can also be used in conjunction with the above listed commands excluding ON GOTO and ON GOSUB.

2.7 PROCEDURES

When using a label to call subroutines with the GOSUB command, it is also possible to define parameters to be passed to the subroutine as local variables. A subroutine using parameter passing is called a procedure.

For example

```
DispArea(R) R=R*R*PII
 PRINT "Area=";R
 RETURN
```

The procedure is called by giving a list of parameters with the label. For example

```
GOSUB DispArea(100*X)
```

The parameter R is local within the procedure with the initial value passed when calling the procedure.The defined parameters can have any name and are assumed real numbers, max. 80 character strings if ending in $ or addresses if ending in @. Up to 8 parameters can be defined for a procedure.

3. CONTROL

This chapter describes the operation of the commands controlling the operation on the McBasic programming environment.

3.1 ED

Command	Edit program.
Syntax	ED[*address*]
address	Place in program to start editing.

ED command allows typing in and editing program from McBasic command mode. While it is possible to type in program lines with linenumbers without using ED, it allows editing of previously entered lines and typing in lines without linenumbers.

When the command is issued, the referred line appears on the screen and can be edited. It is then possible to move up and down in the program with arrow keys. The currently edited line is always printed on the screen below the previous edited line independent ED returns to McBasic command mode with the <return> key.

When in ED editing mode, using an ANSI compatible terminal, other functions are available as follows:

⇦	Move left on line
⇨	Move right on line
⇧	Previous line
⇩	Next line
F1	Insert character
F2	Delete character
F3	Insert line
F4	Delete line
F6	Insert line from buffer.

With the F6 function previously edited lines can be browsed and inserted in the program from the 250 characters long buffer. Moving to another line with ⇧⇩ keys or exiting wiht <enter> completes editing the line.

3.2 HELP

Command	Print (list of) commands and functions.
Syntax	HELP[#*nn*][*searchstring*]
nn	The output device that is used. (default = CN:)
searchstring	Optional string, a substring in the command/function to search for. If omitted, all commands and functions are listed.

Prints a list of available commands and functions containing the specified substring to the specified device. Short syntax descriptions are also included. HELP can be used for example to check the

commands and functions available in the McBasic version used. It can also be used as help when programming.

```
B>HELP

HELP ...
LINE(#expression)=expression
RESUME_NEXT
.
.
OPEN#2,":LP"    ' the printer is connected here
HELP#2          ' print the list...
```

Or to look for help on a specific subject:

```
B>HELP "BUF"
list of functions
MOVEBUFFER(expression, ..)
MOVEBUFFERaxles
```

3.3 DOS

Command	Return to McDos operating system
Syntax	DOS SYSTEM X

The operation has three alternate syntaxes

```
B>DOS

D4:/>        (McDos prompt: current path+">",
              note the McBasic prompt "B>")
```

3.4 SYSTEM

Command	Call resident McDos commands from program.
Syntax	SYSTEM(*string*)
string	Executable resident McDos command

For example create new directory from within an McBasic program.

```
SYSTEM("MKDIR D4:/TEMP")
```

Note that in McBasic command mode it is possible to execute McDOS commands using a period before the command:

```
B>.DIR
```

3.5 NEW

Command	Clear program memory.
Syntax	NEW

Program memory is cleared, variables and program are erased.

```
B>NEW
McBasic16 3.2at
        Program
        System tables
        Variables & compilations
        Recycled
 719550 Free
B>
```

When the program memory has been cleared, a McBasic version/revision and memory status message is displayed. The message shows that program and symbol areas are empty and all available memory is free. The amount of the free memory depends on the control system model and memory size and the McBasic version used.

3.6 RUN

Command	Start program execution
Syntax	RUN

Program starts at the first line. Variable values are cleared.

3.7 END

Command	End of program or task
Syntax	END

Ends program execution. END command can be used as the last command in the program. However, the use of END in the end of the program is not obligatory.

When using the TASK command to create more than one simultaneous program tasks, END can be used to end a task. Thus, a program can contain several END commands to end tasks.

3.8 STOP

Command	Stop program execution.
Syntax	STOP

Equivalent to stopping the program by sending a Ctrl-X in the console serial interface (CN:). A STOP command can be included in the program for example to stop the program in a special condition or they can be used as breakpoints for testing purposes.

The variables in the stopped program can be observed. Program execution can be continued with the CONT command.

3.9 BREAK

Command	Set breakpoints in program.
Syntax	BREAK *addr*
addr	Address for the new breakpoint

The BREAK command allows setting breakpoints without altering the program. Therefore, breakpoints can be inserted and removed when the program is stopped without loosing variable values or having to restart the program.

```
Examples:

BREAK 100          ' set breakpoint at line 100 in a line numbered program
BREAK Skip         ' set breakpoint at label MySub
BREAK Skip+5       ' set breakpoint at 5 lines after label Skip
BREAK Finish-2     ' set breakpoint at 2 lines befor label Finish
BREAK MySub()      ' set breakpoint at procedure label starting
```

3.10 NOBREAK

Command	Remove breakpoints from program
Syntax	NOBREAK *addr* or NOBREAKS
addr	Adress of the breakpoint to remove. Alternate syntax NOBREAKS removes all breakpoints.

3.11 CONT

Command	Continue program execution after a breakpoint, STOP or Ctrl-X.
Syntax	CONT

The program continues from the next line (or lines if several tasks) after the point it was stopped. Variable values are not affected.

3.12 TRACE

Command	Control program tracing.
Syntax	TRACE {ON|OFF|*n*}
ON	Program tracing on
OFF	Program tracing off
n	selected TASK: 0 Program tracing off 1 All TASKs n Only TASK n

Program execution tracing. Executed lines are listed at console while running the program. If more than one TASK is being used, only one TASK can be traced by selecting with its TASK number.

```
TRACE 3
```

TASKs are numbered 2....9. When the program starts, the first TASK is number 2. New tasks are numbered in creating order.

Tracing program execution generates intense listing of program lines and therefore causes programs to run slowly. In some cases it is also possible to insert TRACE commands in the program to control partial tracing of program to preserve processing speed.

3.13 DELETE

Command	Remove program lines.
Syntax	DELETE [*addr1*][,*addr2*]
addr1	The address of the first line to be removed (default beginning of program).
addr2	The address of the last line to be removed (default end of program).

When used without parameters the DELETE command removes all program lines. This operation can be done faster with the NEW command.

With the command DELETE *addr1* line *addr1* and the lines after it are removed.

With the command DELETE ,*addr2* line *addr2* and the lines before it are removed.

With the command DELETE *addr1*,*addr2* line *addr1* and *addr2* and the lines between them are removed.

If *addr1* and *addr2* are the same line, only that line is removed. In line numbered programs the same operation can be performed by giving only the line number. A new line can also be given to replace the old one.

The use of DELETE command clears the values of variables.

For example remove lines Lab1 and Lab2 and the lines between them.

```
DELETE Lab1,Lab2
```

3.14 REN

Command	Renumber program lines.
Syntax	REN [*nn1*[,*nn2*[,{*nn3*|*label3*}[,{*nn4*|*label4*}]]]]
nn1	The first line number of the area to be renumbered (default 10).
nn2	The step between new line numbers (default 10).
nn3|*label3*	The line number/label of the first line to be renumbered (default beginning of program).
nn4|*label4*	The line number/label of the last line to be renumbered (default end of program).

A REN command without any parameters renumbers the program lines beginning from line 10 using steps of 10. The jump addresses etc. are updated automatically according to the new line numbering.

Also only some of the parameters can be given and in this case the other parameters will use their default values.

```
REN
REN 1000,10
REN 2000,,1000,2999
REN 2000,10,1000,2999
```

Numbering must be possible without changing the order of the program lines, otherwise McBasic does not renumber any lines. If in the renumbered block lines without numbers are present, they will not be affected by this command.

REN command does not set the variable values to zero. In some cases programs can be build using the EXEC function so that renumbering changes or even prevents the execution of the program. In these cases REN command must not be used in these part of the programs.

3.15 NUMBER

Command	Numbers all program lines.
Syntax	NUMBER [*nn1*[,*nn2*[,{*nn3*|*label3*}[,{*nn4*|*label4*}]]]]
nn1	The first line number of the program to be numbered (default 10).
nn2	The step between new line numbers (default 10).
nn3|*label3*	The line number/label of the first line to be numbered (default beginning of program).
nn4|*label4*	The line number/label of the last line to be numbered (default end of program).

A NUMBER command without any parameters numbers all program lines beginning from line 10 using steps of 10. The jump addresses etc. are updated automatically according to the new line numbering.

Also only some of the parameters can be given and in this case the other parameters will use their default values.

```
NUMBER
NUMBER 1000,10
```

The operation of the NUMBER command resembles that of the REN command. The difference is that the REN command only renumbers lines that are already numbered wherease the NUMBER command numbers also lines that have no line number. Therefore, when using the NUMBER command, the program cannot contain labels.

3.16 FREE

Function	User memory status	
Syntax	FREE(*n*)	
Type	Integer	
n	-3	size of compressed program source code
	-2	size of system tables like LOG, PROF
	-1	size of variables and compiled program
	0	size of free memory (recycled memory not included)
	1	size of total recycled memory
	2	size of recycled numerical variables
	3	size of recycled string variables
	4	size of recycled variables of mixed sizes, arrays, strings
	5	size of recycled subroutine links
	6	size of recycled task blocks
Value	Size of memory occupied by different program components, determined by *n* value.	

The following relations are valid

FREE(1)=FREE(2)+FREE(3)+FREE(4)+FREE(5)+FREE(6)

Full memory = FREE(-3)+FREE(-2)+FREE(-1)+FREE(0)+FREE(1)

4. STRUCTURE

The structure commands controlling the program operation are used as follows.

4.1 :

Command	Command separator.
Syntax	*command* : *command* [: *command* : ... : *command*]
command	Commands written on same program line.

Several commands can be written on same program line when separated by command separator.

Some limitations:
- DATA command must always be at the beginning of the line.
- DEF command must always be at the beginning of the line.

4.2 GOTO

Command	Jump to given program address.
Syntax	GOTO *addr*
addr	Line number or label, destination address.

Execution of the program will continue from line nnn or label *nnn*.

```
100 GOTO 290   'the program is continued from line 290
120 GOTO Lab1  'program is continued from line labeled Lab1
```

By using the GOTO command from command prompt the program can be started or continued from some other than the very first line. The variable values are not affected.

```
GOTO 5000
```

starts the program from line 5000. Address expressions can also be used:

```
GOTO Label1+3
GOTO 10000+1
```

4.3 GOSUB

Command	Sub-routine call.
Syntax	GOSUB *addr* or GOSUB *label*(*par1,par2,...*)
addr	The first line of the subroutine. Line number or label.
label()	Label of the first line of a procedure using parameter passing.
par1,par2,.	Values for local variables defined in conjunction with the *label*().

Max. 25 subroutine calls can be nested.

The program continues from *addr*. RETURN at the end of subroutine returns the operation to the next command after GOSUB.

GOSUB command can be also used from command prompt, for example for testing the operation of a subroutine. RETURN command at the end of subroutine returns the control back to command prompt.

```
110 GOSUB 2000
120 GOSUB ArrayIni
    GOSUB Draw(100,200)
    GOSUB MoveArm(X(n),Y(n),A(n))
```

4.4 RETURN

Command	Return from a subroutine.
Syntax	RETURN

Return from a subroutine. The last command in a subroutine must always be a RETURN command. After RETURN operation of the program continues from the next McBasic command after GOSUB and all local variable space used in the subroutine is freed. For example:

```
1200 GOSUB 1400 : STOP
1400 PRINT "Subroutine"
1410 RETURN

>RUN

Subroutine

>
```

4.5 ON GOTO

Command	Jump to a selected program line.
Syntax	ON *expression* GOTO *addr1,addr2,..,addrn*
expression	Integer defining which of the given addresses will be used as jump address. This value must be between 1 and n.
addr1	Jump address, when *expression* is 1.
addr2	Jump address, when *expression* is 2.
addrn	Jump address, when *expression* is n.

Selection structure that selects the jump address according to the value of *expression*.

```
1500 ON X0+1 GOTO 2000,3000,1000
```

If X0=0 the jump address is 2000, if X0=1 the jump address is 3000 and if X0=2 the jump address is 1000.

```
ON X0+1 GOTO Lab1,Lab2,Lab3
```

If X0=0 the jump label is Lab1, if X0=1 the jump label is Lab2 and if X0=2 the jump label is Lab3.

If the value of *expression* is not between 0 ... N, the McBasic gives an error message.

4.6 ON GOSUB

Command	Subroutine call to a selected program line.
Syntax	ON *expression* GOSUB *addr1,addr2,..,addrn*
expression	Integer defining which of the subroutines beginning at given linenumbers or labels will called. The value must be between 1 and n.
addr1	Address of subroutine, when value of *expression* is 1.
addr2	Address of subroutine, when value of *expression* is 2.
addrn	Address of subroutine, when *expression* is n.

Selection structure that selects the subroutine to be called according to the value of expression. This structure cannot be used for calling subroutines with parameters (procedures).

```
1600 ON X0+1 GOSUB Lab1,Bal2,Res3
```

Operation is similar to ON .. GOTO structure.

4.7 IF THEN [ELSE]

Command	Conditional jump.
Syntax	IF *condition* THEN *linenumber1* [ELSE *linenumber2*]
condition	Expression whose truth-value defines the jump address.
linenumber1	Jump address, when *condition* is true (<>0).
linenumber2	Jump address, when *condition* is false (=0). If [ELSE *linenumber2*] part is not given, the program continues from the next line.

Using this command the program operation can be directed according to a logical value.

```
IF A=7 THEN Label1 ELSE Label2
```

The same structure can also be used for conditional execution of alternative commands

Command	Conditional commands structure (one line).
Syntax	IF *condition* THEN *commands* [ELSE *commands*] [: ENDIF : .....]
IF *condition*	When *condition* is true, allows the execution of *command*s after THEN. Program execution then continues from next line or ENDIF, if used.
ELSE	If *condition* was not true, allows the execution of *command*s after ELSE or between ELSE and ENDIF, if used.
ENDIF	Only necessary, if the program line continues with a part always executed regardless of *condition.*
commands	Commands to be executed. If several command are used, they must be separated with colons (:).

```
2100 IF K=0 THEN PRINT "Zero division" ELSE GOTO 2300
2200 IF K><0 THEN GOTO 2900
2300 IF A<B THEN C=B-A ELSE C=A-B : ENDIF : A=0
     IF A=0 THEN PRINT "A=0" ELSE PRINT "A<>0"
```

Command	Conditional commands structure.
Syntax	IF *condition* THEN [:*commands*] [*command lines*] [ELSEIF *condition* THEN [:*commands*] [*command lines*] [ELSEIF *condition* THEN [:*commands*] [*command lines*] [ELSE [:*commands*] [*command lines*] ENDIF
IF *condition* THEN	When *condition* is true, allows the execution of *command*s and command lines after THEN until next ELSEIF or ELSE command. Program execution then continues from after ENDIF
ELSEIF *condition* THEN	If preceding conditions were false and *condition* is true, allows the execution of *command*s and command lines after next THEN. Program execution then continues from after ENDIF
ELSE	If all preceding conditions were false, allows the execution of *command*s and command lines between ELSE and ENDIF
commands	Commands to be executed. Can be on the same line separated with colons (:)
command lines	Any number of command lines between IF, ELSEIF, ELSE and ENDIF lines.

This command allows programming various case structures. ELSE and ELSEIF commands must be the first commands on the line, whereas ENDIF can be written in the end of the last *command line* if desired. When necessary, IF structures can be nested up to 20 deep.

```
IF A=>0 AND A<10 THEN
   PRINT "A small"
  ELSEIF A<0 THEN
   PRINT "A negative"
  ELSE
   PRINT "A LARGE"
  ENDIF
```

or shorter

```
IF A=>0 AND A<10 THEN : PRINT "A small"
  ELSEIF A<0 THEN : PRINT "A negative"
  ELSE : PRINT "A LARGE" : ENDIF
```

4.8 FOR NEXT

Command	Beginning of repeat loop.
Syntax	FOR *variable=expression1* TO *expression2* [STEP *expression3*]
variable	Loop variable, that gets values according to *expressions* while repeating the loop.
expression1	Value, that variable gets at first round.
expression2	When variable reaches the value of *expression2*, the repeating will be finished.
expression3	If STEP part is used, *expression3* defines increment of variable between every round (default 1).

Command	End of repeat loop.
Syntax	NEXT *variable*

The program part between FOR and NEXT *expressions* will be repeated. For the first round of execution the variable gets the value *expression1*.

After each repeating execution the variable is incremented in the NEXT statement by 1, or by *expression3* if given, until the finishing condition is reached.

If *expression3* is positive the program part will be repeated until *variable* reaches a value greater than *expression2*. For the last execution the value of *variable* is equal to *expression2* or smaller.

If expression3 is negative the program part will be repeated until *variable* reaches a value smaller than *expression2*. For the last execution the value of *variable* is equal to *expression2* or greater.

When repeating is finished the variable keeps the value that it had during the last repetition.

The default value of *expression3* is 1. The maximum number of nested FOR/NEXT loops is 10. The variable used cannot be an array cell.

The repeat structure must always be terminated with a NEXT command which has the same variable as the respective FOR command.

```
FOR I=1.5 TO 4 STEP 1/2
 PRINT I;" ";
 NEXT I

RUN

1.5 2.0 2.5 3.0 3.5 4.0
```

A repeat loop cannot be exited in any other way than through a NEXT command. If repetition needs to be finished without performing all of the rounds, this can be done by setting the value of *variable* to a value greater or equal than e*xpression2-expression3* and by jumping to the NEXT command.

```
100 FOR N=1 TO 100
110 PRINT N
120 IF INP(32) THEN N=100 : GOTO 140
130 OUT(32)=NOT OUT(32)
140 NEXT N
```

Alternatively the NEXT command can be executed elsewhere in the program, for example

```
100 FOR N=1 TO 100
110 PRINT N
120 IF INP(32) THEN N=100 : NEXT N : GOTO 1000
130 OUT(32)=NOT OUT(32)
140 NEXT N
150 END

1000 PRINT "Finished earlier."
1010 END
```

4.9 DO... UNTIL.... LOOP

Command	Repeat loop.
Syntax	DO [*commands*] UNTIL *condition* : LOOP or DO [*commands*] command lines UNTIL *condition1* *command lines* UNTIL *conditionn* LOOP
commands	Commands that are executed in the loop. If several, separated by colons.
command lines	Command lines within the loop
UNTIL *condition*	Exit point from loop. If *condition* is true, program continues from after LOOP command.
LOOP	Point where program execution returns to beginning of loop (DO).

Up to 20 loop commands may be nested in a program.

It is possible to use several UNTIL commands in one LOOP.

```
DO
  A=A+1 : UNTIL A>100
  OUT(32)=NOT OUT(32)
  UNTIL INP(32)=1
  LOOP

DO UNTIL BYTE(#1)=13 : LOOP
DO UNTIL MOVEREADYXY : LOOP
```

4.10 TASK

Command	Create a task. Branches the program execution.
Syntax	TASK *address*[*expression,...*]
address	Starting point of the task to be created
expression,..	Values for variables local in the new task as defined in conjunction with the label starting the new task *address*. Parameter passing is only possible when *address* is a label.

This command allows several tasks to be executed simultaneously. Program execution continues "simultaneously" both beginning from *address*, and continuing from the next line. The new task will have the lowest available free task number, so generally tasks are numbered from 2 on according to the sequence they were created in. However, a new task may get a lower number if a previously created task has been killed leaving its number free.

The system can switch tasks after finishing a command line, or in conjunction with some commands like DELAY, motion commands waiting for space in MOVEBUFFER or serial output commands waiting for free space in buffer.

A task can be killed by an END command. Max. 32 tasks can be run simultaneously. The maximum task number in the program is set by TASKMAX. Default value for TASKMAX is 9, resulting in max. 8 simultaneous tasks.

Function	Current maximum task number.
Syntax	TASK
Type	Integer (2 ... TASKMAX)
Value	Number of task in starting order. The number of the first task (main program) is 2.

The number of the current task can be read using the TASK function. This may sometimes be useful for reserving global resources like numbered timers or device numbers for subroutines that can be called from several tasks.

4.11 TASKMAX

Command	Set the maximum task number.
Syntax	TASKMAX=*n*
n	maximum task number for simultaneous tasks 2....33, default value is 9.

This command sets the maximum task number available for program tasks.

As the number of the first task is 2 (see 4.8 TASK) with TASKMAX=9 the valid task numbers will be 2...9. In case of TASKMAX=33 it is possible to use up to 32 simultaneous tasks.

Increasing maximum number of tasks consumes memory in the system data structures, so it is advisable to set TASKMAX to the value necessary for the application. TASKMAX must be set in the beginning of the program, before any variables with local scope are declared.

Function	Maximum number of tasks.
Syntax	TASKMAX
Type	Integer
Value	By default number of tasks is 8. May change in the range 2...32.

The maximum task number can be read using the TASKMAX function.

4.12 PRIOR

Command	Set the priority of current task.
Syntax	PRIOR=*expression*
expression	Value for priority. Integer 0 ... 127 0 lock to current task 1 .. 127 priority

The task having the smallest PRIOR value gets the most execution time. For non-critical tasks high PRIOR values can be used.

Function	Priority of current task.
Syntax	PRIOR
Type	Integer
Values	0 ... 127

Each task has its own priority (default value is the priority of the task that creates the new task, the original priority of the main program is 3).

In a simple case a task gets execution turn after each n lines, where n is the priority value. If there are several tasks with approximately same priority, the distribution of execution turns operates as described later. Priority can have values between 0 and 127. 0 priority reserves all execution time and thus prevents changing tasks.

Priority can be set in a task as many times as needed. Priority value 0 can be used if the task must not be interrupted by another task.

```
TASK ToDo
FOR I=1 TO 30
PRINT PRIOR; : NEXT I
END

ToDo
PRIOR=2
FOR J=1 TO 30
PRINT PRIOR; : NEXT J
END
```

Switching tasks is based on a queue of executable tasks, where each task waits for its execution turn. Each task has its own so called wait-counter which defines, how many program lines it has to wait for its execution turn. Each line change decrements the wait-counters of the tasks in the queue.

After the wait-counter of the first task is 0, task switching is performed, and the first task in the queue is put in execution. The task ending its execution turn is put back to the queue according to its PRIOR setting so that the value in the wait counters of the tasks before it smaller or equal to the PRIOR setting. The wait-counter of the task is set to its PRIOR value.

5. MATHEMATICS

Mathematical calculations in McBasic 3.2 are performed with 8 byte floating point numbers with a precision of 14 significant numbers. Operating range is 0, $\pm 1.4*10^{-4897}$...$7.1*10^{4896}$.

The operations in the following chapters are listed in calculating order, this means that for an expression, an earlier operation in the list is executed before a latter operation. Arithmetical operations are executed first, comparisons second and logical operations last.

5.1 ARITHMETICAL OPERATIONS

( )	expressions in parenthesis
-	sign
^	exponent
* /	multiplication and division
+ -	addition and subtraction

5.2 LOGICAL OPERATIONS

Comparisons can be done between numerical values or between character strings. Comparisons return a truth value 0 (false) or 1 (true).

A string comparison returns a result using alphabetical order (according to ASCII code) so, that first character in order is a "smaller" string. If the beginnings of the strings are equal, the longer string is "greater".

Logical operations can be done between truth values 0 (false) and 1 (true). In this case the result is also a truth value 0 or 1. Logical operations can also be performed between other values than 0 and 1. In this case the operations are considered binary (see next chapter).

Comparison operations

=	equal to
<	smaller than
>	greater than
<= , =<	smaller than or equal to
=> , >=	greater than or equal to
<>, ><	unequal from

logical operations

NOT	logical negation
AND	logical AND-function
OR	logical OR-function
XOR	logical absolute OR-function

Comparison and logical operations can be combined. Parenthesis can be used to indicate calculation order. For example

```
200 IF A>B AND (INP(32) XOR INP(33)) THEN 300
```

5.3 BINARY OPERATIONS

Binary operations can be used also between other integers than 0 and 1 up to 47 bits as follows.

AND	AND-operation for a 47-bit integer
OR	OR-operation for a 47 bit integer
XOR	XOR-operation for a 47 bit integer

In this case the result is also a max. 47-bit integer. For example

```
PRINT %01010010 AND $0F
2
```

5.4 NUMBER INPUT FORMATS

Numerical values can be entered and programmed in McBasic in several ways.

1 23 -45 0	basic format
1.0 23.4 -0.0656 .77	decimal format
1E0 2.34E1 1E6 -0.2E-17	exponent format
$41 $BFC0 $0020 $100000	hexadecimal format
%11 %01110101 %1111111111	binary format

```
PRINT %1010,$0D,1E3
10          13          1000
```

5.5 MATHEMATICAL FUNCTIONS

5.5.1 ON

Function	Constant 1.
Syntax	ON

Can be used for example as truth value instead of 1.

```
TRACE ON
TRACE NOT ON
100 OUT(45)=ON
```

5.5.2 OFF

Function	Constant 0.
Syntax	OFF

Can be used for example as truth value instead of zero.

```
TRACE OFF
100 OUT(45)=OFF
200 IF INP(35)=OFF THEN 500
```

5.5.3 ABS

Function	Absolute value.
Syntax	ABS(*expression*)
Type	Non-negative real number.
expression	Real number.
Value	Mathematical absolute value of *expression*

```
PRINT ABS(3.14), ABS(-3.14)
3.14      3.14
```

5.5.4 SGN

Function	Sign
Syntax	SGN(*expression*)
Type	Integer
expression	Real number.
Value	1, if *expression* is positive. 0, if *expression* is 0 -1, if *expression* is negative.

```
PRINT SGN(3.14);SGN(-3.14);SGN(0)
1.00 -1.00 0.00
```

5.5.5 INT

Function	Rounding off to the next smaller integer.
Syntax	INT(*expression*)
Type	Integer
expression	Real number.
Value	An integer next smaller or equal integer to *expression*.

```
PRINT INT(3.14);INT(3.9);INT(-3.1)
3.00 3.00 -4.00
```

5.5.6 MIN

Function	Smaller of two numbers.
Syntax	MIN(*expression1*,*expression2*)
Type	Real number.
expression1	Real number.
expression2	Real number.
Value	*expression1*, if *expression1*<=*expression2* *expression2*, if *expression1*>*expression2*

```
PRINT MIN(-1,-0.5),MIN(2,1)
-1.00 1.00
```

5.5.7 MAX

Function	Greater of two numbers.
Syntax	MAX(*expression1*,*expression2*)
Type	Real number.
expression1	Real number.
expression2	Real number.
Value	*expression2*, if *expression1*<=*expression2* *expression1*, if *expression1*>*expression2*

```
PRINT MAX(-1,-0.5),MAX(2,1)
-0.50 2.00
```

5.5.8 RND

Function	Random number.
Syntax	RDN(*expression*)
Type	Real number.
expression	Real number, a seed for random number.
Value	Random real number between 0 ... 1

Expression other than zero sets the seed for random number generator, zero returns the next random number.

```
PRINT RND(7);RND(0);RND(0)
0.89 0.88 0.76
```

5.5.9 EXP

Function	Power of Neper's constant e (2.71828).
Syntax	EXP(*expression*)
Type	Non-negative real number.
expression	Exponent, real number. With values between approx. -81 ... 96 the function value remains within number range.
Value	$e^{expression}$

```
PRINT EXP(0);EXP(1);EXP(1.5)
```

1.00 2.72 4.48

5.5.10 LOG

Function	Natural logarithm.
Syntax	LOG(*expression*)
Type	Real number.
expression	Real number.
Value	ln(*expression*) Natural logarithm of *expression*. The radix of the logarithm is Neper's constant e (2.71828). With negative values of expression the function returns the absolute value of the logarithm of *expression*.

```
PRINT LOG(1);LOG(EXP(1))
```

0.00 1.00

5.5.11 SQR

Function	Square root.
Syntax	SQR(*expression*)
Type	Non-negative real number.
expression	Real number to take square root from.
Value	$\sqrt{expression}$ Square root of expression. With negative values of expression the function returns the absolute value of the square root of expression.

```
PRINT SQR(2);SQR(100);SQR(0.01)
```

1.41 10 0.10

5.5.12 PII

Function	Constant π.
Syntax	PII
Value	π (3.14159.....)

```
PRINT PII,SIN(0.5*PII)
3.14     1.00
```

5.5.13 SIN

Function	Trigonometric sine.
Syntax	SIN(*expression*)
Type	Real number.
expression	Argument of the function in radians (2π radians = 360 degrees).
Value	sin(*expression*)

```
PRINT SIN(0);SIN(1)
0.00 0.84
```

5.5.14 COS

Function	Trigonometric cosine.
Syntax	COS(*expression*)
Type	Real number.
expression	Argument of the function in radians (2π radians = 360 degrees).
Value	cos(*expression*)

```
PRINT COS(0);COS(1)
1.00 0.54
```

5.5.15 TAN

Function	Trigonometric tangent.
Syntax	TAN(*expression*)
Type	Real number.
expression	Argument of the function in radians (2π radians = 360 degrees).
Value	tan(*expression*)

```
PRINT TAN(0);TAN(1)
0.00 1.56
```

5.5.16 ATAN

Function	Trigonometric arctangent.
Syntax	ATAN(*expression*)
Type	Real number.
expression	Argument in radians (2π radians = 360 degrees).
Value	atan(*expression*) Return values between - π/2 ... π/2

```
PRINT ATAN(0);ATAN(1)
0.00 0.78
```

6. STRINGS

A string is an expression consisting of 0..255 characters. Thus a string is essentially a piece of text. String values are assigned in quotes:

```
PRINT "Hello"
```

String variables are variables holding a string value. McBasic string variable names are consist of characters beginning with a letter and followed by any number of letters or numbers and ending with a $-character. Underline characters are also allowed in variable names.

For example

```
A$ B$ c$ SymVariable$ My_string$
```

String variables with single letter names are automatically defined as 80 character long and can be used without declaring them. Other string variables must be declared using the DIM, STRING or STRING(n) statements. The maximum length of a string variable is by default 80 characters. Other lengths can be set using the STRING(n) command to declare 1...255 characters long variables.

```
STRING My_string$, YourString$
DIM String1$
STRING(150) LongString$
```

Arrays of strings can be declared similarly

```
STRING(10) StrArray$(10,50)
```

A string can be combined from substrings with "+"-sign.

```
"Hello "+N$+", how are you"
F$+".TX:D2"
```

A string may contain any characters, also control characters.

```
CtrlString$=CHR$(27)+"Ä101;0X"       'CHR$(27) is esc
```

6.1 EXEC

Command	Execution of a command in a string.
Syntax	EXEC(*string*)
string	String containing an executable McBasic command.

A command in the form of a string is interpreted and executed. The string must consist of a McBasic command without syntax errors.

! Since the EXEC command must interpret the command contained in the *string*, it takes significantly longer than normally to execute a command using EXEC. Thus, it is not advisable to include EXEC commands in programs with critical timing, or in frequently performed loops.

Also, commands where task switching is possible during the command (DELAY, PRINT, MOV...) should be avoided and at least care should be taken not to create a circumstance where task swithcing would occur within EXEC.

```
DO
 INPUT "Enter a command ";A$
 EXEC(A$)
 LOOP

=>
Enter a command ? PRINT 2+3
5
Enter a command ?
```

6.2 ASC

Function	Conversion from character to ASCII code.
Syntax	ASC(*string*)
Type	Integer
string	Usually a *string* of one character. If s*tring* is longer than one character, the character to be converted is the first character from left.
Value	The ASCII code of the first character of the *string* (0 ... 255). Also an empty *string* returns the value 0.

The function returns the ASCII code of the first character in the string. ASC function in the inverse function of CHR$.

```
PRINT ASC("!");ASC("ABC")
```

```
33.00 65.00
```

6.3 LEN

Function	Length of string.
Syntax	LEN(*string*)
Type	Integer
string	String to be measured.
Value	Length of the *string* 0 ... 255 (0 if empty).

LEN returns the current length of the string. Return value can be any integer between 0...255 depending of the contents of the string. However, a string variable always reserves memory according to the declared length of the variable (or 80 bytes by default).

```
PRINT LEN("HELLO"+" AGAIN")
```

```
11.00
```

6.4 VAL

Function	Type conversion. Converts a string containing a numerical value to numerical form.
Syntax	VAL(*string*)
Type	Real number.
string	String to be converted.
Value	Numerical value in the string.

A string can contain a numerical value in some of McBasic numbers entry formats or for example an expression combined from several number formats. For example:

```
PRINT VAL("PII"),VAL("1E2"),
X=PII
Asym$="COS(X)*10+0.01"
PRINT VAL(Asym$)

RUN

3.14      100.00      -9.99
```

VAL function is the inverse function of STR$.

6.5 CHR$

Function	Type conversion. Converts a numerical ASCII code to one character string.
Syntax	CHR$(*expression*)
Type	String
expression	Code to be converted. ASCII code of the desired character (0 ... 255).
Value	String containing one character, where the ASCII code of the character is expression.

If *expression* is 0, function returns an empty string.

CHR$ function can be used for example to print characters using PRINT command or to add any ASCII-character into a string. The value of *expression* must be between 0..255.

CHR$ function is the inverse function of ASC function.

```
PRINT CHR$(33)+CHR$(65)
!A
```

6.6 STR$

Function	Type conversion. Converts a value of numerical expression to string (decimal).
Syntax	STR$(*expression*)
Type	String
expression	Real value to be converted.
Value	String containing the value of the *expression* as printed with PRINT command. DIGITS setting defines the number of decimals in string.

With the STR$ function the numerical value of expression can be converted to a string as it would be printed using PRINT command. This way i.g. numerical data can be formatted using string functions. Number of decimals set by DIGITS command defines the number of decimals in the resulting string. STR$ function is the inverse function of the VAL function.

```
A$=STR$(SQR(2)) : PRINT A$

1.41
10 DIGITS=2
20 PRINT RIGHT$("000"+STR$(PII),5)

RUN

03.14
```

6.7 BIN$

Function	Type conversion. Converts an integer value to a binary string.
Syntax	BIN$(*expression*)
Type	String
expression	Integer value to be converted (-2^{48} ... $2^{48}-1$).
Value	String containing the value of *expression* in binary (48 bit, 2's complement). If *expression* is not an integer, it is rounded off to closest integer. For numbers greater than number range the last 48 binary numbers or a 0 value is returned.

The value of the BIN$ function is a string equivalent to the binary value of expression.

```
PRINT BIN$(9),BIN$(%100000+1)
1001 100001
```

6.8 DEC$

Function	Type conversion. Converts an integer value to a decimal string.
Syntax	DEC$(*expression*)
Type	String
expression	Value to be converted (-10^12 ... 10^13).
Value	String containing the value of *expression* in decimal form. If *expression* is not an integer, it is rounded off to closest integer. For numbers greater than number range values in exponent form are returned.

The value of the DEC$ function is a string equivalent to hexadecimal value of expression.

```
PRINT DEC$($1000),DEC$(%100000)
```

```
4096   32
```

6.9 HEX$

Function	Type conversion. Converts an integer value to a hexadecimal string.
Syntax	HEX$(*expression*)
Type	String
expression	Integer value to be converted (-2^{48} ... $2^{48}-1$).
Value	String containing the value of *expression* in hexadecimal form (48 bit, 2's complement). If *expression* is not an integer, it is rounded off to closest integer. For numbers greater than number range the last twelwe hexadecimal numbers or a 0 value is returned.

The value of the HEX$ function is a string equivalent to hexadecimal value of expression.

```
PRINT HEX$(1000),HEX$($100000+1)
```

```
3E8 100001
```

6.10 LEFT$

Function	Sub-string of a string.
Syntax	LEFT$(*string,expression*)
Type	String
string	String to be divided.
expression	Length of sub-string.
Value	Sub-string of string, length expression characters, taken from beginning of string.

```
PRINT LEFT$("ABCDEFG",3)
```

ABC

6.11 RIGHT$

Function	Sub-string of a string.
Syntax	RIGHT$(*string*,*expression*)
Type	String
string	String to be divided.
expression	Length of sub-string.
Value	Sub-string of *string*, length *expression* characters, taken from end of *string*.

```
PRINT RIGHT$("ABCDEFG",3)
```

EFG

6.12 MID$

Function	Substring of a string.
Syntax	MID$(*string*,*expression1*,*expression2*)
Type	String
string	String from which the subtring is to be taken from.
expression1	The position of the first character of the substring as counted from the beginning of the *string* (1 represents the first character).
expression2	Length of substring.
Value	A substring of *string*, length *expression2* characters, starting from the character in position *expression1*.

```
PRINT MID$("ABCDEFG",2,4)
```

BCDE

6.13 INSTR

Function	Location of substring in string.
Syntax	INSTR(*expression*,*string1*,*string2*)
Type	Integer
expression	Position in *string1*, where search for substring begins. 1 represents the first character position. (Whole string will be searched for).
string1	String, where substring is being searched.
string2	Substring to be searched for.
Value	Position of the first character of sub-string s*tring2* in *string1*. If substring *string2* is not found, value is 0.

```
100 PRINT INSTR(1,"ABCDEFGH","CD")
```

3.00

6.14 STRING

Command	Declare string variables
Syntax	STRING[(*n*)] var$,....
n	Maximum length of a string *n*=1....255. If used without *n*, length will be 80 characters.
var$,...	List of names of string variables to declare separated by commas.

The declared variables can be seen in the structure where they were defined and the structures (subroutines and tasks) under it. This way variables can be defined to have the exact scope desired.

For example, declare a 125 character long string variable Abc$:

```
100 STRING(125) Abc$
```

6.15 UCASE$

Function	Converts string to upper case.
Syntax	UCASE$(*string*)
Type	String
string	String to be converted (up to 255 characters long).

```
100 First$="abcdef"
110 Second$=UCASE$(First$)
120 PRINT First$,Second$
abcdef   ABCDEF
```

6.16 ADDR$

Function	Convert address to string.
Syntax	ADDR$(*address*)
Type	String
address	Address expression

ADDR$ function converts an address expression to a string to allow manipulation and printing values of adress expressions. The result is the line number or the label at the *address* referred to, if either exists. Otherwise the result is an address expression in brackets, consisting of the nearest line number or label before the address plus followed by *+n*, where *n* is the number of lines the *address* is down from the label or linenumber.

example program:

```
'
'
Label 1
 ' program line
 ' program line
Label 2
 ' program line

B>PRINT ADDR$(0)
(+0)
B>PRINT ADDR$(0+2)
Label1
B>PRINT ADDR$(0+3)
(Label1+1)
B>PRINT ADDR$(Label1+3)
Label2
```

7. VARIABLES AND ARRAYS

A variable name is a string of characters of arbitrary length beginning with a letter. Although both upper and lowercase letters are allowed, it is advisable to use an uppercase letter to begin a (long) variable name, followed by some lowercase letters as this makes it easier to distinguish variables from other McBasic reserved words. Variable names are case sensitive, upper and lowercase letters are considered different characters. A variable name can also contain numbers and _ characters. Reserved McBasic function and command names must not be used as variable names. When using variable names beginning with an uppercase letter and continuing with a lowercase letter, McBasic can distinguish them from command and function names.

For example

```
Variable, Piix2, Profile, Velo34_56, Sin(3)
```

Numeric, string or address variables with long names must be declared by one of the following declaration commands:

```
DIM var1,..,string1$,.,varn
REAL var1,...,varn
STRING string1$,....,stringn$
STRING(nn) string1$,....,stringn$
ADDR addr1@,...
```

Additionally, array variables can be declared as

```
BIT array1(a,b..),..
BYTE array1(a,b..),..
WORD array1(a,b..),..
FLOAT array1(a,b..),..
INTEGER array1(a,b..),..
SHORT INTEGER array1(a,b..),..
```

Declaration is not necessary for variables with short names, composed of one letter or a letter and a number, for example

```
A a A3 a9 z8 z Z b B0 B9
```

All of the above names refer to different variables.

The name of a string variable ends with $. The maximum length of a string variable is set by STRING(n) command (default 80 characters)

```
A$ a$ z$ Z$ Symbol$       'names of string variables

STRING Abc$               'defines 80 char. string variable
STRING(125) Abcd$         'defines 125 char. string variable
STRING(1) Abcde$          'defines 1 char. string variable
```

A numerical array name is a string of characters beginning with a letter (following characters may be letters, _ or numbers with dimension range(s) in parenthesis

```
A(0,3) Arr(23) a2(3) z(a(4)) Z(z(2),z(3)) Zspeed_z(5,7)
```

A string array name has a $ before dimension range(s) in parenthesis.

```
A$(2) a$(17) SymArray$(18)
```

Similarly, the name of an address variable ends with @. An address variable may have values formed by a line numbe or a label with an offset if necessary. Address arrays can also be defined to hold address values.

```
A@(15) AddrArray@(20)
```

The scope of a variable or array determines where the variable can be accessed. McBasic 3.2 variables have a scope according to the program structure where they were defined. All variables declared in the beginning of the program, before creating any further tasks or calling subroutines, have a global scope, so they may be referred to from any task or subroutine.

Variables created in other tasks or subroutines are only visible from them and from subroutines called from them or tasks created from them. Variables with this kind of a local scope are created with DIM, STRING, BIT, BYTE, WORD, SHORT INTEGER, INTEGER, REAL, FLOAT and ADDR commands. They have an initial value zero (“”) and disappear when returning from the subroutine or when the task ends. They can have the same name as used in some other scope. The memory space occupied by the disappeared variables is freed and can be used again.

Thus, variables and arrays can be declared as local. In this case they

- Must be declared with appropriate commands in the (beginning of the) task or subroutine, where the local variables are needed.
- If variables are declared local with the LOCAL command they inherit dimensions for arrays, sizes for strings and values which they had (in case there were any) previously.
- In case of declaring with other declaration commands the array dimensions, string lengths and variable values are not inherited.
- In a new task or subroutine all local variables of the creating task or calling subroutine are visible unless redefined.
- Variable is local until END / RETURN command of the corresponding TASK / subroutine.
- On return from the level where variables were local, the values of variables existing at the higher level have been preserved. Variables that did not exist at the higher level disappear.

Arrays are tables of values referenced by the same variable name. An array can have 1 to 7 dimensions. Array names are a strings of characters first of which is a letter followed by other characters or numbers from 0 to 9 or _. As with variable names, short array names of one letter or one letter followed by one number are automatically defined global when used in the program.

Array entries (cells) are being referenced using indexes separated with commas in parenthesis after the array name.

For example ArrSamp(1,3,4,7,8) Block3(1) h(2,7) asize(1,3) K3(5,9) are entries in different arrays.

It is also possible to define an array for strings or addresses. For example SymArray$(3,6) b$(8) are entries in different string arrays and Addr@(5,5) G@(3) are entries in address arrays.

Each entry in an array reserves memory according to the type of the array variable type.

Array size is defined by DIM or type declaration commands before an array is used. In case if numerical array with the name of one letter or one letter and a number is not defined before its use the default dimension of 10 or 10*10 is assumed. An array can be from one to seven dimensional, and the only limit for array size is the size of memory available. An array index can have negative, zero and positive interger values.

For example

```
DIM A(-3..2,2) : STRING(1) LetArray$(32)
LetArray(1)="A"
A(-2,0)=1
A(2,2)=1
```

7.1 DIM

Command	Declare real, string or address variables or arrays, define array size.
Syntax	DIM *arrname*(*expression1*[,..*expressionn*])[, *var*][, *str*$][, *addr@*]..... also DIM *arrname*(*expr1*..*expx2*, ... ,*expr3* .. *expr4*), ...
arrname	Array variable name.
expression1 *expressionn*	max. first index (range 0 to *expression1*) max. last index (range 0 to *expressionn*)
expr1 *expr2* *expr3* *expr4*	First index lower limit First index upper limit (range *expr1* to *expr2*) Last index lower limit Last index upper limit (range *expr3* to *expr4*)
var	Real number variable name
str$	String variable name. String variables declared using DIM are always 80 characters long.
addr@	Address variable name.

With DIM command variables and dimensions for one or more arrays can be defined in one command.

```
10 DIM ArrOne(12),M(5,5),B$(5)
20 FOR I=0 TO 12
30 A(I)=-1
40 NEXT I
50 FOR I=0 TO 5
60 M(I,I)=1
70 NEXT I
```

It is also possible to define arrays with the type declaration commands (see 2.5 VARIABLES TYPES). For example

```
STRING(125) SymArray$(5,5) : REAL Var(5), Var2(5)
WORD Var(5), Var2(5)
INTEGER Var(5), Var2(5) : BYTE Var(5), Var2(5)
BIT Var(5), Var2(5)
SHORT INTEGER Var(5), Var2(5)
```

Array index can take negative, zero or opositive integer values. By default the lower limit is supposed to be 0.

```
DIM SamArr1(-10..10)      ' one dimensional
                          ' array with index from -10 to 10DIM
SamArr1(10..-10)          ' same as in previous line
DIM Sam2(10)              ' one dimension array with index from
                          ' 0 to 10
DIM Sam3(7..1,54..89)     ' two dimension array, first index
                          ' from 1 to 7, second - 54-89
```

7.2 REAL

Command	Declare double precision real (floating point) variables and arrays.
Syntax	REAL *name*[(*expression1,...,expressionn), ...*],..... or for arrays with specified index ranges REAL *name*(*expr1..expr2,...,expr3..expr4), ...*
name	Variable or array names.
expression1 expressionn	max. first index (range 0 to expression1) max. last index (range 0 to expressionn)
expr1 expr2 expr3 expr4	First index lower limit First index upper limit (range expr1 to expr2) Last index lower limit Last index upper limit (range expr3 to expr4)

Real number is the default number format of McBasic and thus both real variables and arrays can be declared. It is possible to declare other format single variables also, but their internal format will always be a real number. This can be done to show intended usage of a variable but declaring a variable as BIT does not actually limit the range of values the variable can get, like when declaring different format arrays. This arrangement has been made to allow as fast operation as possible when using single variables while conserving memory space when using arrays.

Real numbers and array cells can get values from ±1.4E-4897 to ±7.1E4896 with a resolution of 1.4E-14.

```
For example:

REAL MyVar, BigArray(1000,5,3), YearArray(1900...2099)
```

7.3 FLOAT

Command	Declare single precision real (floating point) arrays.
Syntax	FLOAT a*rrname*(*expression1,...,expressionn), ...* *or* FLOAT a*rrname*(*expr1..expr2,...,expr3..expr4), ...*
arrname	Array name.
expression1 *expressionn*	max. first index (range 0 to *expression1*) max. last index (range 0 to *expressionn*)
expr1 *expr2* *expr3* *expr4*	First index lower limit First index upper limit (range *expr1* to *expr2*) Last index lower limit Last index upper limit (range *expr3* to *expr4*)

Single precision floating point array cells can get values from ±1.4E-42 to ±5.0E41 with a resolution of 2.4E-7.

7.4 BIT

Command	Declare bit arrays.
Syntax	BIT a*rrname*(*expression1,...,expressionn), ...* *or* BIT a*rrname*(*expr1..expr2,...,expr3..expr4), ...*
arrname	Array name.
expression1 *expressionn*	max. first index (range 0 to *expression1*) max. last index (range 0 to *expressionn*)
expr1 *expr2* *expr3* *expr4*	First index lower limit First index upper limit (range *expr1* to *expr2*) Last index lower limit Last index upper limit (range *expr3* to *expr4*)

BIT array cells can get values between 0 or 1.

7.5 BYTE

Command	Declare byte arrays.
Syntax	BYTE a*rrname*(*expression1,...,expressionn), ...* *or* BYTE a*rrname*(*expr1..expr2,...,expr3..expr4), ...*
arrname	Array name.
expression1 *expressionn*	max. first index (range 0 to *expression1*) max. last index (range 0 to *expressionn*)
expr1 *expr2* *expr3* *expr4*	First index lower limit First index upper limit (range *expr1* to *expr2*) Last index lower limit Last index upper limit (range *expr3* to *expr4*)

Byte array cells can get integer values between 0 and 255.

7.6 WORD

Command	Declare word arrays.
Syntax	WORD a*rrname*(*expression1,...,expressionn), ...* *or* WORD a*rrname*(*expr1..expr2,...,expr3..expr4), ...*
arrname	Array name.
expression1 *expressionn*	max. first index (range 0 to *expression1*) max. last index (range 0 to *expressionn*)
expr1 *expr2* *expr3* *expr4*	First index lower limit First index upper limit (range *expr1* to *expr2*) Last index lower limit Last index upper limit (range *expr3* to *expr4*)

Word array cells can get integer values between 0 and 65535.

7.7 SHORT INTEGER

Command	Declare short integer arrays.
Syntax	SHORT INTEGER a*rrname*(*expression1,...,expressionn), ...* *or* SHORT INTEGER a*rrname*(*expr1..expr2,...,expr3..expr4), ...*
arrname	Array name.
expression1 *expressionn*	max. first index (range 0 to *expression1*) max. last index (range 0 to *expressionn*)
expr1 *expr2* *expr3* *expr4*	First index lower limit First index upper limit (range *expr1* to *expr2*) Last index lower limit Last index upper limit (range *expr3* to *expr4*)

Short integer array cells can get integer values between -128 and 127.

7.8 INTEGER

Command	Declare integer arrays.
Syntax	INTEGER a*rrname*(*expression1,...,expressionn), ...* *or* INTEGER a*rrname*(*expr1..expr2,...,expr3..expr4), ...*
arrname	Array name.
expression1 *expressionn*	max. first index (range 0 to *expression1*) max. last index (range 0 to *expressionn*)
expr1 *expr2* *expr3* *expr4*	First index lower limit First index upper limit (range *expr1* to *expr2*) Last index lower limit Last index upper limit (range *expr3* to *expr4*)

Integer array cells can get integer values between -32768 and 32767.

7.9 ADDR

Command	Declare address variables and arrays.
Syntax	ADDR *name@*[(*expression1,...,expressionn), ...*],..... or for arrays with specified index ranges REAL *name@*(*expr1..expr2,...,expr3..expr4), ...*
name@	Variable or array names.
expression1 expressionn	max. first index (range 0 to expression1) max. last index (range 0 to expressionn)
expr1 expr2 expr3 expr4	First index lower limit First index upper limit (range expr1 to expr2) Last index lower limit Last index upper limit (range expr3 to expr4)

Address variables and array cells can contain any address values in the program. Thus, the value of an address variable can be a label or a line number existing in the program with an optional offset.

```
For example:
 ADDR Pointer@

StartOfData
 DATA 100,200,300
 DATA 110,210,310
 DATA 120,220,320
EndOfData
```

```
 Pointer@=StartOfData+2
 RESTORE Pointer@
 READ a,b,c
 PRINT A,B,C

RUN

110     210     310

>
```

7.10 LOCAL

Command	Declare variables local
Syntax	LOCAL *varname*, *arrayname*(), *strname*$ ....
varname, arrayname(), *strname*$, *addr@*	Names of variables or arrays to be declared local. Arrays get dimensions according to already existing arrays, giving dimensions is therefore optional.

Declaring variables local create local instances of the given variables. These local instances get a default value equal to the value of the variable before declaring local. Declaring local makes it possible to import values to structures and thereafter alter them while preserving the value of the original variable.

7.11 [LET]

Command	Assign a value for a variable.
Syntax	[LET] variable=*expression*
variable	Numerical or string variable.
expression	New value of variable.

An assign command. LET command is used for assigning a value *expression* to variable. Old value of variable can be used in *expression*. The word LET is optional.

```
      A=123*5+9
      A(1,7)=A(7,1)
      LET A3=A+A(1,7)
      A$="string"+" and so on... "
      A=A+1
GetC  B=BYTE(#1)
      IF B>0 THEN A$=A$+CHR$(B) ELSE GOTO GetC
```

8. FILES AND SERIAL COMMUNICATIONS

Data input and output can be done through serial ports or by reading and writing files.

When operating under McDos operating system every McBasic file and serial port must be opened before use and closed after use unless operations are meant to point to the default port (console, CN:).

Files are referred to with logical device names (device number) after opening them. In case of an error message or when exiting McBasic all open files are automatically closed. Same commands can be used for both files and serial ports.

Ports and files have both logical and physical device names. The syntax for logical device names is #*n*, where *n* is an integer between 1..39. These are also called device numbers.

Physical device names can be opened for any logical name or device number with the OPEN command. Similarly files can be opened for any logical name by giving a physical file name for a logical name. The unopened device numbers refer automatically to the console port (:CN).

Opening and closing input and output files is explained in more detail in part **8.2 FILES AND DATA OUTPUT.**

8.1 PROGRAM FILES

Under McDos operating system the McBasic environment is called from disk or other memory device by giving the name of the interpreter program, for example "BASIC", as a command. The operating system loads and starts the McBasic programming environment, interpreter and compiler stored in the command file such as BAS16.CK. The name of the McBasic command file may vary according to the system type and version used. (See chapter 2.1, McBasic versions).

If a name or a list of names of McBasic program files is given simultaneously with the command, McBasic loads and starts the program(s) automatically after starting itself. The use of the suffix .BA in the names is optional

```
D8:\>BAS16 PROG1,PROG2,PROG3
```

If no program name is given and there is battery secured CMOS memory available, the interpreter searches the McBasic workspace memory for an executable program. If a program is found in the memory, it is started automatically.

If the memory is empty, McBasic looks for a file WAKEUP.BA from the current disk drive and directory. This can also be a virtual drive, such as system flash memory (D6:) or CMOS RAM drive (D7:) set to be the current drive before starting McBasic. McBasic can be started from the current directory or some other directory specified in conjunction with the command or beforehand using the McDOS PATH command.

If no WAKEUP.BA file is found from the current directory, the memory remains empty and McBasic remains in command mode.

Under McDos operating system, start up can be automated by saving a WAKEUP.EX file on disk (or virtual disk). WAKEUP.EX can contain commands such as

```
D8:/BAS300
```

or

```
PATH D0:,D8:/
BASIC PROGRAM
```

The first command loads McBasic version BAS300 (MC300 system) from system eprom D8: and searches for program first in work memory and, if not found, for a WAKEUP.BA file on the current directory, which may be the drive root directory holding the WAKEUP.EX file that controls the starting of the system. The latter command set the search path to include D8: and always loads the program file PROGRAM.BA.

Program execution can be stopped from console (:CN) with a control-X code. This causes the system to output to console information about memory use: program size, variables and compiled size of program as well as size of free memory.

8.1.1 SAVE

Command	Save a program into a file.
Syntax	SAVE *string*
string	Program file name in the form [*device*:]*name*[.*suffix*]

Default suffix is .BA and default device is D0:, in other words the drive, that has been used last. Program is saved with the given file name. If a file with the same name already exists, it is overwritten.

```
SAVE "TEST7"
SAVE "D7:TEST8.BA"
```

8.1.2 LOAD

Command	Load a program from a file.
Syntax	LOAD *string*
string	Program file name in the form [*device*:]*name*[.*suffix*]

Loads a program. Default suffix is .BA and default device is D0:. Previously loaded or written programs and variables are destroyed.

```
LOAD "TEST2"
LOAD "D4:/TEST8"
```

8.1.3 APPEND

Command	Append a program.
Syntax	APPEND *string*
string	Program file name in the form [*device*:]*name*[.*suffix*]

Appends a file into existing program. String is the name of the file to be appended. Default suffix is .BA and default device is D0:.

Append adds the contents of the program file in the end of the already loaded program. If the file to be appended contains numbered lines, they are put in sequence with existing numbered lines. Append command sets all variables to zero.

```
APPEND "NEWPART"
```

8.2 FILES AND DATA OUTPUT

8.2.1 OPEN

Command	Open a file or port.
Syntax	OPEN #*nn*,*string*
nn	Device name (1 ... 39), for which the file or port is opened.
string	Program file name in the form [*device*:]*name*[.*suffix*] default suffix is .TX default device is D0: or device name in the form device:

After opening, files and ports can be read and written with applicable commands and functions. At the time of system start up (before opening files and ports) all device names refer to console CN:.

The most important suffixes :

.BA	BASIC program file
.DT	binary data file
.TX	text file

The most important devices (MC400):

physical name	device
CPU:	
CN:	console port (also MC300)
LP:	second serial port (also MC300)
S0:	third serial port
S1:	fourth serial port
1. auxiliary serial module:	
S2:	auxiliary serial port
S3:	auxiliary serial port
S4:	auxiliary serial port
S5:	auxiliary serial port
2. auxiliary serial unit:	
S6:	auxiliary serial port
S7:	auxiliary serial port
S8:	auxiliary serial port
S9:	auxiliary serial port

MC300 and MC400 systems can have up to 8 physical memory drives D1: .... D8:. The current drive is the default drive and can be referred to as D0:. Details of the memory device (disk drive) setups can be found in the McDOS 2.2 Operating System User's Manual chapter 3.2, Memory devices.

Details of serial port settings can also be found in the McDOS 2.2 Operating System User's Manual chapter 4.26, SET command.

In MC300 systems, a physical device names according to the type of the display (TR: = 320x240 greyscale lcd, ETR: = 640x400 EL) are available for referencing the inbuilt display and keypad/board.

Additionally a device XX: is available for use as a "trash bin" to simulate a non-existing port for example. Any output in XX: is always lost, no input is ever received from XX:.

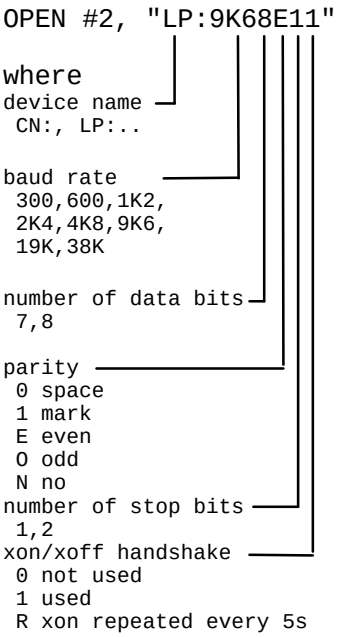
Files are referenced after opening using device numbers. The default device in commands, that do not require a device number, is #1. Console port, in other words the port where the programming terminal is connected, is usually left as #1.

For example :

```
OPEN #2,"LP:"       ' 2. serial port
LIST #2


15 INPUT "Which text file",N$
20 OPEN #3,"D1:"+N$+".TX"     ' disk file
```

When opening serial ports it is possible to set communication parameters by adding the necessary parameters in the control string after the device name.



A port can be locked so that interrupts are disabled and the related buffer is cleared as follows:

```
OPEN #2, "LP:OFF"
```

A port that is OFF will not send or receive characters until opened again.

If no communication parameter is given the operating system uses default values as follows:

baud rate	9600 baud (9K6)
data	8 bits (8)
parity	no parity (N)
stop	1 bit (1)
xon/xoff hand shake	on (1)

8.2.2 CLOSE

Command	Close a file or port.
Syntax	CLOSE #*nn*
nn	Device number (1 ... 39), for which the file or port was opened.

To finish using a file. The file is closed and possible data in buffer is written on disk. When closing ports the communication parameters remain as set, input and output for device number are redirected to CN:.

```
30 CLOSE #3
```

8.2.3 INPUT

Command	Read data from a device.
Syntax	INPUT[#*nn*],[*string*{,|;}]*variable*[,..,*variable*][;]
nn	Device number (1 ... 39), for which file or port was opened. If no device number is given, #1 is assumed (usually =CN:, the console port).
string	If *string* is given, it will be echoed to the device as a prompt.
{,|;}	If *string* is followed by a comma, the echoed prompt will be followed by a question mark, if followed by a semicolon, no question mark is echoed.
variable..	Variables to be read (strings or numbers).
[;]	If expression is followed by semi-colon, no line feed (cr+lf) is echoed after reading from a port.

McBasic program stops when an INPUT command is encountered until necessary data is received from device nn. When reading from serial ports (terminal etc.) this may stop program execution for a relatively long time.

Because of this, INPUT command can only be used for reading from files in cases where the control system should perform other operations or TASKs simultaneously. In these cases input data from serial ports should be read using the BYTE(#n) function.

```
50 INPUT #4,X,Y,A$
60 INPUT #3,A$
65 ' semi-colon at the end
66 ' prevents line feed
70 INPUT "ENTER A NUMBER",N;
75 PRINT " number was ";N
78 ' semi-colon at the end
79 ' prevents echoing question mark
80 INPUT "ENTER ANOTHER NUMBER";N
85 PRINT N
RUN

ENTER A NUMBER? 3+2 number was 5.00
ENTER ANOTHER NUMBER 4
4.00
```

8.2.4 PRINT

Command	Output to a device.
Syntax	PRINT[#*nn*,][*expression*][{,|;}...{,|;}*expression*][,|;]
nn	Device number (1 ... 39), for which the file or port has been opened. If no device number is given, #1 is assumed (usually CN:, the console).
expression	Numerical or string *expressions* to be output. Without *expressions* the command can be used for line feed.
{,|;}	A comma used between *expressions* sets the next output to next column. Each column is 8 characters wide. A semi-colon used between *expressions* sets the next output right next to the previous output.
[,|;]	PRINT command performs an automatic line feed after output. If the last *expression* is followed by a comma or a semi-colon, no line feed (cr+lf) is printed. If the character is comma the cursor is tabulated to the next column.

When printing strings the output can be formatted using string functions and expressions can be combined by + sign.

```
PRINT "ABC"+CHR$(10)+"DEF"      'ASCII 10 is
                                 'line feed

ABC
DEF
```

When several tasks use the same output device it is necessary to control the output so that different tasks do not print simultaneously as printed sequences might get mixed. This may be accomplished by using flag variables to time the operation of tasks or by locking tasks with the PRIOR setting.

8.2.5 LIST

Command	List whole program.
Syntax	LIST [#*nn*]
nn	By giving device number nn program will be listed to device #nn, otherwise to device #1.

Command	List part of program.
Syntax	LIST [#nn,][*address1*][,[*address2*]]
nn	By giving device number *nn* program will be listed to device #*nn*, otherwise to device #1.
address1	The first line to be listed. Default is linenumber=1 or start of program.
address2	The last line to be listed. Default is linenumber=65535 or end of program. If only comma is given, program will be listed until next empty line.

LIST has following features:

- automatic nesting for commands FOR...NEXT, IF..THEN/ ELSEIF/ ELSE/ ENDIF, DO...LOOP;
- removes spaces from before ' on comment lines.
- in case the comment ends with ‘ the comment will be right aligned with a leader consisting of characters similar to the character after the ' starting the comment.

```
LIST              'whole program to console
LIST #2           'whole  program to #2
LIST 1000         'beginning from line 1000
LIST 1000,1000    'only line 1000
LIST Lab1,Lab2    'from line Lab1 to line Lab2 (including)
LIST Lab1,        'from line Lab1 to the next empty line
LIST #2,1000,1990 'lines 1000-1990 to #2
LIST #2 100,      'from line 100 to next empty line
```

8.2.6 DIGITS

Command	Set the number of decimals used when printing.
Syntax	DIGITS=*expression*
expression	Number of decimals (0...9) to be printed in PRINT command when printing numerical values. A value 0.5 can be added to prevent exponent notation when printing small values.

Number of decimals used in printing is defined by the value of expression. Default number of decimals when starting a McBasic program is 2 decimals.

Each task has its own setting for DIGITS so changing DIGITS setting in one task does not affect printing formats in other tasks.

```
DIGITS=3     'print with 3 decimal places
DIGITS=2.5   'print with 2 decimal places and suppress exponent
             'format
```

```
4000 FOR I=0 TO 9
4005 DIGITS=I
4010 PRINT I;TAB(12);PII
4020 NEXT I
```

```
0          3
1.0        3.1
2.00       3.14
3.000      3.141
4.0000     3.1415
5.00000    3.14159
6.000000   3.141590
7.0000000  3.1415900
8.00000000 3.14159000
9.0000000003.141590000
```

8.2.7 LINE

Command	Set length of output line.
Syntax	LINE(#*nn*)=*expression*
nn	The number of the device whose line length is set.
expression	Length of line (integer 0...255). When set to 0 no automatic line feed is performed (default). Other values cause automatic line feed (cr+lf) after expression characters have been output.

```
500 LINE(#1)=132
```

8.2.8 BYTE(#nn)

Command	Output a byte to a device.
Syntax	BYTE(#*nn*)=*expression*
nn	Device number. Can also be a variable or expression.
expression	Value to be output, integer (0 ... 255, ASCII code).

Function	Read a byte from a device.
Syntax	BYTE(#*nn*)
Type	Integer (0 .. 255, ASCII code)
nn	Device number to read from. Can also be a variable or expression.
Value	ASCII code (0 ... 255) of the received character. If there are no characters in the buffer or the file has no more characters, function returns value -1.

When writing to or reading from a file, the file pointer PTR(#nn) is automatically incremented by 1.

When writing numerical values to files using BYTE, WORD, FLOAT, REAL or IEEE.. commands, they are saved in binary mode and therefore can not be inspected or edited for example with a text editor. When reading the file it is advisable to use BYTE, WORD, FLOAT, REAL and IEEE.. functions.

```
120 BYTE(#3)=128
190 A=BYTE(#4) : IF A<>65 THEN 190
```

8.2.9 WORD(#nn)

Command	Write a word (2 bytes) to a device.
Syntax	WORD(#*nn*)=*expression*
nn	Device number. Can also be a variable or an expression.
expression	Value to be written, integer (0 ... 65535).

Function	Read a word from a device.
Syntax	WORD(#*nn*)
Type	Integer (0 .. 65535)
nn	Device number to be read from. Can also be a variable or an expression.
Value	Value (0 ... 65535) of the received word. If there were not 2 bytes available in device buffer or the file did not contain 2 more characters, the function returns value -1.

When writing to or reading from a file the file pointer PTR(#nn) is automatically incremented by 2.

Because WORD requires 2 characters ready for reading, it can mainly be used with files. When using serial ports, the use of BYTE function is recommended.

See BYTE(#nn).

```
130 WORD(#3)=64000
200 P=WORD(#4)
```

8.2.10 FLOAT(#nn)

Command	Write a floating point number (4 bytes) to a device.
Syntax	FLOAT(#*nn*)=*expression*
nn	Device number to be read from. Can also be a variable or an expression.
expression	Value to be written, real number.

Function	Read a floating point number (4 bytes) from a device.
Syntax	FLOAT(#*nn*)
Type	Real number
nn	Device number to be read from. Can also be a variable or an expression.
Value	Received real number. If there were not 4 bytes available in device buffer or the file did not contain 4 more characters, the function returns 0.

When writing to or reading from a file, the file pointer PTR(#nn) is automatically incremented by 4.

Because FLOAT requires 4 character ready for reading, it can mainly be used with files. When using serial ports, the use of BYTE function is recommended.

See BYTE(#nn).

```
130 FLOAT(#3)=1000*PII
200 P=FLOAT(#4)
```

8.2.11 REAL(#nn)

Command	Write a floating point number (8 bytes) to a device.
Syntax	REAL(#*nn*)=*expression*
nn	Device number to be read from. Can also be a variable or an expression.
expression	Value to be written, double precision real number.

Function	Read a floating point number (8 bytes) from a device.
Syntax	REAL(#*nn*)
Type	Real number
nn	Device number to be read from. Can also be a variable or an expression.
Value	Received real number. If there were not 8 bytes available in device buffer or the file did not contain 8 more characters, the function returns 0.

When writing to or reading from a file, the file pointer PTR(#nn) is automatically incremented by 8.

Because REAL requires 8 character ready for reading, it can mainly be used with files. When using serial ports, the use of BYTE function is recommended.

8.2.12 IEEE

Command	Write a IEEE format floating point number (4 or 8 bytes) to a device.
Syntax	IEEE*nn*[I](#*devicenr*)=expression
nn	bits in format, 32 or 64
I	PC style format indicator. If omitted, unix style is assumed
devicenr	Device number to be read from. Can also be a variable or an expression.
expression	Value to be written, real number.

Function	Read a IEEE format floating point number (4 or 8 bytes) from a device.
Syntax	IEEEnn[I](#*nn*)
nn	bits in format, 32 or 64
I	PC style format indicator. If omitted, unix style is assumed
Type	Real number
nn	Device number to be read from. Can also be a variable or an expression.
Value	Received real number. If specified number of bytes was not available from device buffer, the function returns 0.

IEEE format read/write can be used for floating point data exchange between Arlacon MC and other systems.

8.2.13 DATE$

Command	Set file date.
Syntax	DATE$(#*nn*)=*string*
nn	Device number, for which the file has been opened.
string	New date in form *yy*[*mmdd*[*hhmmss*]] *yy* = year 80..79 (80 = 1980, 79 = 2079) *mm* = month 01..12 (01 = January) *dd* = day 01..31 *hh* = hours 00 23 *mm* = minutes 00..59 *ss* = seconds 00..59 or alternatively *yyyy*[*mmdd*[*hhmmss*]] where *yyyy* = year (0000 ..... 9999) other items as above

Function	Read file date or convert date in number format to string.
Syntax	DATE$(#*nn*) or DATE$(*a*)
Type	String
nn	Device number, for which the file has been opened.
a	Date in number format (as produced by DATE function) to be converted to string.
Value	File or system date in form *yymmddhhmmss* (see above) Converted number format date in form *yyyymmddhhmmss* (4 digit year).

Device CN: (console), usually #1, is the real time clock of the system (see chapter 9. REAL TIME CLOCK.)

Disk files are automatically dated according to the console date when written to a disk.

For example string 900224123456 represents the date February 02, 1990 and the time 12:34:56

```
DATE(#1)="900224123456"
PRINT DATE$(#1)
```

900224123456

DATE$ function also provides means to convert date calculation results from number format back to string format and to convert 2 digit year formats to 4 digits.

```
Example:

PRINT DATE$(DATE(DATE$(#1)))
19990825135344
```

8.2.14 DATE

Function	Convert date string to number.
Syntax	DATE(*date*)
Type	Real number
date	Date to convert in form [*yy*]*yy*[*mmdd*][*hhmmss*]
Value	A number representing the date in days from 2.1.2000. Dates before 2.1.2000 are negative numbers and after 2.1.2000 positive numbers.

The DATE function allows calculations with dates. The date 2.1.2000 (first Sunday of year 2000) has been chosen to be the "zero" date. DATE gives the difference from this date in days, so the integer part of the value represents full days while the fractional part tells the time. Thus time and date differences can be calculated.

For example there are 236 days between 1.1.1999 and 25.8.1999 whereas there are 237 days between 1.1.2000 and 25.8.2000 (2000 is a leap year):

```
PRINT DATE("990825")-DATE("990101"), DATE("000825")-DATE("000101")
236.00              237.00
```

DATE also allows week day calculations (0=Sunday, 1=Monday ...). 25.8.1999 is Wednesday:

```
X$="990825"
PRINT INT(DATE(X$)-7*INT(DATE(X$/7))
3.00
```

8.2.15 PTR

Command	Set a file pointer.
Syntax	PTR(#*nn*)=*expression*
nn	Device number, for which the file has been opened.
expression	New value of pointer. Value 0 points to the first byte of file, SIZE(#nn)-1 points to the last byte of file.

Function	Read a file pointer.
Syntax	PTR(*#nn*)
Type	Integer
nn	Device number, for which the file has been opened.
Value	Value of file pointer.

The value of the file pointer must be positive and smaller than or equal to the size of free memory in the device.

The file pointer of a newly opened file is 0.

This command allows reading data from any part of a file and thus use the file as a random access file.

```
 5 OPEN#3,"FILE.DT:D1"
10 PTR(#3)=0          'first character
20 A=BYTE(#3)         'is read to A
30 PTR(#3)=SIZE(#3)-1 'last character
40 B=BYTE(#3)         'is read to B
200 IF PTR(#3)>=SIZE(#3) THEN STOP
```

If the value of PTR is set greater than the current size of the file (see SIZE), the file size is automatically increased accordingly. The new empty space at the end of the file is not set to zero, so some data that has been deleted before, may become readable.

8.2.16 SIZE

Command	Set file size.
Syntax	SIZE(*#nn*)=*expression*
nn	Device number, for which the file has been opened.
expression	New size (bytes).

Function	Read the file size.
Syntax	SIZE(*#nn*)
Type	Integer.
nn	Device number, for which the file has been opened.
Value	Size of file (bytes). When using for a serial port the function returns the size of the free space in output buffer.

Value of the file size must be smaller than or equal to the size of free memory in the device..

This command can be used for example to destroy a file or part of a file. If the size of the file is set to zero and the file is closed, the file will be removed from disk.

Output to a port with too little free space in the buffer can be avoided using the SIZE function thus avoiding interrupting the program and task changing.

```
SIZE(#3)=PTR(#3)       'let's destroy a file
                       'beginning from PTR
SIZE(#4)=0 : CLOSE(#4) 'delete the file

OPEN#4,"FILE.TX"
IF SIZE(#4)=0 THEN PRINT "File not found"

'sub-routine, prints M$ to #2 when enough space
DO WHILE SIZE(#2)<250 : LOOP
PRINT#2,M$
RETURN
```

8.2.17 DIR$

Function	Read a directory entry.
Syntax	DIR$(*#nn,expression*)
Type	String
nn	Device number, for which the memory device/directory has been opened.
expression	Number of directory entry.
Value	Contents of the directory entry in form "nnnnnnnnsss" (11 characters). nnnnnnnn file name sss suffix A name of a subdirectory is an entry in form "nnnnnnnnsss/" (12 characters). Directory entries are filled from position 0. The first empty entry indicates that the rest of the directory is empty. If the directory entry is empty, the function returns an empty string "".

Maximum number of files on one memory device is dependent on the type of device. When reading the directory cells it is possible for example to list the directory in desired order and format.

```
'DIR
OPEN #2,"D1:"
n=0
DO
 A$=DIR$(#2,n)
 UNTIL A$=""
 IF LEN(A$)=12 THEN
  PRINT LEFT$(A$,8)+" "+RIGHT$(A$,4)+"   ";
 ELSE
  PRINT LEFT$(A$,8)+"."+RIGHT$(A$,3)+"    ";
 ENDIF
 C=C+1
 IF C=5 THEN C=0 : PRINT
 LOOP

RUN
```

```
SUBDIR      /   DISK    .CM     MEM     .CM     DELETE  .CM     STRIP   .BA
PROM    .CM     CMP     .BA     COMGEN  .CM     DOBACKUP.EX     TX      .CM
```

```
EXECUTES.TX     LF      .EX     WAKEUP  .EX     VD011   .EX     BTDDD51 .DT
BTDDD01 .DT     BTD0011 .DT     PAL     .CM     FOR27513.EX     LILBUGHX.DT
FOR27011.EX     NOZZLE  .DT     PIP4    .BA     BTDD051 .DT     MCDOS   .CM
LIST    .BA     PIP5    .BA     LINES   .BA     E       .EX     README  .TX
P513    .EX     UPLOAD  .EX     NOLF    .EX     DIR     .CM     ASM     .CM
XDISK   .CM     EP      .CM     COPY    .CM     DATA    .CM     DEL     .CM
BTD0051 .DT     COLUMBIA.DT     DOPROM  .EX     LOAD    .EX     BOOTS   .TX
VD513   .EX     BTDDDD1 .DT     VDEMO   .BA     DIR     .BA     SETCLOCK.BA
P513T   .EX
```

8.2.18 TAB

Function	Set cursor on output line in PRINT command.
Syntax	TAB(*column*)
column	Column, where the next output is desired.

Set cursor on output line (1 ... 255). Used only with PRINT command. Moves cursor to desired column. New column must higher than the current position of cursor.

```
350 PRINT X,Y,TAB(20+65*Y);"*"
```

8.2.19 CURS$(column,row)

Function	Set cursor position on display.
Syntax	CURS$(*column,row*)
Type	String
column	Column, where the next output is desired 0 leftmost column on screen
row	Row, where the next output is desired 0 topmost line on screen
Value	Cursor control sequence for Arlacon terminals.

Text output can be directed to desired position on a display screen with this function. Coordinate range depend on the terminal type and settings used. Function can be used for example when printing with PRINT command.

```
5 PRINT CURS$(40,12);"X: ";POSX;"    ";
```

When cursor control is based on CURS$ or other control sequences, TAB function cannot be used.

8.2.20 LINK

Command	Link input/output of data to two devices.
Syntax	LINK#*n1,n2,n3*
n1	Device number to which other devices are linked.
n2	Number of the first linked device
n3	Number of the second linked device

Printing to a device number *n1* linked to two other device numbers copies the output to both devices. Reading from *n1* reads data from either linked device if available.

It is possible to link more that 2 devices to one device number by using more than one level of links. Max. 30 links can be used simultaneously.

To break a link CLOSE#*n1* command is used. It closes *#n1*, but not the devices linked to it. Devices linked to another device number can be used also directly to their own device numbers. Several links can also lead to one device.

```
OPEN#2,"S2:"
OPEN#3,"S3:"
OPEN#7,"D7:RECORD.TX"
LINK#10,2,3      'link channels #2 and #3 to #10
LINK#11,10,7     'link channels #10 and #7 to #11
PRINT#11,"START" 'print to #2, #3 and #7

DO: B=BYTE(#10)  'read from #2 and #3
UNTIL B<0 : LOOP 'if channels #2 and #3 are empty, loop

CLOSE #3 : OPEN #3, "CN:" 'change of device in channel #3
PRINT #11,"STOP"
CLOSE #10 : CLOSE #11          'break links #10 and #11
CLOSE #2 : CLOSE #3 : CLOSE #7 'closing primary channels
```

8.3 GRAPHICS CONTROL FUNCTIONS

Graphics control functions can be used for controlling graphics on ARLACON ETR and CGA terminals and screens. These McBasic functions help to generate automatically the necessary PLOT-10 control sequences required by terminals.

8.3.1 TEXT$

Function	Display control to text mode.
Syntax	TEXT$
Type	String
Value	Control sequence for terminal.

TEXT$ sets display to text mode. For example

```
10 PRINT TEXT$;
```

8.3.2 PNT$

Function	Define a point in graphics.
Syntax	PNT$(x,y)
Type	String
x	X (horizontal) coordinate. 0 is the left side of the screen, 1 is the right side of the screen.
y	Y (vertical) coordinate. 0 is the bottom of the screen, 1 is the top of the screen.
Value	Control sequence for terminal.

This function is used for entering points in connection with drawing functions. The function defines a point on screen so that the x defines the horizontal position and the y defines the vertical position.

Further information about the use of PNT$ function in paragraphs LINE$ and FILL$.

8.3.3 LINE$

Function	Set display to line draw mode.
Syntax	LINE$
Type	String
Value	Control sequence for output.

LINE$ sets a graphics display to line draw mode. For example

```
PRINT LINE$;
```

In the line draw mode jointed lines can be drawn with current color settings by giving points with PNT$ function. For example

```
PRINT COLOR$(1)+LINE$+PNT$(0,0)+PNT$(1,0);
PRINT PNT$(1,1)+PNT$(0,1)+PNT$(0,0)+TEXT$
```

draws a frame on screen. In the line draw mode the PNT$ function is the only available function. The color changes must be done in the text mode. The start point of a jointed line is defined by the first PNT$ function and the end point with the last PNT$ function.

The TEXT$ function is used to return to text mode.

8.3.4 FILL$

Function	Set display to fill mode.
Syntax	FILL$
Type	String
Value	Control sequence for output.

FILL$ sets a graphics display to fill mode. In this mode the drawn jointed lines are automatically closed with a straight line and filled with the current color. Otherwise FILL$ operates as LINE$.

An area that has been filled with FILL$ function can be surrounded with a frame of different color by drawing the frame with LINE$ command using the same points.

8.3.5 COLOR$

Function	Color definition.
Syntax	COLOR$(*color*)
Type	String
color	Color code for desired color. Available values depend on the display type.
Value	Control sequence for output.

Color definition. Sets both the text color and the line color as well as the fill color for an area. This function operates only in the text mode.

```
45 PRINT COLOR$(1);    'set color 1
```

For example in CGA02804 color display cause the following color numbers are available:

operation of LINE$ and FILL$

0	draw with the background color
1	draw with the foreground color
2	draw with shifted foreground and background color
3	draw with the complementary colors
4-7	halftone, only the background
8-11	halftone, only the foreground
12-15	halftone, both the background and the foreground
16	background black
17	background blue
18	background red
19	background magenta
20	background green
21	background cyan
22	background yellow
23	background white
32	foreground black
33	foreground blue
34	foreground red

35	foreground magenta
36	foreground green
37	foreground cyan
38	foreground yellow
39	foreground white

Halftones:

```
 4 -=--      5 -=--      6 -=-=      7 -===
    ----        ---x        =-=-        ==-=
 8 =*==     9 =*==     10 =*=*     11 =***
   ====        ===*        *=*=        **=*
12 -*--     13 -*--     14  -*-*     15 -***
     ----        ---*        *-*-        **-*
```

- background
= not changed
* foreground

The background color is the screen color behind the text and the foreground color is the color of the text itself when printing the text to screen. When screen is cleared with a code such as CHR$(26) the whole background of the screen is changed to the last defined background color. Further display control sequences are discussed in respective display device documentation.

9. TIMING AND REAL TIME CLOCK

If the system is equipped with a real time clock, all the files are dated automatically according to the current time of the real time clock. The real time clock is set to current time by setting the console date. The date and time can also be read from console (:CN, usually #1).

9.1 REAL TIME CLOCK

Clock is set with command:

Command	Set date/time of the real time clock.
Syntax	DATE$(#1)=*string*
string	The new date in form *yy*[*mmdd*[*hhmmss*]] *yy* = year 80..79 (80 = 1980, 79 = 2079) *mm* = month 01..12 (01 = January) *dd* = day 01..31 *hh* = hours 00 23 *mm* = minutes 00..59 *ss* = seconds 00..59 or alternatively *yyyy*[*mmdd*[*hhmmss*]] where *yyyy* = year (0000 ..... 9999) other items as above

```
100 DATE$(#1)="900224123456"
```

sets the date to 24/02 1990 and the time to 12:34:56.

Function	Read the real time clock date/time.
Syntax	DATE$(#1)
Type	String
Value	The current date and time in form *yymmddhhmmss* as described above.

Note that short/long date format conversions and date calculations are possible using the DATE$() and DATE() conversion functions described in chapters 8.2.13.-14.

```
100 PRINT DATE$(#1)
900919161354
```

```
100 D$=DATE$(#1)
110 PRINT MID$(D$,5,2)+"."+MID$(D$,3,2)+".19";
120 PRINT MID$(D$,1,2)+"  at ";
130 PRINT MID$(D$,7,2)+":"+MID$(D$,9,2);
140 PRINT " o'clock"

19.09.1990  at 16:13 o'clock
```

9.2 TIME MEASUMENTS

9.2.1 TIMER

Command	Set a timer.
Syntax	TIMER[(*expression1*)]=*expression2*
expression1	Timer number 0 ... 99 If used without timer number, refers to TIMER local for each task.
expression2	Value to set [s] 0 ...2.1E9

The resolution of a timer is <1µs. The maximum value to set is 2.1E9 seconds (about 68years).

There are 100 global timers available in McBasic. Additionally, there is an unnumbered local timer for each task. Normally timer value is 0 when read. If a timer needs to be started, the timer must be set to a positive non-zero value. This causes the timer to start counting downwards according to time.

Timers can be used, for example, for generating delays and for measuring time intervals.

```
1000 TIMER(0)=5
```

Function	Read a timer.
Syntax	TIMER[(*expression*)]
Type	Real number
expression	Timer number 0 ... 99. If used without timer number, refers to TIMER local for each task.
Value	Status of a timer, remaining time [s]. If timer has stopped, value is 0.

The status of a timer can be read with this function.

For example measuring the execution time of the subroutine that begins from line 400:

```
TIMER(3)=1000
GOSUB Delay1
PRINT 1000-TIMER(3)
```

For example to generate a delay of 3,5 seconds:

```
Delay1 TIMER(0)=3.5
 DO UNTIL TIMER(0)=0 : LOOP
 RETURN
```

9.2.2 DELAY

Command	A delay.
Syntax	DELAY *expression*
expression	Time to wait [s]. Maximum delay is 2.1E9 seconds (about 68 years).

With DELAY command a delay can be generated using only one command. DELAY is independent of timers.

Each of the simultaneous tasks started with TASK command have DELAY systems of their own. So a delay in one task does not affect execution of another TASK. When more than one task is being used, DELAY automatically passed the control to the next task in queue until the specified delay has elapsed.

For example to list the program after 20 seconds to give the user time to connect a printer to console port before listing begins.

```
DELAY 20 : LIST
```

10. OTHER COMMANDS

10.1 DATA LINES

Data lines form a program structure for defining data in the program. Data can contain numerical, string or address data. Also expressions can be used as data.

10.1.1 DATA

Command	DATA definition.
Syntax	DATA *expression*,....,*expression*
expression	Data entry. Expressions can be numerical, string or address data.

Data definitions of DATA expressions can be read during the program execution with READ command.

DATA expressions can be located in any part of program.

```
100 DATA 10,13,15,"ARLACON MOTION CONTROLS","MANUFACTURER"
110 DATA 3,2,7,"AUTOMATIC MACHINE LTD","CLIENT"
```

10.1.2 READ

Command	Read data from DATA lines.
Syntax	READ *variable*,..,*variable*
variable	Variables, where the data is read to. Types of variables must correspond to the types of expressions in DATA lines.

Reading of data begins from the DATA line and variable where the read pointer is. If READ command has not been used before, reading of data begins from the first DATA line in the program.

The division of the variables in DATA lines and the length of DATA lines are not significant. The data is read from DATA expressions with READ command in the order in which the data is encountered in DATA lines.

```
DATA 120+3," HELLO",34.567,LEN(A$)
READ A,A$
PRINT A;A$,
READ A,B
PRINT A,B
```

```
123.00 HELLO  34.56      6
```

10.1.3 RESTORE

Command	Set the read pointer in DATA expressions.
Syntax	RESTORE [*address*]
address	If *address* is given, the read pointer is set to the beginning of the line at it If *address* is not given, the read pointer is set to the beginning of the first DATA line in program.

This command can be used for pointing the DATA line where the next READ command starts to read the data.

If no RESTORE command is used in a program, the read pointer is set to the beginning of the first DATA line when the program starts.

Each task has an own read pointer for DATA lines so reading DATA lines or using RESTORE command in one task does not affect other tasks.

```
  DATA 1
Data2 DATA 2,3
  RESTORE
  READ A1,A2
  RESTORE Data2
  READ B2,B3
```

10.1.4 DATAPTR@

Function	Read current data pointer
Syntax	DATAPTR@
Value	Address of the DATA line where READ will read data next. If no DATA lines exist or pointer is past them, value is: End of program.

DATAPTR@ function allows reading the current status of the data pointer in the current task.

10.2 USER DEFINED FUNCTIONS

Often used expressions can be defined as user functions using the following commands. This will conserve memory and make programs more efficient, understandable and easier to modify.

10.2.1 DEF

Command	Define a user function.
Syntax	DEF FN*name*[$|@][(*var1*[,..,*varn*])]=*expression*
name	Function identification name. Name can be any lenght and always starts with FN. Letters, numbers or_.can be used in *name.*
[$|@]	If the value of *expression* is a string or an address, a $ or @ -character must be added to the name of user function respectively (string or address function).
var1 ... varn	Internal variables of a user function. (0 .. 8 pcs).
expression	Definition of the value returned by the user function. Both internal and global variables as well as all the McBasic operators and functions can be used (also previously defined user functions).

If function identification name is followed by $ or @, the user function is a string or address function and returns a string or address value respectively. Otherwise return values of user functions are numerical.

Maximum number of internal variables in a user function is eight. In addition to these variables all McBasic global variables can be used. Internal variables (*var1 ... varn*) of a user function are declared automatically local. String parameters are 80 characters long and numerical are of the REAL type.

```
DEF FNCasd(X,X$)=ASC(MID$(X$,X))
A$="ABCDEFG"
N=3
PRINT FNCasd(N+1,A$)
```

```
68.00
```

10.2.2 FN*name*

Function	A user defined function.
Syntax	FN*name*[$|@][(*expression1*[,..,*expressionN*])]
name	function identification name as in DEF FN*name*.
[$|@]	$ or @ character indicates a string or address function returning a string or address value respectively.
expression1 ... expressionN	Arguments of function. 0 .. 8 pcs numerical and/or string or address expressions according to the definition in DEF command.

Call a user function. *Expression1 .. expressionN* are internal variables (arguments) of function that must be given when the function is called. Also values of external variables (instances valid in the structure where function is used) that were used in the user function definition affect the value of user function. If no internal variables have been defined for a user function, it is not necessary to give any arguments when calling the function.

Note: The user function must be defined before it is called in a program.

```
DEF FNS(X,Y)=X+Y
PRINT FNS(3-1,2*3)
```

```
 8.00
```

```
DEF FNX=POSX-SIN(POSY)
PRINT FNX
```

10.3 COMMENTS

10.3.1 REM

Command	Comment line.
Syntax	REM *text*
text	Comments, can be any text.

Comment. This command is used for writing informative text between program lines in order to make the program more readable. Comments do not affect the execution of a program.

```
REM this line is a comment
A=3 : REM this comment also works
```

10.3.2 '

Command	Comment line.
Syntax	' text
text	Comments, can be any text.

Comment. An alternative command for REM command. In addition comments separated by ' sign can also be written after other commands without ":" -separator.

```
' this line is comment
A=A+1  ' also this comment works
```

11. MOTION CONTROL

The control software for servo motors runs continuously in background of the McBasic environment. The axes positions are controlled by PID algorithms with separately adjustable parameters for each axis. The common refresh rate of the algorithms can be set with the PIDFREQ= command.

Motion commands initialize the execution of the desired motion and program execution can continue immediately simultaneously with the motion. The system the takes care of the performing of the motion in background. This way, the motion commands in the program actually only start motion and do not represent the performing of the whole motion.

The available axes are labelled in two ways. The first 10 axes in the system have letter names X,Y,Z,W,A,B,C,D,T and U. Axes can also be referred to with numbers starting from 0. The number of axes X-U are 0 to 9 respectively. For all motion control commands and functions, two alternate syntaxes for letter named and numerically referenced are available. Axes >9 have no letter names and can only be referred to by their number.

Motion commands can be issued for a desired number of axes simultaneously. Thus, commands such as

```
ACCELXYZ=10 : ACCEL(2)=20 : ACCEL(2,4,6)=30
```

are valid.

For some motion functions producing a single value only one axis can be used as argument. For example to read current accelerations for axes 2 and 4 functions ACCEL(2) and ACCEL(4) should be used. However, some functions, such as MOVEBUFFER(*n1,n2*) or MOVEBUFFERXYZ and MOVEREADY(*n1,n2*) can also refer to the status of a combination of axes.

In combined motion commands (XYZ..) or (0:*targ1*,1:*targ2*,2:*targ3*,...) the axes move at speeds resulting in simultaneous completion of the translation. With cartesian mechanisms this corresponds to a straight line from starting point to end point (linear interpolation). With separate motion commands X,Y,Z,.. or (0),(1),(2),.. axes can be controlled independently.

There are separate commands for control of continuous movement. Parameter settings of PID algorithms have immediate effect on control. Speed and acceleration/deceleration settings affect the next translation commands. However acceleration/deceleration has immediate effect for speed controlled motion (CREEP).

Commands with no axis reference can be used to set some parameters for axes 0..9 (axes with letter names). For example SPEED=100 affects axes 0...9, but SPEEDXZ=50 affects only X and Z axes (0 and 2). SPEED(13)=100 affects only axis number 13. Accelerations and speeds of combined axes motion commands are defined so that limitations (for speed and acceleration) for all axes in the command are taken into account. This way for example the acceleration of a translation is defined by the axis with the lowest acceleration or speed.

Also, a combination of axes can be configured to use specified combined (vector) speed and acceleration. To do this the speeds and accelerations of the axes involved must be set using combined commands such as:

```
SPEEDXYZ=50 : ACCELXYZ=100
```

This causes all motion using the X, Y or Z axes (for example XY, XZ, YZ, XYZ) to use the given track speed and acceleration. An axis can be removed from a track speed group by setting its speed or acceleration separately (for example SPEEDY=SPEEDY). In this case only X and Z axes use the track speed setting given before.

Motion control commands such as GAIN=, INTG=, DERV= , SCOMP=, ACOMP=, DCOMP= and FILTERSIZE= set parameters for position control. Position controllers are used to keep the actual position (measured from a position encoder) reasonably close to the set value (position generated with motion commands). Accuracy, stability, softness, etc. of control can be tuned by setting control parameters according to need. Parameters can be set for each axis separately and they can also be set during motion.

Operation of position controllers is influenced by the properties of actuators and servo amplifiers as well as by the properties and gear ratios of transmissions, mechanisms and the location and resolution of speed and position sensors.

Therefore, the parameter settings of controllers may be quite different in different applications. McBasic has default values for control parameters and for other parameters of motion control. When started, McBasic uses these values until set in the program otherwise.

It is recommended that motion control parameters are set at the beginning of program even if some of the default values could be used in the application. This makes the program easier to read and modify.

11.1 ENCODER OPERATION

11.1.1 RES

Command	Set resolution for position scale of axes.
Syntax	RES[{*axes*...|(*n*,...,*m*)}]=*expression*
{*axes*...|(*n*...)}	List of axes (or (*n*...) - axes numbers), whose resolution is set. If not defined resolution is set to all axes.
expression	Value for resolution [pulse edges/measuring unit].

Function	Read resolution for position scale of axis.
Syntax	RES{*axis*|(*n*)}
{*axis*|(*n*)}	Identification letter or number of axis.
Type	Real number.
Value	Resolution of position scale of axis [pulse edges/measuring unit].

Set resolution for axes. Expression defines the number of pulse edges (counts) for a distance or angle unit (for example [edges/mm]).

1 encoder pulse cycle produces 4 pulse edges from two channels. This means that if a motor axis has an encoder that gives 500 pulses/revolution and one revolution of the motor moves the mechanism 5 mm, the resolution should be set to 4*500/5, or 400.

```
RES(1,2,3)=4*500/5
```

or

```
RES(1)=400
PRINT "Resolution is about";1/RES(1);"mm"
```

RES command affects the position scale and, among other things, interpretation of speed and acceleration.

Because RES affects many settings, it is advisable to set it at the beginning of the program, before setting any other parameters.

Setting resolutions of several axes to same value can be done using the combined command

```
RESXY=400
```

A combined RES command sets resolutions of different axes as they were set with separate commands. The resolutions of all axes 0 thru 9 can be set to the same value using a command such as:

```
RES=400
```

11.1.2 ENCSIZE

Command	Set encoder (counter) bitcount.
Syntax	ENCSIZE[{*axes*...|(*n*,...,*m*)}]=*expression*
{*axes*...|(*n*...)}	List of axes (or (*n*...) - axes numbers), whose encoder sizes are set.
expression	New encoder size, bits For incremental encoder -32...32 default 32 bits (max. counter range) For absolute encoder -24 ... 24, default -24 (centered 24 bit). Sign controls coordinate system: positive positive coordinate system negative centered coordinate system.

Function	Read encoder (counter) bitcount.
Syntax	ENCSIZE{*axis*|(*n*)}
{*axis*|(*n*)}	Identification letter or number of axis.
Type	Real number.
Value	Current encoder setting.

ENCSIZE setting allows control of counter operation for incremental encoders and data decoding for absolute encoders.

When using incremental encoders, setting ENCSIZE to a value less than 32 causes the position counters to wrap around after reaching the specified counter size. Thus for example setting ENCSIZEX=12 causes the position counter to return to 0 when reaching 4096 counts. Depending on RESX this may mean any POSX. This feature can be used in connection with binary line count encoders to wrap the position after for example 1 revolution to keep POSX between 0...1 in

mechanisms such as rotating knives etc. Wrapping can also be achieved after other (non-binary) count numbers using a rational FOLLOW ratio and a virtual axis connected to the actual axis.

When using absolute encoders, ENCSIZE can be used to limit the bit count used to equal or less than available from the encoder used.

With either encoder type, the sign of the ENCSIZE setting allows choosing a coordinate system from 0 to $2^{ENCSIZE}$ counts (positive system) or from $-2^{ENCSIZE-1}$ to $2^{ENCSIZE-1}$ counts (centered system). However, in ENCSIZE values 0 and 32 always set a 32 bit centered coordinate system as does ENCSIZE=-32.

11.1.3 OFFSET

Command	Set offset value or move current position
Syntax	OFFSET[{*axes*...|(*n*,...,*m*)}]=*expression*
{*axes*...|(*n*...)}	List of axes (or (*n*...) - axes numbers), whose offset are set. If not defined limits are set for axes 0...9 in the system.
expression	Offset value for specified axes, dimension as defined by RES..= command, for example [mm].

Function	Read offset value of axis.
Syntax	OFFSET{*axis*|(*n*)}
{*axis*|(*n*)}	Identification letter or number of axis.
Type	Real number.
Value	Current offset value for specified axes, dimension as defined by RES..= command, for example [mm].

The offset value of an axis represents the difference between the actual position counter (or value read from a position transducer) and the current position value (POS*axis*) of the axis. When McBasic starts, all OFFSET values are 0. Since the dimension of offset is dependent on the resolution, it is necessary to set RES before setting offsets in the program.

In case an incremental encoder is being used, the position counters are also reset to 0 when power is applied to the system, resulting in a zero current position value (POS*n*). Typically, the HOME.. command is then used in the program to find the correct zero position.

In case an absolute encoder is being used, the initial position at power up is determined by the encoder. The OFFSET..= command can then be used to set the coordinate system as necessary. Because the applicable offset value will remain the same unless the absolute encoder is replaced or moved relative to the mechanism, it is not necessary to find the zero position every time the system is started. However, it is possible to use the HOME.. command to determine the correct OFFSET value as a commissioning or service procedure.

When using commands that set the current position such as POS..= or HOME... the value of OFFSET is changed respectively. It is also possible to change the offset using a command like:

```
OFFSETX=OFFSETX+100
```

which adds 100 to the current position of X-axis (POSX). Setting the offset rather than POSX directly

```
POSX=POSX+100
```

has the advantage that in case X-axis is moving during the operation, no inaccuracy is introduced because of time difference between reading and writing the values.

Also, OFFSET can be used to "wrap" the position of an axis moving infinitely in one direction, such as a roller in a converting machine, in order to keep the position between 0 and 1, for example. A command like:

```
IF POSX>1 THEN OFFSETX=OFFSETX-1
```

will operate correctly (and wrap around) even when the value of OFFSET exceeds its 32bit range.

11.1.4 ENCERR

Function	Read encoder error counter.
Syntax	ENCERR{*axis*|(*n*)}
{*axis*|(*n*)}	Identification letter or number of axis.
Type	Integer 0...255
Value	Number of errors in encoder decoding since last read.

ENCERR provides a tool for monitoring errors and interference in encoder operation of WAX2 and WAX2A .

In conjunction with WAX2 (incremental encoder) ENCERR counts decoding errors where edges have been detected simultaneously in both channels (A and B). This situation indicates that accumulating errors probably result in encoder operation and therefore the system should be tested for ENCERR to be always zero during normal operation (first read may report errors occurred during power-up).

In conjunction with WAX2A (absolute encoder) ENCERR reports errors occurred in SSI transmission from encoder to WAX2A. 3 succeeding errors automatically result in a position loop error (MOVEREADY..=-64) and drive disable.

11.2 POSITION CONTROL SETTINGS

11.2.1 DRIVETYPE

Command	Configure operation of motion control.
Syntax	DRIVETYPE[{*axes*...|(*n*,...,*m*)}]=*type*
{*axes*...|(*n*...)}	List of axes (or (*n*...) - axes numbers), whose type is set.
type	Type setting: 0 DCD internal amplifier (AXE02704) 1 ±10V reference (AXE02703 or WAX[2][A] module) with WAX2[A] enable at ENA2 only 2 0...10V reference with direction output at WAX2 ENA1 3 as 1 but for WAX2[A] enable both at ENA1 and ENA2 Additional type data can be added to *type*. Each of the following addition removes the related function from axis: addition disabled function +16 limit switches, MAXERR and EMRG intervention +32 encoder counter +64 motion commands (MOVE,MOVER,MOVC,CREEP etc.) +128 position controller

Function	Read operation configuration of motion control.
Syntax	DRIVETYPE{*axis*|(*n*)}
{*axis*|(*n*)}	Identification letter or number of axis.
Type	Integer
Value	Type of axis as explained above.

The desired type is calculated by taking the sum of the basic type and additional type data.

For example axes to be controlled with normal ±10V reference output

```
10 DRIVETYPEXY=1
```

For example axis number 3 to be used as encoder input and analog output

```
10 DRIVETYPE(3)=1+16+64+128
```

McBasic has preset axis identifications, DRIVETYPE and LIMITTYPE default values (values which are valid before they are set with DRIVETYPE= and LIMITTYPE= commands).

These settings are system specific and their values depend on the system model and McBasic language version.

11.2.2 LIMITTYPE

Command	Configure axis limit switch operation.
Syntax	LIMITTYPE[{*axes*...|(*n*,...,*m*)}]=*type*
{*axes*...|(*n*...)}	List of axes (or (*n*...) - axes numbers), whose type of limit switches is set.
type	Type setting. Integer. *type* function 1 no limit switches 2 NLIM and PLIM, normal closed 3 LIM (n.c.) and MASK (open in negative end) 6 NLIM and PLIM, normal open 7 LIM (n.o.) and MASK (closed in negative end) 9 only index pulse (CLKX, edge-activated) 10 as 2, with index pulse 11 as 3, with index pulse 14 as 6, with index pulse 15 as 7, with index pulse where the signals are as follows: signal function connect to NLIM limit switch NLIM in negative end NLIM PLIM limit switch PLIM in positive end PLIM LIM common limit switch (activated in both ends) PLIM MASK limit switch mask, indicates the section of motion area where the servo is currently located NLIM CLKX index pulse, the exact 0-position CLKX

Function	Read axis limit switch configuration.
Syntax	LIMITTYPE{*axis*|(*n*)}
{*axis*|(*n*)}	Identification letter or number of axis.
Type	Integer
Value	Limit switch type of inspected axis as explained above.

McBasic can be configured with the LIMITTYPE command to use desired type of limit switch signals. The setting affects also operation of HOME command. When a motion being performed activates a limit switch while its reference is moving in the direction of the switch, servo power (enable) is cut off for the axis. However, it is possible to use motion commands to move away from the limit switch. In case a higher level of security is needed, the EMRG signal can be connected to outer limits and emergency switch arrangement to disable the axis completely thus requiring manual or mechanical override to get the axis back to operating area. In cases with high power or dangerous mechanisms both precautions are often used for maximum safety. It is also advisable to use power contactors to cut off servo system or motor power when EMRG is activated.

11.2.3 PIDFREQ

Command	Set refresh rate of position control loops.
Syntax	PIDFREQ=*expression*
expression	New refresh rate (50...2000). The feedback loop and position control algorithms will be executed *expression* times per second.

Function	Read refresh rate of position control loops.
Syntax	PIDFREQ
Type	Integer 50 ... 2000
Value	Current refresh rate [cycles/second]

PIDFREQ setting provides means for setting the refresh rate of the position control loops. By default PIDFREQ is 400 in standard McBasic versions. The setting is mutual for all axes in the system. Because the setting affects all time based motion parameters, it is recommended that PIDFREQ be set in the beginning of the program before any motion parameter settings.

11.2.4 GAIN

Command	Set proportional gain of position control.
Syntax	GAIN[{*axes*...|(*n*,...,*m*)}]=*expression*
{*axes*...|(*n*,..,*m*)}	List of *axes* or (*n*,...,*m*) - axes numbers), whose GAIN is set. If not defined, GAIN is set for axes XYZWABCDTU or 0..9 in the system.
expression	Value for gain. Integer 0..65535. 0 prevents operation of the feedback system. 65535 is the highest possible gain.

Function	Read proportional gain of position control.
Syntax	GAIN{*axis*|(*n*)}
{*axis*|(*n*)}	Identification letter or number of axis.
Type	Integer 0 ... 65535
Value	Gain value for position control of axis.

Proportional gain of the PID-control. GAIN defines the amount of output from the position controller in proportion to the position error of the axis.

In other words, GAIN represents the P of PID-control, that is the amplification factor. If GAIN is set to zero, operation of the controller is prevented and output (REF) is set to zero. However, even with GAIN set to zero, the feedforward part (SCOMP) of the output of the controller remains operable when using motion commands.

Too low GAIN causes an inaccurate control and too high GAIN causes system oscillation.

```
GAINXY=400
GAIN(2,12,14)=300
PRINT GAINX,GAINY,GAINZ
PRINT GAIN(12),GAIN(14)


400     400     300
300     300
```

In ARLACON MC300 and MC400 control systems, McBasic uses a 32-bit counter for position error in position control. Therefore the maximum position error is equal to the maximum operating area of the position measurement system, +/-2147483648 encoder pulse edges (counts). However, the proportional area, when the smallest available GAIN value (1) is used, equals to ±524288 encoder counts.

When using very low GAIN values, SCOMP parameter must usually be adjusted to correspond to drive full speed at 10V ref output to avoid excessive position lag. Using SCOMP values larger than full speed allow for some lag if necessary for tuning positioning settling time (see also FILTERSIZE).

11.2.5 INTG

Command	Set integrating factor of position control.
Syntax	INTG[{*axes*...|(*n*,...,*m*)}]=expression
{*axes*...|(*n*...)}	List of axes (or (*n*...) - axes numbers), whose INTG is set. If not defined, INTG is set for all axes.
expression	Value for INTG. Integer 0..255. 0 prevents integration. 255 is the quickest possible integrating factor.

Function	Read integrating factor of position control.
Syntax	INTG{*axis*|(*n*)}
{*axis*|(*n*)}	Identification letter or number of axis.
Type	Integer 0 ... 255
Value	Intg value of position control of axis.

INTG defines the part of reference output the position controller gives in relation to position error and time. The higher the INTG value and the position error are, the faster the reference output increases during time.

Constant position error, which can not be eliminated by proportional control, can be eliminated by using INTG. Because integrating increases the order the control system, it also increases the tendency to oscillation. Therefore, the use of INTG usually requires lower GAIN value for stability.

A too low INTG causes slow error correction and a too high INTG causes oscillation of the system.

In control systems already having one or more integrators, as usually when using a tacho generator feedback speed control circuit, INTG is usually set to zero.

```
INTG(1,2)=4
INTGY=3
INTGZ=0
PRINT INTG(1),INTGY,INTGZ

4       3       0
```

In control loops accurately speed compensated with SCOMP parameter and with fast acting speed loop, INTG can in some cases be used to reach better path accuracy without losing stability. This kind of a control is usually possible in accurate and stiff mechanisms.

INTG value 0 removes the integrating operation. INTG represents the I in PID-control, that is, the integrating factor.

11.2.6 DERV

Command	Set derivation factor of position control.
Syntax	DERV[{*axes*...|(*n*,...,*m*)}]=expression
{*axes*...|(*n*...)}	List of axes (or (*n*...) - axes numbers), whose DERV is set. If not defined, DERV is set for all axes.
expression	Value for DERV. Integer 0..255. 0 prevents operation of derivation. 255 is the highest possible derivation factor.

Function	Read derivation factor of position control.
Syntax	DERV{*axis*|(*n*)}
{*axis*|(*n*)}	Identification letter or number of axis.
Type	Integer 0 ... 255
Value	Derv value of position control of axis.

DERV defines the part of reference caused by quick change in position error for every axis. The higher DERV is set the stronger the reaction to change is. By compensating for delays in actuators and mechanisms and reducing the control output when the error is diminishing, DERV helps to stabilize the operation of control circuits.

Usually a high inertia mass and/or slow reacting drive require higher DERV value. On the other hand a too high DERV value may cause unstability or sluggish settling.

DERV value 0 removes derivative operation. DERV represents the D of PID-control, that is the derivation factor.

```
GAIN=10 : INTG=0          'affects all axes
DERVXY=7 : DERV(3)=6
PRINT GAINX,INTGX,DERVX
PRINT GAIN(3), INTG(3), DERV(3)

10      0       7
10      0       6
```

11.2.7 SCOMP

Command	Set speed compensation of position control.
Syntax	SCOMP[{*axes*...|(*n*,...,*m*)}]=*expression*
{*axes*...|(*n*...)}	List of axes (or (*n*...) - axes numbers), whose SCOMP is set. If not defined, SCOMP is set for axes 0....9 (XYZWABCDTU) in the system.
expression	Value for speed compensation. 0 prevents operation of compensation.

Function	Read speed compensation of position control.
Syntax	SCOMP{*axis*|(*n*)}
{*axis*|(*n*)}	Identification letter or number of axis.
Type	Real number.
Value	SCOMP value of position control of axis.

Speed feedforward or speed compensation for position control. Value of expression is greater or equal than speed of axis when control output is set to maximum (=full speed).

SCOMP parameter can be used to add to the control (reference) output a part depending on the theoretical instantaneous speed of the position set value.

The factor allows for accurate path control of motion even without integration in the position loop. SCOMP parameter is often used, when the position controller is used in connection with a tacho generator feedback speed control circuit. In this case the best possible control result is often reached by setting INTG=0 and using SCOMP parameter as needed.

SCOMP setting is called critical, when the controlled axis follows the generated motion without noticeable position error at all speeds.

Setting SCOMP according to the speed of the axis when the reference output reaches its maximum value (for example 10V) results in critical SCOMP.

For example if X axis runs 400 mm/s when the servo amplifier is controlled with a 10V reference voltage and RESX is set to [pulse edges/mm], the critical compensation is set with

SCOMPX=400

Setting SCOMP to a greater value results in undercritical compensation. Undercritical compensation can be used for example to prevent overshoot at the end of a motion. A too low

SCOMP setting results in negative position lag, the axis is ahead of the position set value, which is usually not applicable.

SCOMP value 0 removes the operation of speed compensation.

When calculating actual speed compensation out of the SCOMP setting, McBasic uses the current RES values.

11.2.8 ACOMP

Command	Set acceleration compensation of position control.
Syntax	ACOMP[{*axes*...|(*n*,...,*m*)}]=*expression*
{*axes*...|(*n*...)}	List of axes (or (*n*...) - axes numbers), whose ACOMP is set. If not defined, ACOMP is set for axes 0...9 (XYZWABCDTU) in the system.
expression	Value for acceleration compensation. 0 prevents operation of compensation.

Function	Read acceleration compensation of position control.
Syntax	ACOMP{*axis*|(*n*)}
{*axis*|(*n*)}	Identification letter or number of axis.
Type	Real number.
Value	ACOMP value of position control of axis.

Acceleration feedforward or acceleration compensation for position control. Value of expression is greater or equal than acceleration of the axis when 100% (10V) is added to the control output.

ACOMP parameter can be used to add to the control (reference) output a part depending on the theoretical instantaneous acceleration of the position set value.

The factor allows for accurate path control of motion during acceleration even without integration in the speed loop. ACOMP parameter is often used, when the position controller is used in connection with a proportional speed control circuit without integration and with limited gain.

ACOMP setting is called critical, when the controlled axis follows the generated motion with similar position error during acceleration and constant speed. In connection with a critical SCOMP value this error is near zero.

For example if X axis requires an additional 2V (20% of full 10V scale) of reference in order to accelerate at 400 mm/s2 and RESX is set to [pulse edges/mm], the critical compensation is set with

ACOMPX=400/0.2
or
ACOMPX=2000

Thus 2000mm/s2 is the theoretical acceleration produced with a 10V (100%) reference at zero speed (providing such current would be applicable to produce the acceleration).

Setting ACOMP to a greater value results in undercritical compensation. Undercritical compensation can be used to compensate only partly for the lag caused by acceleration for optimum dynamic performance. A too low ACOMP setting results in negative position lag during acceleration, the axis is ahead of the position set value, which is usually not applicable.

ACOMP value 0 removes the operation of acceleration compensation.

When calculating actual acceleration compensation out of the ACOMP setting, McBasic uses the current RES values.

11.2.9 DCOMP

Command	Set deceleration compensation of position control.
Syntax	DCOMP[{*axes*...|(*n*,...,*m*)}]=*expression*
{*axes*...|(*n*...)}	List of axes (or (*n*...) - axes numbers), whose DCOMP is set. If not defined, DCOMP is set for axes 0...9 (XYZWABCDTU) in the system.
expression	Value for deceleration compensation. 0 prevents operation of compensation.

Function	Read acceleration compensation of position control.
Syntax	DCOMP{*axis*|(*n*)}
{*axis*|(*n*)}	Identification letter or number of axis.
Type	Real number.
Value	DCOMP value of position control of axis.

Deceleration feedforward or deceleration compensation for position control. Value of expression is greater or equal than deceleration of the axis when 100% (10V) is deducted from the control output.

The value of DCOMP is set to equal to ACOMP when ACOMP is set for the axis. DCOMP setting can then be altered after the ACOMP has been set. The dimension of DCOMP is similar to that of ACOMP, so DCOMP values differing from ACOMP can be used to adjust for differences caused by friction and efficiency behaviour during acceleration and deceleration thus allowing optimisation of the dynamic behaviour of the system during different phases of motion. Typically DCOMP must be set to a somewhat greater value than ACOMP to allow less compensation during deceleration.

11.2.10 FILTERSIZE

Command	Set filter type and length for position set value to limit jerk or noise.
Syntax	FILTERSIZE[{*axes*...|(*n*,...,*m*)}]=*expression*
{*axes*...|(*n*...)}	List of axes (or (*n*...) - axes numbers), whose FILTERSIZE is set. If not defined, FILTERSIZE is set for axes 0...9 (XYZWABCDTU) in the system.
expression	Value defining type and length of position set value filter. Positive filter values represent averaging filters with a length of (*expression*)*(position loop cycle time). Negative values represent filters with zero position lag at constant speed with a response length of (*-expression)**(position loop cycle time) 0 prevents operation of compensation. Maximum value is dependent on the filter lengths available in different McBasic versions (256 in MC400 BAS16, 128 in MC400 BAS100 and 64 in MC300 BAS300)

Function	Read position set value filter type and length.
Syntax	FILTERSIZE{*axis*|(*n*)}
{*axis*|(*n*)}	Identification letter or number of axis.
Type	Integer.
Value	Current FILTERSIZE setting for *axis*.

Position set value filtering is typically used for limiting the rate of change of acceleration (jerk) in various types of motion. This is often referred to as using S-ramps in motion profiles. With FILTERSIZE it is possible to define the time it takes for the acceleration to change during any type of motion. The period of time used to reach the new acceleration after a change in the acceleration is defined as a number of position loop cycles. Thus, for example a FILTERSIZE value of 10 corresponds to 25ms period for a system with PIDFREQ=400 (1s / 400 = 2.5ms). Limiting jerk effectively allows limiting the rate of change of torque and current in the drive system. As in practical drive systems rate of change of current is limited by the bandwidth of the current loop and the maximum available voltage and the inductance of the motor, using suitable FILTERSIZE allows generation of position profiles that the drive is able to follow. Also, it is often even more important to limit the frequency content of motion to be performed by a given machanism. By using suitable FILTERSIZE values unwanted oscillation and strain in mechanisms can be avoided.

Using positive FILTERSIZE values causes an axis to have a position lag compared to the original position reference (before fitering). At constant speed this lag is equal to half of the filter period multiplied by current speed. For example a 25ms filter at 1000mm/s would cause a 12.5mm lag.

Using a filter also causes the total time for a translation to be one filter period longer than the original translation. Usually this is well compensated by the shorter actual settling time when using a suitable filter.

Negative FILTERSIZE values use a different digital FIR (finite response) filter to allow filtering of measured position such as a reference encoder giving position or speed information for other axes.

A negative FILTERSIZE value causes overshoot when changes in acceleration and speed occur, but has no position lag during motion at constant speed. While removing unwanted noise from measured signals FILTERSIZE can also generate a higher resolution reference position from a low resolution position encoder by adding a 16 bit fractional part to it and thus dividing the actual increments in 65535 parts for extra resolution in generating new filtered position values every position control cycle.

The following picture shows the effect of FILTERSIZE for a typical translation (MOVE).

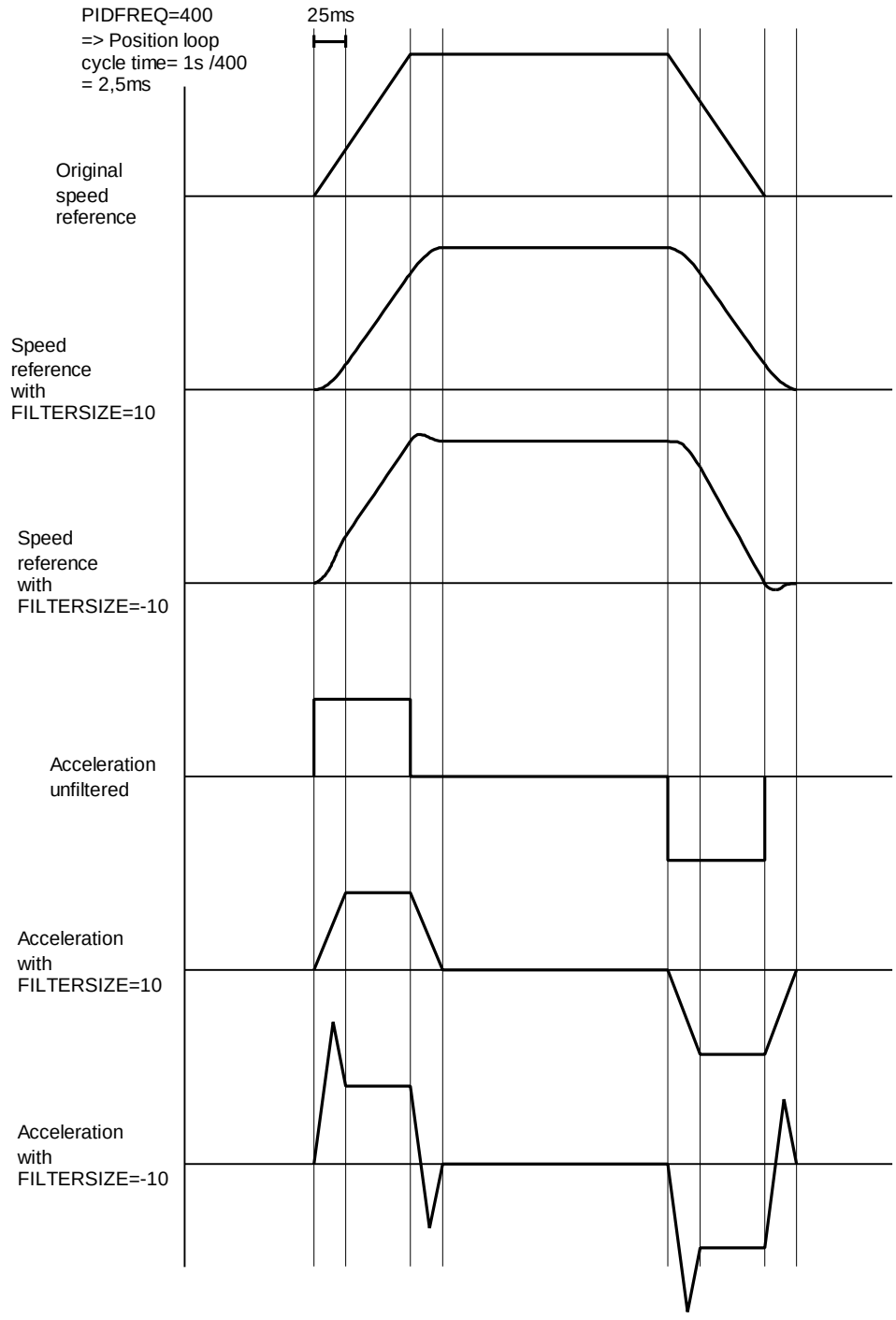


Fig. 11.2.10, The effect of FILTERSIZE

11.2.11 SPEED

Command	Set motion speed.
Syntax	SPEED[{*axes*...|(*n*,...,*m*)}]=*expression*
{*axes*...|(*n*...)}	List of axes (or (*n*...) - axes numbers), whose vector speed is set. If there is only one axis, its speed is set. If not defined, SPEED is set for all axes.
expression	Setting for speed.

Function	Read motion speed.
Syntax	SPEED{*axis*|(*n*)}
{*axis*|(*n*)}	Identification letter or number of axis.
Type	Real number.
Value	Speed setting for MOVE and MOVC commands for specified axis.

Set the motion speed to value expression for the axes axes.. ([mm/s], if RES is set to [pulse edges/mm]). Setting influences all future motion commands. SPEED command sets the speed the translations started with MOVE and MOVC commands use between acceleration and deceleration phases.

To reach the speed set with SPEED command the length of the translation must be long enough and value of ACCEL high enough to allow for a constant speed phase between acceleration and deceleration.

Speed can be set for a single axis, for example:

```
SPEEDX=750
SPEED(5)=523
```

The specified axis follows this setting when moved alone. If several axes, with speeds set with different SPEED commands, are moved by common MOVE or MOVC command, the translation is executed using linear interpolation and limiting the motion speeds so, that none of the axes exceed their set speeds or accelerations.

Speed can also be set for a combination of axes, for example:

```
SPEEDXYZ=750
SPEED(2,5,6)=230
```

This setting affects the axes involved as if the speeds were set separately when any of the axes is moved alone. If the axes are moved by common motion commands, the translation follows the set track (vector) speed. The vector speed is calculated by taking the square root of the sum of squares of every speed in the group, for example

Vxyz = (Vx2+Vy2+Vz2)0.5

When setting all axes available in the system the axis names may be left away. For example in a two axis system the command SPEED=100 is equivalent to SPEEDXY=100.

The maximum speed value for calculation of a motion is ±32767 counts/control cycle and resolution is 2.3E-10 counts/control cycle. This means that for example in a normal MC300 or MC400 system with a control cycle of 2,5ms with a resolution such as 100 edges/mm the maximum speed is about 131 m/s and minimum speed is about 0,00000007 mm/s. Same limitations are valid also for CREEP command.

11.2.12 ACCEL

Command	Set acceleration of motion.
Syntax	ACCEL[{*axes*...|(*n*,...,*m*)}]=*expression*
{*axes*...|(*n*...)}	List of axes (or (*n*...) - axes numbers), whose vector acceleration is set. If only one axis, its independent acceleration is set. If not defined acceleration is set to all axes in the system.
expression	Value for acceleration.

Function	Read acceleration of motion.
Syntax	ACCEL{*axis*|(*n*)}
{*axis*|(*n*)}	Identification letter or number of axis.
Type	Real number.
Value	Acceleration used in MOVE, MOVC and CREEP commands with axis.

Sets motion acceleration and deceleration to value expression to axes.. ([mm/ss], if RES has been set to [pulse edges/mm]).

Setting ACCEL affects motion commands executed after the setting.

```
110 SPEED=50 : ACCEL=250
115 SPEED(2)=60 : SPEEDX=45

310 PRINT SPEED(2),ACCEL(2),ACCELX,SPEEDX

60    250    250     45
```

ACCEL command affects the axes as SPEED command and therefore it must be set to same combinations of axes as SPEED to achieve the desired vector speed and acceleration.

ACCEL setting is limited by the maximum change of speed being ±32767 counts/(control cycle)2 with a resolution of 1/16777216 counts/(control cycle)¨. This means that for example in a normal MC300 or MC400 system with control cycle of 2,5ms and resolution such as 100 edges/mm the maximum acceleration is about 52000m/s2 and minimum acceleration is about 3E-7mm/s2.

11.2.13 MAXERR

Command	Set limit for position controller position error intervention.
Syntax	MAXERR[{*axes*...|(*n*,...,*m*)}]=*expression*
{*axes*...|(*n*...)}	List of axes (or (*n*...) - axes numbers), whose limits are set. If not defined limits are set for all axes in the system.
expression	Value for limit, for example [mm].

Function	Read limit for position controller position error intervention.
Syntax	MAXERR{axis|(n)}
{axis|(n)}	Identification letter or number of axis.
Type	Real number.
Value	Current maximum error limit.

Set limit for position controller position error intervention according to the value expression for axes axes.. ([mm], if RES is set as [pulse edges/mm]).

If motor (=encoder) position differs from the position set value more than MAXERR, motor control is automatically disabled. Setting MAXERR=0 prevents the intervention of MAXERR.

To ensure quick and reliable protection function MAXERR should be set to a value somewhat higher than the practical position error during motion.

11.3 POSITION CONTROL FUNCTIONS

11.3.1 POS

Command	Set position counter.
Syntax	POS[{*axes*...|(*n*,...,*m*)}]=*expression*
{*axes*...|(*n*...)}	List of axes (or (*n*...) - axes numbers), whose positions are set. If not defined position is set for axes 0...9 (X,Y,Z,W,A,B,C,D,T and U).
expression	New position in units defined by RES..=, for example [mm].

Function	Read position counter.
Syntax	POS{*axis*|(*n*)}
{*axis*|(*n*)}	Identification letter or number of axis.
Type	Real number.
Value	Current actual position of axis in units as set by RES..= command.

With POS command the position value of an axis can be set to desired value. Position of axis is set to expression independent of position of axis at time of setting. In other words, POS command moves the coordinates as specified.

```
POSZ=10 : POSX(2)=0
```

When desired, coordinates can be moved relative to the current position of axis by considering the actual position, for example

```
POSX=POSX+100
```

moves the coordinates 100mm in negative direction; in other words the position value is increased by 100mm.

The POS function can be used to read the current actual position. POS is also convenient for reading encoder inputs not configured for position control.

```
IF POS(1)>200 THEN STOPMOVE(1)
```

The size of McBasic 3.2 position counters is 32 bits. This means, that for example with a resolution of 100 [edges/mm] position can have values between ±214748364800 mm or ±214km. If position exceeds either the maximum or minimum value of counter, it "wraps" over to the other end of the range. This allows moving over the limits of position counters when for example using relative motion commands or with FOLLOW or CREEP etc..

11.3.2 FPOS

Function	Read filtered position set value.
Syntax	FPOS{*axis*|(*n*)}
{*axis*|(*n*)}	Identification letter or number of axis.
Type	Real number.
Value	Current position set value of axis in units as set by RES..= command.

FPOS allows reading the position set value as seen by the position control algorithm. The effect of FILTERSIZE..= ,if filtering is being used, is also seen in FPOS

11.3.3 RPOS

Function	Read unfiltered position set value.
Syntax	RPOS{*axis*|(*n*)}
{*axis*|(*n*)}	Identification letter or number of axis.
Type	Real number.
Value	Current position set value of axis before filtering in units as set by RES..= command.

RPOS allows reading the position set value unfiltered. As the use of filtering causes a lag in the response of FPOS, RPOS can be used for observing the operation of the filter. It may also be preferred in algorithms programmed in the application requiring the unfiltered position as an input.

11.3.4 FSPEED

Function	Filtered current speed.
Syntax	FSPEED{*axis*|(*n*)}
Type	Real number.
{*axis*|(*n*)}	Identification letter or number of axis.
Values	The instantaneous speed of the position set value of *axis* after filtering if FILTERSIZE*axis* set >0.

The true theoretical speed of an individual axis axis can be inspected with this function. If filtering with the FILTERSIZE..= command is being used to limit the acceleration rise times (S-ramp), the effect is also seen in FSPEED. See also RSPEED.

```
SPEEDX=500 : SPEED(1)=250
ACCELX=500 : ACCEL(1)=250
MOVEX 3000 : MOVE(1:1500)
DELAY .1
PRINT FSPEEDX, FSPEED(1)
DELAY 1
PRINT FSPEEDX, FSPEED(1)

50.00    25.00
500.00  250.00
```

11.3.5 RSPEED

Function	Unfitered current speed.
Syntax	RSPEED{*axis*|(*n*)}
Type	Real number.
{*axis*|(*n*)}	Identification letter or number of axis.
Values	The instantaneous speed of the position set value of *axis* before filtering if FILTERSIZE*axis* set >0.

In case filtering with FILTERSIZE is being used for the axis, RSPEED can be used to observe the position reference speed before the filter. This may be necessary for monitoring purposes or for certain algorithms where extra delay caused by filtering may affect the operation of feedback loops performed by the application program.

11.3.6 POSERR

Function	Read value of position error.
Syntax	POSERR{*axis*|(*n*)}
{*axis*|(*n*)}	Identification letter or number of axis.
Type	Real number.
Value	Current position error, for example [mm].

The difference between the set value of position and the actual value of position can be read with the POSERR function. When motion is stopped, POSERR is equal to the positioning error. During motion POSERR represents the deviation from desired track along the axis. For example

```
320 IF ABS(POSERRX)>2 THEN STOPMOVE
```

stops motion if error of X-axis is more than 2 mm.

11.4 HOME

Command	Automatic synchronization (zero point search) of coordinate system.
Syntax	HOME{*axes*...|(*n*...)}
{*axes*...|(*n*...)}	List of axes (or (*n*...) - axes numbers), participating in search.

Motion axes axes.. begin the zero point search sequence. MOVEREADYaxes..=0 until the zero points are found and the motion has stopped. Speed when searching is one eight (1/8) of the speed set with SPEED command.

The operation of HOME command is affected by the selected limit switch configuration (LIMITTYPE) as follows:

No limit switches or index (LIMITTYPE 1)

Position of axes axes is set to zero. Equal to POSaxes=0.

Limit switches (LIMITTYPE=2 or 6)

When only limit switches (NLIM and PLIM) are used the zero point search operates so, that the HOME command causes motion into negative direction until the negative limit switch. When the limit switch is influenced, the motion changes its direction and continues until the limit switch is no longer influenced. The zero point is set to this position. The axis continues to move to the positive direction for the deceleration distance.

Limit switches and index (LIMITTYPE=10 or 14)

If the index channel is also used, the axis continues after leaving the limit switch until a pulse is received from index channel (CLKX). The origin of coordinates is set according to index pulse and motion stops at the distance of deceleration in positive direction.

Only index (LIMITTYPE=9)

When using only index channel for search of origin the motion moves in the positive direction until a pulse is received from the index channel. The origin of coordinates is set according to index pulse and motion stops at the distance of deceleration in positive direction.

Limit switch and mask (LIMITTYPE=3 or 7)

Operation using index mask. In this case limit switch signals are connected so, that the limit switches in both ends of motion influence the PLIM -input. A signal, which changes its state somewhere in the motion area close to the position where origin is searched, is connected to NLIM-output.

HOME function will then move the axis to the position where mask signal changes its state and set origin to a location where mask signal changes its state when running to positive direction. The axis stops after deceleration distance from this point. Motion speed while searching the edge of the mask is as set with SPEED command, unlike in other motion performed by HOME command.

Limit switch, mask and index (LIMITTYPE=11 or 15)

If also the index channel used, the axis continues to move after it has passed the edge of mask until a pulse is received from index channel (CLKX). The origin of coordinates is set at the index pulse and the axis stops after deceleration to positive direction. Speed when searching the index is one eight (1/8) of the speed set with SPEED command.

```
100 ' RUNNING HOME
110 HOMEXYZ
120 IF NOT MOVEREADYXYZ THEN 190
130 RETURN
```

11.5 STOPMOVE

Command	Stop motion.
Syntax	STOPMOVE[{*axes*...|(*n*...)}]
{*axes*...|(*n*...)}	List of axes (or (n...) - axes numbers) to stop. If not defined, axes 0..9 are stopped.

STOPMOVE stops motion generated by MOVE, MOVER, MOVC, MOVCR, CREEP or MOVEPROF commands using the currently defined deceleration. If higher deceleration is required,

DECEL can be set before STOPMOVE command. Note that STOPMOVE does not cancel any FOLLOW ratios.

Desired axes can be selected to be stopped. Stopping is performed with servo control active. Servo control also remains active, unless MAXERR limit is not exceeded during stopping.

```
130 ACCELXZ=900 : STOPMOVEXZ

600 STOPMOVE ' axes 0 thru 9
```

STOPMOVE also provides a method to change the destination of a translation or to change the type of motion performed. For example:

```
MOVEX 10000
DELAY 3
STOPMOVEX : MOVCX 1500
```

would cancel the first translation after 3 seconds and change the destination to 1500 without stopping.

```
MOVEPROFXY
DELAY 3
STOPMOVEX : MOVCX 1500
```

would cancel the profile motion after 3 seconds and start a translation to position 1500 obeying set ACCELX or DECELX to reach the set SPEEDX.

11.6 MOVEREADY

Function	Read motion status.	
Syntax	MOVEREADY[{*axes*...|(*n*...)}]	
Type	Integer	
{*axes*...|(*n*...)}	List of axes (or (*n*...) - axes numbers). If not defined all axes are considered.	
Values	1	motion ready, axis enabled
	0	motion not ready (busy)
	-1	servo control disabled by setting PWRn=0
	-2	negative limit NLIM exceeded
	-4	positive limit PLIM exceeded
	-8	emergency switch EMRG open
	-16	MAXERRn exceeded
	-32	excessive errors in McWay i/o loop (WAYERR)
	-64	encoder errors in an absolute encoder (WAX2A)

Read motion status. With the MOVEREADY function it is possible to read whether motion with defined axis or axes is not ready, ready or stopped (servo control disabled) for some other reason.

If more than one of the above mentioned conditions exist simultaneously, MOVEREADY gets a value where different error values are added, for example MOVEREADYn=-10 if negative limit switch is influenced and emergency stop is open.

MOVEREADY is -1 also when control is not yet enabled. Control is enabled for example with the command

```
PWRaxis=1
```

or by performing any translation command. For example MOVERX(0) (relative translation of zero length) starts the controller, but no motion is performed.

```
330 IF MOVEREADY<0 THEN STOP
340 IF MOVEREADYZ THEN PRINT "Z ready"
```

11.7 TRANSLATIONS

11.7.1 MOVE

Command	Absolute translation.
Syntax	MOVE{*axes*..(*expr*,..,*expr*)|(*n*:*expr*,..., *n*:expr)}
{*axes*..|*n*...}	axes to perform the translation.
expr	Destinations of translation, expressions in same order as the axis identification letters are defined in the system.

Translation is performed using the given axis combination axes.. along a straight line (linear interpolation) using accelerations and speeds set with ACCEL and SPEED commands.

Parameters expression must always be given in the order, in which the axes, that are used for the translation, are defined in the control system. List axes.. defines the axes which are used to perform the translation. For simplicity, it is recommended to follow the definition order of system. The axis identification letters (if such type of axis identification is used) are usually in order X,Y,Z,W,A,B,C,D,T,U.

```
140 MOVEXZ(X0,Z1+1)
144 MOVEY(20.050)
146 MOVEX100
MOVE(4:X0) : MOVE(1:Xcod0,2:Ycod0,3:Zcod0+Xcod0)
```

11.7.2 MOVER

Command	Relative translation.
Syntax	MOVER{*axes*..(*expr*,..,*expr*)|(*n*:*expr*,..., *n*:expr)}
{*axes*..|*n*...}	List of axes used to perform the translation.
expr	Lengths of translations, expressions in same order as the axis.

Similar to MOVE, with the difference that axes are moved a distance defined by expressions from their current position.

```
140 MOVERXZ(X0,Z1+1)
144 MOVERY(20.050)
146 MOVERX100
MOVER(4:X0) : MOVER(1:Xcod0,2:Ycod0,3:Zcod0+Xcod0)
```

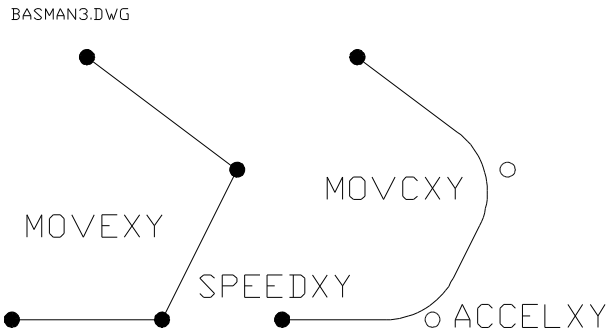


Fig. 11.7.2, two axis translations

11.7.3 MOVC AND MOVCR

Command	Absolute continuous translation.
Syntax	MOVC *{axes..(expr,..,expr)|(n1:expr,..., nn:expr)}*
{*axes*..|*n*...}	List of axes to perform the translation.
expr	Destinations of translation, expressions in same order as the axis.

Command	Relative continuous translation.
Syntax	MOVCR{*axes*..(*expr*,..,*expr*)|(*n*:*expr*,..., *n*:expr)}
{*axes*..|*n*...}	List of axes to perform the translation.
expr	Lengths of translation, expressions in same order as the axis.

MOVC.. commands operate as MOVE.. commands, with the difference that the translation is started before the previous translation has stopped.

Deceleration and acceleration phases of translations are combined so, that the result is continuous motion.

When performing MOVC translations with several axes, the path does pass accurately through every corner point. Instead, the path is "shaved" to allow for continuous motion. Amount of rounding depends on speed and acceleration settings. Low speed with high acceleration produces sharp corners and high speed with low acceleration produces smooth corners.

When performing motion commands McBasic calculates phases of translation and saves them into the motion buffer (MOVEBUFFER). This is called initializing a translation. If no motion is being executed by axes concerned, the translation is performed immediately.

If a translation is currently being executed, the new translation remains waiting in the buffer. For continuous motion using MOVC commands, at least one initialized translation defined by a MOVC command must be waiting in the buffer when the deceleration phase of the previous translation begins. The maximum number of translations in the motion buffer is 4 for each axis combination.

If a MOVC.. motion command is executed during the previous translation deceleration phase, the acceleration phase for the new translation begins immediately after motion command has been executed. This can be used for example to limit speed to a desired level in motion path corners.

```
150 FOR A=0 TO 2*PII STEP 0.1
160 MOVCXY(R*SIN(A),R*COS(A))
170 NEXT A
```

or the same in an other form

```
REAL Angle, Radius
:
FOR Angle=0 TO 2*PII STEP 0.1
MOVC(1:Radius*SIN(Angle),2: Radius*SIN(Angle))
NEXT Angle
```

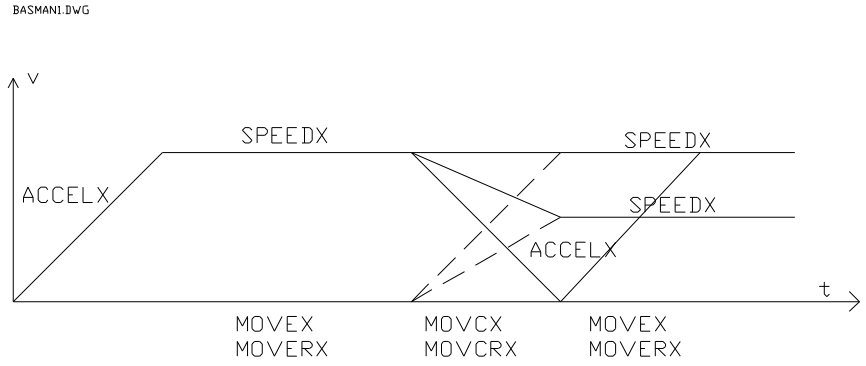


Fig. 11.7.3, Speed profiles

11.7.4 MOVEBUFFER

Function	Read motion buffer status.
Syntax	MOVEBUFFER[{*axes*...|(*n*...)}]
Type	Integer 0 .. 4
{*axes*...|(*n*...)}	List of axis (or (*n*...) - axes numbers) combination to inspect. If not defined, the buffer for axes 0..9 in the system is inspected.
Values	0 motion ready n one translation not ready, n-1 waiting to start

Read motion buffer memory status. The number of initialized translations for a given axis combination axes..(see MOVC -commands) can be read with this function.

if MOVEBUFFER is

0	motion is ready
1	1 translation not ready, none waiting
2	1 translation not ready, 1 in buffer
3	1 translation not ready, 2 in buffer
4	1 translation not ready, 3 in buffer (buffer full)

Because there is not space for more than 4 translations in the motion buffer, giving a motion command for an axis combination axes.. while MOVEBUFFERaxes.. is 4 causes the program to stop at the motion command until free space is available in motion buffer (an unfinished translation becomes ready). If this happens in a program with several tasks, the task waiting for space in MOVEBUFFER passes control to the next task waiting to be put in execution.

```
 100 PWRXY=1
1000 R=100 '              CIRCLE, RADIUS R
1010 X0=50 : Y0=50 '      MIDDLE POINT X0,Y0
1020 FOR A=0 TO 2*PII STEP PII/8
1030 IF MOVEREADYXY<0 THEN STOP
1040 IF MOVEBUFFERXY>3 THEN 1030
1040 MOVCXY(X0+R*COS(A),Y0+R*SIN(A))
1050 NEXT A
```

or the same in an other form

```
REAL Radius,Ang ‘Circle radius, current angle
REAL Xorig,Yorig ‘coordinates of the center
PWR(1,2)=1
Radius=100 : Xorig=50 : Yorig=50
FOR Ang=0 TO 2*PII STEP PII/8
IF MOVEREADY(1,2)<0 THEN STOP
IF
  MOVEREADY(1,2)>3
THEN
  DO : UNTIL MOVEREADY(1,2)<4 : LOOP
ENDIF
MOVC(1:Xorig+Radius*COS(Ang),2:Yorig+Radius*SIN(Ang))
NEXT Ang
```

11.8 CREEP

Command	Start axis motion at given speed.
Syntax	CREEP{*axes*..(*expr*,..,*expr*)|(*n*:*expr*,...,*n*:*expr*)}
{*axes*..|*n*...}	List of axes, whose speed is set.
expr	Speeds of motion, expressions in the same order as axes are.

With the CREEP command it is possible to produce servo axis motion according to speed setting without destination. For example with command

```
CREEPX(10)
CREEP(2:100,3:X0*3.0)
```

X axis is set to run at 10mm/s (if RES is [edges/mm]). Acceleration and deceleration are performed according to current ACCEL and DECEL values. Changes in parameters influence immediately, also during acceleration and deceleration.

```
CREEPXYZ(10,20,15)
  5 ACCELX=100    'see Fig. 11.8
 10 CREEPX100
 20 DELAY 3
 30 ACCELX=150
 40 CREEPX70
 50 DELAY 1
 60 ACCELX=50
 70 CREEPX30
 80 DELAY .5
 90 ACCELX=100
100 DELAY .75
110 ACCELX=75
120 CREEPX100
130 DELAY.2
140 CREEPX0
```

BASMAN2.DWG

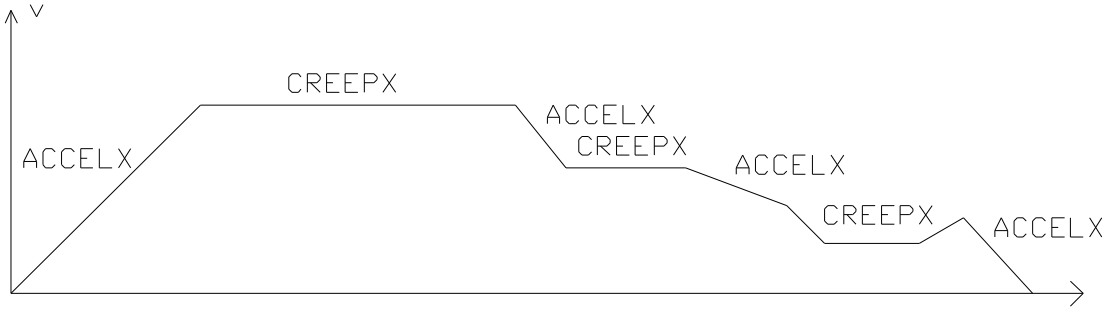


Fig. 11.8, motion with CREEP -command

11.9 FOLLOW [AT]

Command	Set an axis to follow another axis.
Syntax	real ratio: FOLLOW*axis1axis2*(*i*) FOLLOW(*axnr1*,*axnr2*,*i*) [AT (*axnr3*,*pos*)] or rational ratio: FOLLOW*axis1axis2*(*n*,*m*) FOLLOW(*axnr1*,*axnr2*,*n*,*m*) [AT (*axnr3*,*pos*)]
axis1, *axnr1*	Axis, which follows.
axis2, *axnr2*	Axis, which is followed.
i	Gear ratio between the axes. Setting gear ratio to 0 disables the follow function between the axes and also resets any condition set with FOLLOW AT.
n,m	Gear ratio between axes as rational number. n is the number of teeth in the primary gearwheel and m the number of teeth in the secondary gearwheel. *n* and *m* can be integers between 1 to 8000000.
axnr3	Defines the axis that triggers the follow ratio to be activated when using FOLLOW AT. If [AT (*axnr3*,*pos*)] is omitted, follow ratio is activated immediately. *Axnr3* can be same as *axnr1* or *axnr2* or any other axis in the system.
pos	Defines the position at which the follow ratio is activated when using FOLLOW AT.

Set *axis1* to follow position of *axis2* with a gear ratio of *i* between axes. *Axis1* has to be enabled. FOLLOW can be also operate for example while other types of motion (translations, CREEP, profile) is performed by these axes.

```
FOLLOWXY(.1) ' X follows Y with a ratio 1:10
```

The FOLLOW AT command can be used to start the follow function when a specified axis reaches a specified position. This can be used to accurately syncronize axes also when using profile motion.

```
FOLLOW(0,1,17,23) AT (1,500)  ' start axis 0 to follow 1 with
                              ' ratio 17:23 when 1 reaches 500
```

Because the ratio used in FOLLOW command is defined internally as a fixed point binary number with an 8 bit integer and 16 bit decimal part, the following limitations are valid its operation:

Maximum value of ratio i is ±128 counts/count with accuracy of 1/65536 counts.

To follow continuous reference motion without inaccuracy caused rounding error, at least the reference encoder must be selected to have a power of 2 pulse number. (for example 512 or 1024 pulses/round).

11.10 FOLLOWRATIO

Function	Read current follow ratio for axis.
Syntax	FOLLOWRATIO(*axis*)
Type	Real number
axis	Identification number of axis.
Value	The ratio with which *axis* is currently following some other axis. 0 if no current ratio exists.

FOLLOWRATIO function can be used for example to test if a FOLLOW AT condition has been reached.

11.11 PWR

Command	Start and stop position control. Set maximum value of control output.
Syntax	PWR[{*axes*...|(*n*...)}]=expression
[{*axes*...|(*n*...)}]	List of axes. If not defined, axes 0...9
expression	Value to set. (0 .. 1) 0 Control off. 0<a<1 Start with maximum output of a*physical maximum (a*10V) 1 Start normal operation

Function	Read control output limit.
Syntax	PWR{*axis*|(*n*)}
Type	Real number (0 .. 1)
{*axis*|(*n*)}	Identification letter or number of axis.
Values	Maximum value of control output (ref), as explained above for expression.

```
PWRXYZ=1 ' enable XYZ
PWR=0 'all axes off

PWRX=0 'only X axis off
```

The output of a position controller can be limited by using values of expression between 0 < expression < 1 for example to dampen the torque glitch when starting and stopping the controller or to prevent damages when testing or during critical parts of work cycle, for example, in case of an encoder fault.

1 represents the full output (for example +/-10V). 0.5 represents the half of maximum value (for example +/-5V).

When using a drive in torque control mode, the maximum torque can be limited with PWR setting.

```
200 PWR=0
210 FOR N=0 TO 1 STEP .1
220 PWRX=N
230 TIMER(0)=.1
240 IF TIMER(0) THEN 240
250 NEXT N
```

11.12 OPWR

Command	Forced set position control reference output.
Syntax	OPWR[{*axes*...|(*n*...)}]=*expression*
{*axes*...|(*n*...)}	List of axes (or (*n*...) - axes numbers). If not defined, outputs of all axes are set.
expression	Value to set. (-1 .. 1). Output is set to expression*physical maximum.

Function	Read the position controller output.
Syntax	OPWR{*axis*|(*n*)}
Type	Real number (-1 .. 1)
{*axis*|(*n*)}	Identification letter or number of axis.
Values	State of reference output of axis. As expression above.

For example in a ±10V control output, the following values correspond to each other:

OPWR	VREF
-1	-10V
-0.5	-5V
0	0V
0.5	5V
1	10V

Using OPWR= command disables the normal operation of position controller.

If limit switches are in use, they are also operable when using OPWR command. EMRG and MAXERR also operate normally.

The state of the reference output can be read with the OPWR function also when the controller is operating. This can be used to study load effects etc.

For example to set speed compensation of X axis automatically:

```
MOVERX1000 : DELAY 2 : SCOMPX=RSPEEDX/OPWRX
```

11.13 FAST POSITION CAPTURE

11.13.1 CAPTTYPE

Command	Activate position capture operation.
Syntax	CAPTTYPE[{*axes*...|(*n*,...,*m*)}]=*expression*
{*axes*...|(*n*...)}	List of axes (or (*n*...) - axes numbers), whose capture operation is activated.
expression	Controls mode of operation (input and edge) Capture position when: 0 encoder index channel falling edge 1 encoder index channel rising edge 2 inp0 falling edge 3 inp0 rising edge

The CAPTTYPE= command can be used with the WAX2 servo connection module with incremental encoder to arm the fast position capture logic included in the module hardware. CAPTTYPE= command allows use of the encoder index channel (X-channel) or WAX2 module first input (inp0) falling or rising edge as a trigger for the capture event. Note that the inp0 referred to is the first input on the WAX2 module with the encoder for the axis in question. It also has an input address as defined by WAYMOD$..= command for use with the INP() function.

Function	Read position capture status.
Syntax	CAPTTYPE{*axis*|(*n*)}
{*axis*|(*n*)}	Identification letter or number of axis.
Type	Real number.
Value	Current status of position capture of axis. -2 not used -1 ready (position captured) 0 waiting for index falling edge 1 waiting for index rising edge 2 waiting for inp0 falling edge 3 waiting for inp0 rising edge

Note that since the operation of the capture function involves communication between the processor and the axis module, CAPTTYPE= command should only be executed once to arm the logic. About 2 position loop cycles should be allowed for the logic to be active. When CAPTTYPE function reports that a position has been captured (CAPTTYPE=-1), it takes some time (about 2 position loop cycles) for CAPTTYPE function to return the corresponding value. Thus it is advisable to set CAPTTYPE only once and the ensure that the value of the CAPTTYPE function indicates that the logic has been armed. Trying to arm the logic several times with a CAPTTYPE= command without waiting for CAPTTYPE to reach the set value first may cause position loop malfunction. Example:

```
DO
  CAPTTYPEX=0                        'look for index rising edge
  DELAY .05                          'wait for logic to be armed
  DO UNTIL CAPTTYPEX=-1 : LOOP       'wait for edge to be found
  IF CAPTTYPEX=-1 THEN X=CAPTPOSX 'read captured position to X
  LOOP
```

11.13.2 CAPTPOS

Function	Read captured position.
Syntax	CAPTPOS{*axis*|(*n*)}
{*axis*|(*n*)}	Identification letter or number of axis.
Type	Real number.
Value	Position captured from *axis*. 0, when no position has been captured yet.

CAPTPOS allows reading the position captured by WAX2 module fast position capture logic. It should only be read after CAPTTYPE.. returns -1 (ready). The position thus acquired represents the actual position at the trigger event.The accuracy with WAX2 using the index channel is approximately ±1 encoder pulse edge, and . ±0.1 millisecond using inp0.

11.14 JOYSTICK OPERATION

11.14.1 JOY

Function	Read joystick position.	
Syntax	JOY(*n1*) or JOY{X|Y}	
Type	Real number (-1 .. 1)	
n1	Joystick channel (0...15) 0 (X) 1 (Y) *n1*	 horizontal direction vertical direction numbered joystick channels
Values	0 -1 1	Joystick in center position. Joystick in left/down position. Joystick in right/up position.

If a proportional control device, such as a joystick, is connected to the control system through a control terminal, the position of the device can be read both in horizontal (for example JOY(0) or JOYX) and in vertical (for example JOY(1) or JOYY) direction with this function.

The value of function varies between -1 ... +1. 0 represents the center position.

JOY function can be used for manual axis control. For example:

```
 V=100
DO
 CREEPX JOY(0)*V
 CREEPY JOY(1)*V
 LOOP
```

allows both X and Y axes manual speed control with a joystick with a maximum speed of 100.

11.14.2 JOYINP

Command	Assign joystick channels and control joystick data transfer.
Syntax	JOYINP#*n1*[,*n2*,*n3*]
n1	Device number, where the joystick data is read from. If *n1* =0 joystick channels *n2* and *n3* are disabled.
n2	Joystick channel number for x-data (0...15, default 0)
n3	Joystick channel number for y-data (0...15, default 1)

Maximum 8 devices can be connected to terminals with joysticks simultaneously

Defines the channel number (which may be in the range of 0...15) from which to read joystick data. If for example a hand control terminal equipped with a joystick is connected to port #2, the command

```
JOYINP#2
```

can be used to start serial port #2 joystick communication. When joystick communication is active, the joystick position can be read with JOY(*n1*) function.

Command

```
JOYINP#0,n2,n3
```

ends the joystick communication. JOYINP#0 is also valid.

For example, display the position of a joystick in a device connected to #2:

```
10 OPEN#2,":LP"
20 JOYINP#2=1
30 PRINT JOY(0),JOY(1),CHR$(13);
40 GOTO 30
```

JOYINP#*n1*,*n2*,*n3* is used to connect more then one joystick to system at the same time. It also enables easy indexing of joystick inputs.

```
OPEN#2,"S1:"
OPEN#3,"S2:"
JOYINP#2,0,1   'x-data to 0, y-data to 1
JOYINP#3,2,3   'x-data to 2, y-data to 3
DO
  FOR I=0 TO 3
  CREEP(I:S0*JOY(I)) 'CONTROLS FOUR SPEEDS
  NEXT I
LOOP
```

11.15 PROFILE CONTROLLED MOTION

With MOVEPROF commands it is possible to connect one or more axes to operate relative to another axis according to a pre-programmed position profile.

In this case the synchronizing axis controls an array pointer in the profile array of the axis to be synchronized. The position of the synchronized axis is defined by the value in the profile array. The values between array values are calculated using linear interpolation.

11.15.1 PROFSIZE

Command	Set profile array size.
Syntax	PROFSIZE{*axes*|(*axnr*,...}=*expression*
axis | *axnr*	Axes, whose profile array size is being redefined.
expression	New size for profile array. Must be a power of 2, max. 2^{18} within limits of available memory.

Function	Read profile array size.
Syntax	PROF{*axis*|(*axnr*)}
Type	Real number
axis | *axnr*	Axis, whose profile array size is being read.
Value	Current profile array size of *axis*.

PROFSIZE provides a method for setting the size of the motion profile array of any axis individually. By default the setting is 2048 when McBasic is started. McBasic reserves the memory for the profile only after using the profile (writing to it), so profile settings for unused axes do not reserve memory.

After using a profile it is not possible to redefine the profile size for the axis until McBasic is restarted from McDOS or NEW command is used.

11.15.2 PROF

Command	Write to a profile array.
Syntax	PROF{*axis*(*n*)|(*axnr*,*n*)}=*expression*
axis, axnr	Axis, whose profile array is being written to.
n	Number of the profile array entry.
expression	Value to write into cell (position).

Function	Read from a profile array.
Syntax	PROF{*axis*(*n*)|(*axnr,n*)}
Type	Real number
axis, axnr	Axis, whose profile array is being read from.
n	Number of the profile array entry.
Value	Value of cell (position).

The number of array entries for each axis is can be set with the PROFSIZE*n*= command.

For example, when using a 512 size profile, cells 0-511 form the actual motion profile. Array entry 512 is set in non-progressive motion equal to entry 0. In progressive motion the progression of the profile is PROFaxis(512)-PROFaxis(0) for each cycle.

For example to set a sine formed profile for X axis and a cosine formed profile for Y axis (circulating motion in a plane)

```
10 FOR N=0 TO 511
20 PROFX(N)=SIN(2*PII*N/512)
30 PROFY(N)=COS(2*PII*N/512)
40 NEXT N
50 PROFX(512)=PROFX(0)
60 PROFY(512)=PROFY(0)  'Note! PROF as function
```

11.15.3 MOVEPROF

Command	Start profile motion.
Syntax	MOVEPROF*axes*(*axis2*) MOVEPROF(*axnr1*:*axnrp1*,...,*axnrn*:*axnrpn*)
axes	Axes, which are started to move according to their PROF arrays.
axis2	Axis controlling the array pointer.
axnr1 .. axnrn	Axes, which are started to move according to their PROF arrays.
axnrp1 ... axnrpn	Axes controlling the array pointers.

Moving by a profile array is performed by moving axis2 or axnrpn which may be same or different axes. Usually the axes used as pointer axes are so called virtual axes, existing only theoretically (DRIVETYPE=176). A virtual axis does not represent any real, physical axis.

A virtual axis is actually an axis, whose DRIVETYPE is set so, that only motion commands are operable. Usually an axis that has no physical control connection is used as a virtual axis.

For example

DRIVETYPET=16+32+128

When starting profile motion the necessary axes must be active (in other words PWRaxis1=1 and PWRT or PWRaxis2=1). It is recommended to set the resolution of the synchronizing axis to same as the number of entries in profile array, for example REST=512. With this resolution a command MOVERT(1) moves the axis (axes) one profile array cycle.

For example to start a circulating motion (see the profile tables generated in the example in paragraph (12.28.1)) with axes X and Y:

```
100 REST=512 : ACCELT=2
120 PWRXYT=1
130 MOVEPROFXY(T)
140 CREEPT(2)
```

11.16 POSITION CONTROL LOG

11.16.1 LOGSIZE

Command	Set motion control data log size.
Syntax	LOGSIZE*axes*=*n* or LOGSIZE(*axnr,...*)=*n*
axes	Axis letter or a list of axis letters, whose log size is set.
axnr,...	Axis number or a list of axis numbers, whose log size is set.
n	Size of log array (samples) to be reserved for specified axes. Max. 65535 within available memory.

Function	Read data log size.
Syntax	LOGSIZE{*axis*|(*axnr*)}
Type	Real number
axis | *axnr*	Axis, whose log size is being read.
Value	Current log array size (samples) of *axis*.

The LOGSIZE..= command is used before LOGDATA or LOG commands to set the size of the log array used for storing logged data. The default value for the log array size is 400 samples.

McBasic reserves the memory for the log only after using it (LOG...=), so log size settings for axes not logged do not reserve memory.

After using a log it is not possible to redefine the log size for the axis until McBasic is restarted from McDOS or NEW command is used.

11.16.2 LOG

Command	Motion control data log control.
Syntax	LOG*axes*=*k* or LOG(*axnr,...*)=*k*
axes	Axis letter or a list of axis letters, whose log is controlled.
axnr,...	Axis number or a list of axis numbers, whose log is controlled.
k	Logging interval expressed in control cycles (1/PIDFREQ). After each interval the data is written into the log. If *k* =0, the log is stopped.

Start/stop motion control data log on specified axes. Value *k* specifies the log interval as a multiple of position control cycles. If *k* is 1 the control data is saved for every control cycle (for example after every 2,5 ms if PIDFREQ is set to 400). With higher values of *k*, data is saved after each *k* control cycles. This way data can be logged for a longer period while not using more memory.

The array is always filled so, that the first entry in the array is the latest data. The older data is automatically shifted in the log array. This way, history of data from the desired time period can easily be maintained in the log array. If *k* is 0 data logging is stopped.

```
example:
LOGSIZEXY=1000  ' set log array sizes for X and Y
LOGXY=5         ' data logging every 5 cycles
SPEEDXY=100
ACCELXY=1000
MOVERXY(150,150)
DO UNTIL MOVEREADYXY : LOOP 'during motion
DELAY .1        'small delay
LOGXY=0         'stop logging
```

11.16.3 LOGDATA

Function	Read motion control log data.
Syntax	LOGDATA*axis*(*sample*,*data*) or LOGDATA(*axnr*,*sample*,*data*)
Type	Real number.
axis	Identification letter axis.
axnr	Number of axis when number reference is used
sample	The number of sample read, integer (0..LOGSIZE-1). Sample 0 is the latest sample, LOGSIZE-1 is the oldest sample.
data	The number of data to read, integer (0..7, see below)

Read the data stored in log array gathered using the LOGaxes..=n command. With different values of parameter data the following data can be read. Type of all data is real number. The variable t_{sample} means the time when sample was stored calculated backwards from latest sample.

data	content	dimension
0	time t0-tsample	s (gets value 0, if sample greater than size of log)
1	actual position	mm
2	set position	mm
3	position error	mm
4	control output	-1 ... 1 as OPWR
5	actual speed	mm/s
6	set speed	mm/s
7	analog channel	-1 ... 1 as INPA()

In the above RES is assumed to be set as [pulse edges/mm].

To set an analog channel of a WIA analog input module to be logged synchronously with the other data, use the LOGSIZE..= command before starting logging with the LOG..= command.

Logging an analog input can be used to monitor values such as motor torque or current that may be available as analog signals from a drive or some other transducer.

Command	Set analog input for LOGDATA data 7
Syntax	LOGDATAaxis=a or LOGDATA(axnr)=a
axis	Identification letter axis.
axnr	Number of axis when number reference is used
a	Analog input number. Number of analog input INPA(a) to be logged.

Each log entry uses 10 bytes of memory without and 14 bytes with analog data logging.

12. I/O CONNECTIONS

The digital and analog inputs and outputs in Arlacon control systems are available for programming with dedicated McBasic commands and functions.

Inputs and outputs exist in the control system both in various i/o -modules and for example in motion control connection modules. These modules can reside in an MC400 rack or connected to a McWay distributed i/o loop.

12.1 I/O CONFIGURATION

Before a McWay loop can be used, it must be initialised using the McBasic WAYMOD$ command.

12.1.1 WAYMOD$

Command	I/o system configuration.
Syntax	WAYMOD$(*n,m*)=*string*
n	Loop number.(0...7)
m	module number (0...120)
string	specification and information of module: "END" end of modules in loop "EMPTY" no modules in current position "WIN INP(*nn*)" WIN with 24 inputs from nn (0..4x65535) "WOU OUT(*nn*)" WOU with 32 outputs from *nn* (0..4x65535) "WIO IO(*nn*)" WIO with 16 in/ 16 out from *nn* ((0..4x4095) "WIO INP(*n1*) OUT (*n2*)" WIO with 16 in from *n1* and 16 out from *n2* (0..4x4095) "WOA OUTA(*nn*)" WOA with 6 outputs from *nn* (0..65535) "WIA INPA(*nn*)" WIA with 6 inputs from *nn* (0..65535) "WIA6 INPA(*nn*)" WIA with 6 inputs from *nn* (0..65535) "WIA4 INPA(*nn*)" WIA with 4 inputs from *nn* (0..65535) "WIA2 INPA(*nn*)" WIA with 2 inputs from *nn* (0..65535) "WAX INP(*n1*) [OUT(*n2*)] IO(*n3*)" WAX 02006 (32bits) "WAX2 INP(*n1*) [OUT(*n2*)] IO(*n3*)" WAX2 (32bits) "WAX2A INP(*n1*) [OUT(*n2*)] IO(*n3*)" WAX2A for absolute encoder (40bits) 4 limit inputs starting from *n1* (0..4x255) 4 control outputs starting from *n2* (0..4x255) 8 digital i/o starting from *n3* (0..4x65535) *n2* = *n1* if OUT(*n1*) omitted "WAX POS(*n1*) [OUTA(*n2*)] IO(*n3*)" WAX for position input(32bits) *n1* position input (0..255) *n2* analog output (0..255) 12 digital i/o starting from *n3* (0..4x65535) *n2* = *n1* if OUT(*n1*) omitted

For further details of module specific syntaxes refer to chapter 3 of 'ARLACON McWay I/O - system user's manual'.

12.1.2 WAYERR

Function	Read and reset McWay i/o loop error counter.
Syntax	WAYERR(*loopnr*)
Type	Integer 0...255
Value	Number of failed refresh cycles after last read. Reading resets the counter.
loopnr	McWay loop number (0...7).

The WAYERR function gives access to an error counter in the control system that counts defective transmissions in the McWay i/o loop. Each loop in the system has its own counter that can be accessed using the number of the specific loop. In MC300 based systems loopnr is always 0. MC400 systems can be configured to have up to 8 loops (0...7).

Each error counter advances when the controller sends a loop refresh message but does not receive a correct response from the loop. Thus, WAYERR essentially counts failed refresh cycles. The maximum error count can be 255. When reading the WAYERR function for a loop, the respective counter is reset to zero. Therefore, if WAYERR is used in a program to monitor the correct operation of the installation, it should only be read after suitable intervals, such as some minutes, or the value should be accumulated in a separate variable. When starting a system, several error may accumulate in the counters because of power-up sequencing. Therefore the first read-reset of WAYERR should be ignored.

As 3 consecutive failed cycles cause axis position control to automatically switch off, error should not occur regularly in any loop. In a correctly operating system no more than 1 error occurs within a minute and no more than 10 errors occur within a day. While considerably higher error rates can occur without affecting the operation of an application, it is a good practice to observe that the error levels are within normal and even include error level check in the program.

```
Example of a simple WAYERR check routine:

CheckWay
  IF TIMER(5)=0
   IF WAYERR(0)>3 THEN PRINT "Wayerrors"
   TIMER(5)=300
   ENDIF
  RETURN
```

12.1.3 MOTION CONTROL I/O LOGICAL ADDRESSES

When using WAX2 connection modules, the motion control related inputs and outputs on the modules are numbered according to the axis number. Each axis occupies four i/o addresses in and out as follows.

axisnr	*address*	INP(*address*)	OUT(*address*)
n	*n**4	ENCX encoder index	n/a
	*n**4+1	NLIM negative limit switch	ENA1 relay output
	*n**4+2	PLIM positive limit switch	ENA2 relay output
	*n**4+3	EMRG emergency stop	n/a

Signal names represent the physical signal located in WAX2 modules. Output addresses marked n/a are not in use. In a limit switch configuration with index mask, nlim is the mask and plim is the limit switch data.

The axes in the system are numbered starting from 0. The first I/O address for each axis is its number multiplied by 4. This address is also used in conjunction with axis module settings (WAYMOD$) to specify the axis for an axis connection module. Axes can be numbered freely within the available axis count in the system (usually 16 axes). The first 10 axes numbered 0 thru 9 have also letter names X,Y,Z,W,A,B,C,D,T,U in the same order.

12.1.4 I/O LOGICAL ADDRESSES

When PIA02301 i/o modules are used in MC400 systems, the addresses of inputs and outputs are as follows.

I/o - units PIA02301 (8 inputs and 8 outputs)

1.	unit	INP(32-39)	OUT(32-39)	EDGE(32-39)
2.	unit	INP(40-47)	OUT(40-47)	EDGE(40-47)
3.	unit	INP(48-55)	OUT(48-55)	EDGE(48-55)
4.	unit	INP(56-63)	OUT(56-63)	EDGE(56-63)
5.	unit	INP(64-71)	OUT(64-71)	EDGE(64-71)
6.	unit	INP(72-79)	OUT(72-79)	EDGE(72-79)
7.	unit	INP(80-87)	OUT(80-87)	EDGE(80-87)
8.	unit	INP(88-95)	OUT(88-95)	EDGE(88-95)
9.	unit	INP(96-103)	OUT(96-103)	EDGE(96-103)
10.	unit	INP(104-111)	OUT(104-111)	EDGE(104-111)

Edge -commands available only in model PIA02301A.

Other i/o connected via McWay loops are configurable as described in McWay documentation.

12.2 DIGITAL I/O

12.2.1 INP

Function	Read status of input.	
Syntax	INP(*expression*)	
Type	Truth value.	
expression	Input address	
Values	0	input not active (off)
	1	input active (on)
	-1	Communications error (McWay
	-2	Missing module (McWay)

INP function is used for reading the status of a binary input.

```
250 IF INP(3)=0 AND INP(4) THEN 250
```

12.2.2 OUT

Command	Control an output.
Syntax	OUT(*expr1*)=*expr2*
expr1	Address of output.
expr2	Value to set 0 output off 1 output on

Command sets a binary output on or off.

For example

```
110 OUT(35)=1
120 OUT(36)=ON
```

sets on outputs number 35 and 36.

Function	Read output status.
Syntax	OUT(*expression*)
Type	Truth value.
expression	Address of output.
Values	0 output off 1 output on -1 Communications error (McWay -2 Missing module (McWay)

OUT function is used for reading the status of an output. If the output has not been set previously, its status is 0.

```
IF OUT(5)=0 THEN
  OUT(5)=1 : DELAY 0.5
  ENDIF
OUT(5)=0
```

12.2.3 EDGE

Command	Control edge indicator of input.
Syntax	EDGE(*expr1*)=*expr2*
expr1	Address of input.
expr2	Control data 0 acknowledge and inhibit 1 acknowledge and reset

Edge indicator operates by setting an edge flip-flop, when input INP(*expr1*) receives an edge (a change in state). Direction of change that causes the transmission (rising/falling edge) depends on settings of input unit.

```
EDGE(1)=0
```

Operation of the edge flip-flop can be prevented by setting EDGE(*expr1*)=0. By setting EDGE(*expr1*)=1 the flip-flop is reset to be set by the next edge received by the input. The edge flip-flop status can be read with EDGE function.

Function	Read status of edge indicator.
Syntax	EDGE(*expression*)
Type	Truth value.
expression	Address of input.
Values	0 no edge detected 1 edge detected

With EDGE function it is possible to inspect if an edge has been detected in input INP(expression) after it was last reset (see EDGE=).

```
DO
 IF EDGE(7) THEN EDGE(7)=1 : N=N+1
 UNTIL N=>N0
 LOOP
```

12.3 ANALOG I/O

Analog i/o is available for MC400 systems as modules in the rack and for all Arlacon MC systems as McWay modules

MC400 analog input modules operate by converting the incoming analog signal to a frequency signal in the control system. Similarly the outputs operate by converting the frequency signal generated by the control system to an analog signal. Conversions are performed in conversion modules isolated from control system logic and therefore possible interference is prevented from disturbing the operation of the control system.

Analog i/o on McWay modules and other analog i/o is accessed using the INPA and OUTA commands.

12.3.1 INPA

Function	Read analog input.
Syntax	INPA(*expression*)
Type	Real number.
expression	Address of analog input.
Values	Status of input -1 highest negative input voltage(current) 0 zero 1 highest positive input voltage(current)

The function can be used for reading the voltage or current that is connected to an analog input.

```
PRINT INPA(0)
```

```
0.54            (5.4V in input with ±10V scale)
```

Expression defines the address of the analog input to read. Function returns the value 0, if there is 0V/mA at the input.

For McWay analog i/o the values are read from the A/D converters on the i/o modules.

In MC400 rack analog inputs a VFC03603 V/F converter is used to convert the input voltage or current to frequency. A frequencies 0/5000/10000Hz correspornd to -1/0/1 readings respectively

12.3.2 OUTA

Command	Set analog output.	
Syntax	OUTA(*expr1*)=*expr2*	
expr1	Address of analog output.	
expr2	Value to set (-1 .. 1)	
	-1	highest negative value
	0	zero
	1	highest value

Function	Read analog output.	
Syntax	OUTA(*expression*)	
Type	Real number.	
expression	Address of analog output.	
Values	Status of out	
	-1	highest negative input voltage(current)
	0	zero
	1	highest positive input voltage(current)

Analog outputs can be set using this command. When starting the control system, all analog outputs are set to 0.

Expr2 is the value to be set to output and can vary between -1 .. 1. Value zero represents the smallest output value (usually 0V or 0mA). Value 1 represents the highest positive value (for example 10V or 20mA, depends on the scale of output). Value -1 represents the most negative output value when using ± type output.

```
OUTA(2)=0.7
```

With McWay analog i/o outputs can also be read with the OUTA() function.

With the MC400 FVC03604 unit the analog output OUTA can not be read as a function.

12.3.3 IFREQ

Function	Read a frequency input.	
Syntax	IFREQ(*expression*)	
Type	Real number.	
expression	Number of frequency input.	
Values	Measured frequency [Hz]	
	0	underflow
	min	20
	max	200000

The frequency in an MC400 rack analog or frequency input can be read with the IFREQ function. McBasic does not distinguish the analog an frequency inputs in program processing, so frequency of an analog input converter module (5-10kHz or 0-10kHz) can also be read with IFREQ function.

```
PRINT IFREQ(0)
```

5420.00

Frequencies read with IFREQ function can vary between 20Hz and 200 kHz. When using high frequencies the TMR unit must be equipped with fast optocouplers. Measuring period is 1 millisecond or one period with frequencies under 1000Hz.

13. ERRORS

When an error condition occurs, McBasic normally stops program execution, closes open files and prints an error message and the address where the error was found. Program execution can be continued from the error line by 'CONT' command. Usually it is not desirable to stop program execution for example because of a mistake the user makes on keyboard. For this kind of cases an error handling routine can be defined to sort the error situation and continue program execution. However, if an error is encountered in the error handling routine, the program stops and a normal error message is generated.

Error messages used in McBasic:

```
1            parameter overflow
2            'INPUT' error
3            strange character or variable
4            closing parenthesis missing
5            'DIM' error
6            strange expression
7            linenumber error
8            variable overflow
9            too many subroutines
10           strange 'RETURN'
11           strange variable
12           strange command
13           parenthesis error
14           too big program
15           index error
16           too many 'FOR'/'NEXT'-loops
17           odd 'NEXT'
18           'FOR'/'NEXT'-loop structure error
19           unfinished 'FOR'/'NEXT'-loop
20           'ON' error
21           Error #21
22           'DEF' structure error
23           function error
24           string error
25           string overflow
26           I/O error
27           strange address
28           address error
29           internal string error
30           '=' error
31           'IF' structure error
32           end of DATA
33           renumber error
34           cannot CONT
35           internal stack error
36           stack overflow
37           internal structure error
38           ','-error
39           odd 'RESUME'
40           too many TASKs
41           structure stack overflow
42           structure nesting error
43           'DO/LOOP' structure error
44           strange label
```

```
45          same label twice

53          file error
54          strange date
55          too many links
56          you can not use links here
57          loop in links

60          strange module
61          address error
62          I/O-loop full
63          address should be multiple of four
```

13.1 ERROR

Command	Print an error message on console.
Syntax	ERROR *expression*
expression	Number of error message. (1 .. 127)

Error message. This command is used to generate an error message. Program execution stops or jumps to error handling program (see ON ERROR). Number of error message is the value of expression.

```
IF A>100 THEN ERROR 1
```

13.2 ON ERROR

Command	Jump in case of an error. Set error trap.
Syntax	ON ERROR *address*
address	Address, where to jump in case of an error.

Defines the address of the error handling routine. If this command has been executed, an error anywhere in the program causes a jump to line *address*. The error trap is task specific, so it can be set differently for each task if necessary. By default, every new task inherits its error trap setting from its parent task.

```
   ON ERROR ErrHandling

ErrHandling
  STOPMOVE
  FOR N=32 TO 47
  OUT(N)=0
  NEXT N
  PRINT "CALL FOR SERVICE"
  PRINT "ERROR ";ERR,ERR$(ERR)
  PRINT "ON LINE",ERL
  STOP
```

13.3 RESUME

Command	Return from the error handling routine of ON ERROR command.
Syntax	RESUME [NEXT]
[NEXT]	If NEXT part is not used the return address is the beginning of the line where the error occurred. If NEXT is used, return address is the beginning of the next line.

```
RESUME
IF ERR=2 THEN RESUME NEXT
```

13.4 ERR

Function	Number of the last occurred error.
Syntax	ERR
Type	Integer (0 .. 127)

13.5 ERL

Function	Line number of the line, where an error last occurred.
Syntax	ERL
Type	Integer 0 ... 65535.
Values	Line number of the line where the error occurred. 0 for line without linenumber 1..65535 program line

This function is not effective if line numbers are not used in the program. In this case it is always equal to 0. For programs without line numbers, use the ERR@ function instead to obtain the address of the line where the error last occurred.

```
10010 PRINT "Error #";ERR;
10020 PRINT " on line ";ERL
```

13.6 ERL$

Function	Contents of the line, where an error last occurred.
Syntax	ERL$
Type	String 80 characters.
Values	Contents of the line as text string.

```
PRINT "Error #";ERR;
PRINT " on line ";ERL$
```

13.7 ERR$

Function	Error message as string.
Syntax	ERR$(*expression*)
Type	String
expression	Number of error message.
Values	Error message as defined in error message table.

This function can be used for example to print the error message corresponding to an error number.

```
PRINT ERR$(ERR)
FOR I=1 TO 255
PRINT ERR,ERR$(I) : NEXT I
```

13.8 ERR@

Function	Error line address.
Syntax	ERR@
Type	Address
Values	Address of the line where an error last occurred.

```
For example:

PRINT ERR@

(Label+3)
```

13.9 ONERR@

Function	Error trap current address.
Syntax	ONERR@
Type	Address
Value	Current error trap address for current task. If error trap not set, value is (+0).

ONERR@ function can be used to check the status of the error trap.