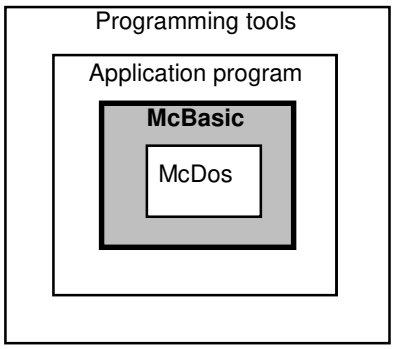
McBasic 3.3 programming language reference manual for SKS Control ACN MPU3 based motion control systems



SKS Control Oy,
Martinkyläntie 50 01720 VANTAA
tel +358-20 76461 fax +358-207646740, email: control@sks.fi
30.3.2017, Ari Lindvall

Table of contents:

1. GENERAL

This manual describes the use of McBasic programming language version 3.3 supplied with SKS Control ACN motion control systems with MPU3 motion controller unit.

Since the programming environment is used under the McDos operating system we recommend also 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 monospace font is used in them.

```
StartOfProgram
  ' example
```

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

GOTO *address* 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 enters are written in normal text

```
PRINT "Text"
Text
```

The controller output is 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 older 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 2016

SKS Control Oy

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
ABCONF$(*a*)=*expr*	set Anybus module configuration	FIELDBUSES
ACCEL*axes..=expr*	set acceleration	MOTION CONTROL
ACCEL(*axnr,..,axnr*)=*expr*	set acceleration	MOTION CONTROL
ACOMP*axes..=expr*	set acceleration feedforward	MOTION CONTROL
ACOMP(*axnr,..,axnr*)=*expr*	set acceleration feedforward	MOTION CONTROL
ADDR *var,....,var*	declare address variables	VARIABLES AND TABLES
APPEND *filename*	append to program	FILES
BIT *arrayname(dim,..,dim), ..*	declare bit array	VARIABLES AND TABLES
BLOCK$(*expr*)=*string*	write block	STRINGS
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
CIRCLEMOVCR(*expr,expr:expr,…*)	circular motion	MOTION CONTROL
CIRCLEMOVER(*expr,expr:expr,…*)	circular motion	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$(*#devicenr*)=*datestring*	set date	FILES,CLOCK
DCOMP*axes..=expr*	set deceleration feedforward	MOTION CONTROL
DCOMP(*axnr,..,axnr*)=*expr*	set deceleration feedforward	MOTION CONTROL
DECEL*axes..=expr*	set deceleration	MOTION CONTROL
DECEL(*axnr,..,axnr*)=*expr*	set deceleration	MOTION CONTROL
DEF FN*fnname....*	define user function	OTHER COMMANDS
DELAY *expr*	delay	TIMING
DELETE *addr,addr*	remove program lines	CONTROL
DERV*axes..=expr*	set position control derivation	MOTION CONTROL
DERV(*axnr,..,axnr*)=*expr*	set position control derivation	MOTION CONTROL
DIGITS=*expr*	set number of decimal places	FILES
DIM *variablename*(*dim,..,dim*), ..	declare variables or arrays	VARIABLES AND TABLES
DO..UNTIL..LOOP	repeat loop	STRUCTURE
DOS	exit to operating system	CONTROL
DRIVETYPE*axes..=type*	configure servo connections	MOTION CONTROL
DRIVETYPE(*axnr,..,axnr*)=*expr*	configure servo connections	MOTION CONTROL
ECAX(*ax,aout,inp,ena,nrun,prun*)=	set EtherCat axis I/O configuration	FIELDBUSES
ECCO(*node*,*index*,*subindex*,*bytes*)=	write Ethercat CoE register	FIELDBUSES
ECMOD$(*n*,*m*)=	set EtherCat fieldbus configuration	FIELDBUSES
ECPAR(*n*,*par*)=	set EtherCat node parameters	FIELDBUSES
ECSERNUM(*node*)=	set EtherCat node s/n specification	FIELDBUSES

ED *address*	edit program	CONTROL
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
ETHERCAT(*port,function*)	control Ethercat master	FIELDBUSES
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,*]..	read value for 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
JCOMP*axes..=expr*	set jerk compensation	MOTION CONTROL
JCOMP(*axnr,..,axnr*)=*expr*	set jerk compensation	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
LONG INTEGER *var*(*dim,..,dim*), ..	declare long integer variables or arrays	VARIABLES AND TABLES
LONG *var*(*dim,..,dim*), ..	declare long integer variables or arrays	VARIABLES AND TABLES
LONG(*#devicenr*)=expression	write long integer	FILES

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
MBCLOSE [(*expr)*]	close ModBus channel[s]	FILES AND COMMUNICATIONS
MBDATA(*expr,expr)=expr*	write ModBus registers	FILES AND COMMUNICATIONS
MBREG(*expr,expr*)=*expr*	write ModBus server settings	FILES AND COMMUNICATIONS
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
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
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
OVERRIDE*axes= expression*	set axes speed/accel scale	MOTION CONTROL
OVERRIDE(*axnr,..,axnr*)= *expression*	set axes speed/accel scale	MOTION CONTROL
OVERRIDERATE*axes= expression*	set axes override change rate	MOTION CONTROL
OVERRIDERATE(*axnr,..,axnr*)= *expr*	set axes override change rate	MOTION CONTROL
PIDFREQ=expression	set position and i/o 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
REAL(*#devicenr*)=*expression*	write IEEE 64 bit real number	FILES AND COMMUNICATIONS
REALMC(*#devicenr*)=*expression*	write 64 bit real number in legacy format	FILES AND COMMUNICATIONS
REALMC32(*#devicenr*)=*expression*	write 32 bit real number in legacy format	FILES AND COMMUNICATIONS
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 *filename*	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
STATUSOUTS(*run*,*noerr*,*timeout*)	configure status outputs	I/O CONNECTIONS
STOP	stop program execution	CONTROL
STOPMOVE*axes..*	stop motion	MOTION CONTROL
STOPMOVE(*axnr,..,axnr*)	stop motion	MOTION CONTROL
STRING[(*length*)] *varname*$(dim, ..)..	declare string variables	STRINGS
SYMBOLS	display symbols in use	CONTROL
SYSTEM	exit to McDos	CONTROL
SYSTEM(*string*)	execute McDos command	CONTROL
TASK *address*	create task	STRUCTURE
TASKMAX=*n*	set max. task number	STRUCTURE
TIMER[(*n*)]=..	set timer	TIMING
TRACE {ON|OFF|*tasknr*}	trace program execution	CONTROL
TRIPGROUP*axes.. =expression*	set group of axes for simultaneous trip	MOTION CONTROL
TRIPGROUP(*axis,..*) =*expression*	set group of axes for simultaneous trip	MOTION CONTROL
UNTIL *condition*	exit condition, see DO .. UNTIL .. LOOP	STRUCTURE
WAYMOD$(*n,m*)=	set McWay i/o configuration	I/O CONFIGURATION
WAYSLAVE=*loopnr*	start McWay slave operation	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
ABCONF$(*a*)	read Anybus module configuration	FIELDBUSES
ACCEL*axis*	read axis acceleration	MOTION CONTROL
ACCEL(*axnr*)	read axis acceleration	MOTION CONTROL
ACOMP*axis*	read acceleration feedforward	MOTION CONTROL
ACOMP(*axnr*)	read acceleration feedforward	MOTION CONTROL
ADDR$(*address*)	convert address to string	STRINGS
AND	boolean or binary AND function	MATHEMATICS
ANGLE(*expression,expression*)	calculate vector direction angle	MATHEMATICS
ASC(*string*)	ASCII character to number	STRINGS
ATAN(*expression*)	arcus tangent	MATHEMATICS
BIN$(*expression*)	expression to binary string	STRINGS
BLOCK$(*#expr,expr)*	Read block	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$(*expression*)	number to ASCII character	STRINGS
COS(*expression*)	cosine	MATHEMATICS
CURS$(*xcoord,ykoord*)	set cursor position	FILES AND COMMUNICATIONS
CRC16$(*string*)	CRC16 checksum of a string	STRINGS
DATAPTR@	read data pointer	OTHER COMMANDS
DATE$(*#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
DIGITS	read DIGITS setting	FILES AND COMMUNICATIONS
DIR$(*#devicenr,entry*)	read directory item	FILES AND COMMUNICATIONS
DRIVETYPE*axis*	read axis type	MOTION CONTROL
DRIVETYPE(*axnr*)	read axis type	MOTION CONTROL
ECCO(*node,index,subindex)*	read EtherCat CoE register	FIELDBUSES
ECPAR(*n,m)*	read EtherCat node parameter	FIELDBUSES
ECSERNUM(*node*)	read EtherCat node s/n specification	FIELDBUSES
ENCERR*axis*	read encoder error counter	MOTION CONTROL
ENCERR(*axnr*)	read encoder error counter	MOTION CONTROL
ENCSIZE*axis*	read encoder type	MOTION CONTROL
ENCSIZE(*axnr*)	read encoder type	MOTION CONTROL
ERL	read error line number	ERRORS
ERL$	read error line	ERRORS
ERR	read error number	ERRORS

ERR@	read error line address	ERRORS
ERR$(*errornumber*)	error message	ERRORS
EXP(*expression*)	power of e	MATHEMATICS
FILTERSIZE*axis*	read axis position reference filter setting	MOTION CONTROL
FILTERSIZE(*axnr*)	read axis position reference filter setting	MOTION CONTROL
FLOAT(*#devicenr*)	read real number	FILES AND COMMUNICATIONS
FN*name*(*argument,..*)	user defined numerical function	OTHER COMMANDS
FN*name*$(*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
HEX$(*expression*)	expression to hexadecimal string	STRINGS
IEEE32(*#devicenr*)	read 4 byte IEEE unix format fp number	FILES AND COMMUNICATIONS
IEEE32I(*#devicenr*)	read 4 byte IEEE pc format fp number	FILES AND COMMUNICATIONS
IEEE64(*#devicenr*)	read 8 byte IEEE unix format fp number	FILES AND COMMUNICATIONS
IEEE64I(*#devicenr*)	read 8 byte IEEE pc format fp number	FILES AND COMMUNICATIONS
INP(*input*)	read binary input	INPUT/OUTPUT
INPA(*input*)	read analog input	ANALOG I/O
INSTR(*expr,string,substring*)	locate substring	STRINGS
INT(*expression*)	integer part of number	MATHEMATICS
INTG*axis*	read position control integration	MOTION CONTROL
INTG(*axnr*)	read position control integration	MOTION CONTROL
LEFT$(*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
LOG(*expression*)	natural logarithm	MATHEMATICS
LOG2(*expression*)	2 base 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
LONG(*#devicenr*)	read long integer	FILES AND COMMUNICATIONS
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
MBDATA(*expr,expr*)	read ModBus registers	FIELDBUSES
MBOPEN(*string,expr,expr*)	open ModBus server and read id	FIELDBUSES
MBREG(*expr,expr*)	read ModBus server status	FIELDBUSES
MC$(*string*)	string convert	STRINGS
MID$(*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
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
OVERRIDE*axis*	read axes speed/accel scale	MOTION CONTROL
OVERRIDE(*axnr*)	read axes speed/accel scale	MOTION CONTROL
OVERRIDERATE*axis*	read axes override change rate	MOTION CONTROL
OVERRIDERATE(*axnr*)	read axes override change rate	MOTION CONTROL
PIDFREQ	read position loop repeat rate	MOTION CONTROL
PI	constant pi (3.1415926..)	MATHEMATICS
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
PROGRAMCRC	calculate program CRC16	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 IEEE 64 bit (real) number	FILES AND COMMUNICATIONS
REALMC(*#devicenr*)	read 64 bit real number in legacy format	FILES AND COMMUNICATIONS
REALMC32(*#devicenr*)	read 32 bit real number in legacy format	FILES AND COMMUNICATIONS
RES*axis*	read axis resolution	MOTION CONTROL
RES(*axnr*)	read axis resolution	MOTION CONTROL
REV$(*string*)	string in reverse order	STRINGS
RIGHT$(*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
SOURCECRC	calculate program source CRC16	CONTROL
SPEED*axis*	read axis set speed	MOTION CONTROL
SPEED(*axnr*)	read axis set speed	MOTION CONTROL
SQR(*expression*)	square root	MATHEMATICS
STATUS(*#devicenr,type)*	read device status	FILES AND COMMUNICATIONS
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
TIMER[(*n*)]	read timer value	TIMING
TRIPGROUP*axes..*	read axes trip group number	MOTION CONTROL
TRIPGROUP(*axis,..*)	read axes trip group number	MOTION CONTROL
UCASE$(*string*)	convert string to uppercase	STRINGS
VAL(*string*)	string to number	STRINGS
WAYERR(*waynr*)	read McWay error counter	I/O CONFIGURATION
WAYMOD$(*waynr,modnr*)	read McWay configuration	I/O CONFIGURATION
WAYSLAVE	read McWay slave channel number	I/O CONFIGURATION
WIN$(string)	string convert	STRINGS
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 SKS Control ACN motion and 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 ACN control system models are equipped with 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 installs the correct McBasic software for your ACN control system at factory. The McBasic programming environment and runtime come as an McDOS executable command file, such as BAS32.C4 and is saved on the D8: internal flash memory. It can also be saved and run from other memory devices if desired.

Current versions of McBasic for MPU3 include:

BASIC.C4	McBasic without i/o and motion control functionality for system utility
BAS32.C4	32 logical axes version (standard version)
BAS100.C4	100 logical axes version

2.2 STARTING THE SYSTEM

At power up, the ACN 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.

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

BAS32E [*progname1*, ..., *prognamen*]

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

If no *progname* is specified, McBasic stays in the command mode.

2.3 WRITING PROGRAMS

As the McBasic program file is essentially a text file, programs can be written using many different tools. The recommended tool for working with ACN control systems is the McBench Windows® programming environment supplied with your ACN system.

One way to write a program for the system is to use the ED command in the command mode of the McBasic environment. In this mode program lines can be entered from the console (CN: or wherever the logical console CO: has been redirected to) using a terminal or a PC with a terminal program. Among other communications programs, the Terminal program of the McBench programming environment is well suited for this. Program lines should be entered in the order of their execution.

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 an SD card, USB comms or Ethernet (Telnet). (see McBench documentation).

McBasic supports legacy Basic language numbered program lines. Since numbered lines are not commonly used or recommended for new program development, their use is not described in this manual. However, they may be useful for running old McBasic programs that still contain numbered lines. To learn about using line numbers, please refer to earlier McBasic version documentation.

Generally, program lines are written starting from a position with one or more leading spaces. A program line can be max. 80 characters long. An empty line is also interpreted as a program line.

Labels can be used to define an address in the program in order to use it in conjunction with command such as GOSUB, TASK, GOTO, RESTORE etc. Because of the legacy support for line numbers, a label must not be a number or start with a number although it may contain numbers.

A label must begin from the first position of the line. The rest of the line can be empty of contain a program line.

```
Lab1 ' sample program snippet
 PRINT "first line"
 PRINT "second line"
 PRINT "third line"

 n=3
Lab2 ' the following lines will be repeated 3 times
  PRINT "fourth line"
  PRINT "fifth line"
  n=n-1
  IF n>0 THEN GOTO Lab2
 PRINT "End of program"
```

In McBasic command mode program lines with line numbers can be entered by just typing them and ending the line with a <return>. 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.

2.4 COMMAND AND VARIABLE NAMES

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

```
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 stop the program and 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$(8,8)          'strings and string arrays of 80 character strings
STRING(5) SymFirst$,SymArray(3)    'string of an arbitrary length in the range 1...255,
                                    here 5 characters.
REAL A,Varia(4),B(3,3)             '8 byte floating point numbers (IEEE64 format)
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 (IEEE32 format)
LONG INTEGER A(5)                  '32 bit signed integer (-2147483647...2147483648)
LONG A(5)                          '32 bit signed integer
WORD a(5),c(5,5)                   '16 bit unsigned integer ( 0...65535)
INTEGER Abc(3), Cba(6,6)           '16 bit signed integer (-32768...32767)
BYTE Abc(3), Cba(6,6)              '8 bit unsigned integer (0...255)
```

```
SHORT INTEGER  Cba(6,6)          '8 bit signed integer (-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 condition is generated if the value is not within valid limits.

A single bit in a numerical variable can be referred to using the format *Var.n* to refer to the bit *n* of the variable *Var*. Bit reference *n* can also be a variable or expression. All numerical values can be read using this notation, but only variable values can be set using it. For example:

```
DIM Status
DIGITS=0
Status.0=1
Status.3=1
PRINT "Status: ";Status
FOR n=0 TO 5
  PRINT n,Status.n
NEXT n

Status: 9
0       1
1       0
2       0
3       1
4       0
5       0
```

2.6 LABELS

McBasic uses labels to mark program lines to be referred to from applicable commands. Thus, a label essentially defines an address. 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

```
StartOfProgram
  REAL Abc     'some program lines
  Abc=2
  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 a command addresses a nondefined label
- error #45, when an attempt is made to define a label twice

Labels 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

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=PI*R^2
 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 variable 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 commands controlling the operation of the McBasic programming environment.

3.1 ED

Command	Edit program.
Syntax	ED[*address*]
address	Place (line) in program to start editing. Label or address expression.

ED command allows typing in and editing program from McBasic command mode. It allows editing of previously entered lines and typing in new lines.

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. 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 with <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 = CO:)
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, ..)
MOVEBUFFERaxes
```

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
McBasic32 3.3cg
        Program
        System tables
        Variables & compilations
        Recycled
65628360 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 also 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 (kill) a task. Thus, a program can contain several END commands to end tasks.

For example:

```
  .
  OUT(1000)=1 : TASK Delayed_off(1000,0.5)
  .
  .

Delayed_off(Output,Time)
  DELAY Time
  OUT(Output)=0
  END
```

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 Skip         ' set breakpoint at label Skip
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 a label starting a procedure
```

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....n. 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 useful to insert TRACE commands in the program to control partial tracing of program to preserve processing speed in other parts of program.

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.

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 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)

3.15 SOURCECRC

Calculate checksum of the current source program in memory.

Function	Source program checksum
Syntax	SOURCECRC
Type	Integer
Value	Cyclic redundancy checksum (CRC16) of the current source program.

SOURCECRC can be used as an additional safety measure to verify that the current source program has not changed since the checksum has been calculated. For example, it can be calculated and stored in a variable or also in a file to be able to later recalculate and compare whether the source program has been changed.

3.16 PROGRAMCRC

Calculate checksum of the current source program and compilation in memory.

Function	Source program and compilation checksum
Syntax	PROGRAMCRC
Type	Integer
Value	Cyclic redundancy checksum (CRC16) of the current source program and its compilation.

PROGRAMCRC can be used as an additional safety measure to verify that the current source program or the compilation has not changed since the checksum has been calculated.

PROGRAMCRC can be used the same way as SOURCECRC. It gives some additional security since it also checks the compilation which is actually the program that is running. However, PROGRAMCRC calculated from a program is also different is some of the system software such as the McBasic version have been changed.

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 the first command of the line.
- DEF command must always be the first command of the line.

4.2 GOTO

Command	Jump to given program address.
Syntax	GOTO *addr*
addr	Destination address. Label of address expression.

Execution of the program will continue from addresslabel *nnn*.

```
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 MyLabel
```

starts the program from label MyLabel. Address expressions can also be used:

```
GOTO Label1+3

A@=Label1+4
GOTO A@         ' A@ is an address variable
```

Generally, use of the GOTO command especially for long jumps in the program is not good programming practice, since it makes the program flow difficult to follow. Using subroutines/procedures and conditional structures should be preferred.

4.3 GOSUB

Command	Sub-routine or procedure call.
Syntax	GOSUB *addr* or GOSUB *label*(*par1*,*par2*,...)
addr	Address of the first line of the subroutine.
label	A label in the beginning of the procedure.
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.

```
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:

```
  GOSUB MySub : STOP

MySub PRINT "Subroutine"
  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*.

```
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).

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

Operation is similar to ON .. GOTO structure.

4.7 IF THEN [ELSEIF] [ELSE] [ENDIF]

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 *commands* after THEN. Program execution then continues from next line or ENDIF, if used.
ELSE	If *condition* was not true, allows the execution of *commands* 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 (:).

```
  IF K=0 THEN PRINT "Zero division" ELSE GOTO Lbl1
  IF K><0 THEN GOTO Lbl2
Lbl1 IF A<B THEN C=B-A ELSE C=A-B : ENDIF : A=0
Lbl2 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 *commands* 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 *commands* and command lines after next THEN. Program execution then continues from after ENDIF
ELSE	If all preceding conditions were false, allows the execution of *commands* 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 loop variable 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.

```
 FOR N=1 TO 100
 PRINT N
 IF INP(32) THEN N=100 : GOTO EndOfLoop
 OUT(32)=NOT OUT(32)
EndOfLoop NEXT N
```

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

```
 FOR N=1 TO 100
 PRINT N
 IF INP(32) THEN N=100 : NEXT N : GOTO Lbl1
 OUT(32)=NOT OUT(32)
 NEXT N
 END

Lbl1 PRINT "Finished earlier."
 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 is less or equal to the PRIOR setting. The wait-counter of the task is set to its PRIOR value.

5. MATHEMATICS

Mathematical calculations in McBasic 3.3 are performed with 8 byte (IEEE754 binary64) floating point numbers with a precision of 15 significant numbers. The range of values is $\pm 10^{-308}...10^{308}$.

Values exceeding the range will produce "Infinity" or "-Infinity" and undefined results "Not-a-number" when printed with the PRINT command:

```
B>PRINT  1/0,-1/0,0/0
Infinity        -Infinity       Not-a-number
```

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 bitwise 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:

```
IF A>B AND (INP(32) XOR INP(33)) THEN GOSUB Sub1
```

5.3 BINARY OPERATIONS

Binary operations can be used also between other integers than 0 and 1 up to 49 bits as follows.

AND	AND-operation for a 49-bit integer
OR	OR-operation for a 49 bit integer
XOR	XOR-operation for a 49 bit integer

In this case the result is also a max. 49-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
0x41 0xBFC0 0x0020 0x100000	alternative hexadecimal format
%11 %01110101 %1111111111	binary format
0b11 0b01110101 0b1111111111	alternative 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
OUT(45)=ON
```

5.5.2 OFF

Function	Constant 0.
Syntax	OFF

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

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

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	RND(*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. -708 ... 709 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 base 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 LOG2

Function	Base 2 logarithm.
Syntax	LOG2(*expression*)
Type	Real number.
expression	Real number.
Value	$\log_2$(*expression*) Base 2 logarithm of *expression*. With negative values of expression the function returns the absolute value of the logarithm of *expression*.

```
B>PRINT LOG2(1),LOG2(2),LOG2(4)
0.00            1.00            2.00
```

5.5.12 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
```

5.5.13 PI

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

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

5.5.14 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
```

5.5.15 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
```

5.5.16 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
```

5.5.17 ATAN

Function	Trigonometric arc tangent.
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)
0.0  0.78
```

5.5.18 ANGLE

Function	Calculates vector angle
Syntax	ANGLE(*xx,yy*)
Type	Real number. Unit radians.
xx	Vector X component
yy	Vector Y component
Value	Vector angle 0 ... 2π [rad]

6. STRINGS

A string is an expression consisting of 0..255 characters. Thus a string is essentially a piece of text, although it may contain any 8bit value in each character position. String values of visible text are assigned in quotes:

```
PRINT "Hello"
```

String variables are variables holding a string value. McBasic string variable names 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) commands. 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)+"[101;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 switching would occur during 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("PI"),VAL("1E2"),
X=PI
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
DIGITS=2
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 (-10^14 ... 10^14).
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^{53} ... $2^{53}-1$).
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 REV$

Function	Reorder string backwards.
Syntax	REV$(*string*)
Type	String
string	String to reorder.
Value	*string* in reverse order.

```
PRINT REV$("ABCDEFG")
GFEDCBA
```

6.14 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.

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

```
3.00
```

6.15 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$:

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

6.16 UCASE$

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

```
First$="abcdef"
Second$=UCASE$(First$)
PRINT First$,Second$
```

```
abcdef   ABCDEF
```

6.17 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 address 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
```

6.18 MC$

Command	Windows string convert to 7 bit ASCII
Syntax	MC$(*string*)
string	String to convert.
value	String converted in 7 bit ASCII character set

6.19 WIN$

Command	Convert 7 bit ASCII string to Windows character set
Syntax	MC$(*string*)
string	String to convert.
Value	String converted in Windows character set

6.20 CRC16$

Command	Calculate CRC16 2 character cyclic redundancy check of a string
Syntax	CRC16$(*string*)
string	Input string.
Value	String containing the 2 characters of the CRC16 of *string*.

CRC16$ is useful to calculate the popular CRC16 checksum as used in protocols such as MODBUS RTU. As the output of the function is in string format, it can easily be added to the the message when sending or compared with the checksum included in a received message.

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, Pix2, 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..),..
LONG 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 3.3 variables have a scope according to the program structure where they were declared. 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, LONG 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 a 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 integer 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.

```
DIM ArrOne(12),M(5,5),B$(5)
FOR I=0 TO 12
 ArrOne(I)=-1
NEXT I
FOR I=0 TO 5
 M(I,I)=1
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 positive integer values.
By default the lower limit is assumed 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 ±1.7E-307 to ±1.7E308 with a resolution of 15 significant numbers.

```
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 ±0.2E-37 to ±3.4E38 with a resolution of 7 significant numbers.

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 (8bit unsigned) 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 (16bit unsigned) 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 (8bit signed) 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 (16bit signed) 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 LONG INTEGER

Command	Declare long (32bit signed) integer arrays.
Syntax	LONG [INTEGER a*rrname*(*expression1,...,expressionn), ...* *or* LONG [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*)

Long integer array cells can get integer values between -2147483648 and 2147483647.

7.10 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

>
```

7.11 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.

7.12 [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 COMMUNICATIONS

In McDos/McBasic systems, data input and output can be done through serial ports, ethernet, or by reading and writing files.

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

An exception to this rule are the program file operations described in 8.2 PROGRAM FILES. Program file commands can refer to files using their path/name only like many McDos commands.

8.1 DEVICE NUMBERS

Both physical device names and logical device names are used for ports and files. The syntax for logical device names is #*n*, where *n* is an integer between 1..99. These are also called device numbers. Various system software versions may limit the maximum available device number to less than 99.

Under McBasic, a physical device, whether a port or a memory file, is referred to by its device number, for which it has been opened.

After opening, files are referred to by their device number using command and functions described in 8.3 DATA INPUT AND OUTPUT. In case of an error or when exiting McBasic all open files are automatically closed. Same commands can generally be used for both files and ports.

Physical device names can be opened for any device number using the OPEN command. Similarly files can be opened for any device number. The unopened device numbers refer automatically to the current console device (CO:).

Opening and closing input and output files and ports is explained in more detail in conjunction with each device type**.**

8.2 PROGRAM FILES

Under McDos operating system the McBasic environment is called from a memory device by giving the name of the interpreter/runtime program as a command. The operating system loads and starts the McBasic programming environment, interpreter and compiler stored in the command file such as BAS32.C4. 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

```
D4:\>BAS32 MYPROGRAM
D4:\>BAS100 PROG1,PROG2,PROG3
```

8.2.1 STARTING MCBASIC

At power-up, the ACN control system main memory is empty. Usually a startup file called WAKEUP.EX is used to start McBasic and load the application program in a finished application. For development/testing the program can be manually loaded from the McBench programming environment.

8.2.2 USING WAKEUP.EX

Under McDos operating system, start up can be automated by saving a WAKEUP.EX file in the root directory of a mass memory device available in the system at startup. McDos looks for a WAKEUP.EX file starting from the lowest device (D1: ... D8:). The first WAKEUP.EX file found will be executed. WAKEUP.EX can contain McDOS commands such as

```
D8:/BAS32
```

which would start the McBasic environment located in D8: root and remain in command mode.

or

```
PATH D8:/
D8:/BAS32 D4:/PROGRAM
```

which would set the command search path to include first D0: (the alias for the current directory) and secondly D8:/ (the system directory where system software is stored when the controllers are shipped). Then it would start McBasic from D8: root and load a program file PROGRAM.BA from D4: root and start executing it.

It is generally a good idea to include a PATH command in the WAKEUP.EX file setting at least D8:/ to be included in the search path to allow non-resident McDos commands to be accessed independent of current device. This is necessary for example for McBench McFiles utility to be able to work.

Once started, program execution can be stopped from console (CO:) 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.

The default WAKEUP.EX file stored in and ACN MPU D8: root directory contains the following commands:

```
* SKS Control ACN default WAKEUP.EX file
* 2.4.2012
* CONSOLE 38K
SET CN:38K
PATH D8:/
*  address     ,subnet mask  ,gateway addr
IP 192.168.0.20,255.255.255.0,192.168.0.1
CNTO TELNET://
D8:/BAS32 D6:/PROGRAM.BA
```

The default WAKEUP.EX contains commands to set the USB console data transfer speed to 38Kbit/s and the command search path to include D8: root. The IP address of the system is set to 192.168.0.20 with a subnet mask 255.255.255.0 and gateway address 192.168.0.1.

The CNTO command sets the system to accept connecting to the console via TELNET. This allows the use of the TELNET connection from McBench, for example.

Finally, the McBasic version BAS32 (the default version in ACN systems) is started from D8:/ and a program PROGRAM.BA is loaded from D4:/ and started. This program name is included as an example, if such a program file does not exist in D4:/, McBasic is started and remains in command mode.

8.2.3 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/path, that has been used last ("current directory"). Program is saved with the given file name. If a file with the same name already exists, it is overwritten.

```
SAVE "TEST7"
SAVE "D4:TEST8.BA"
```

8.2.4 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"
```

8.2.5 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"
```

8.3 DATA INPUT AND OUTPUT

For data input and output, number of commands and functions are available for use with any type of device.

8.3.1 INPUT

Command	Read data from a device.
Syntax	INPUT[#*nn*],[*string*{,|;}]*variable*[,..,*variable*][;]
nn	Device number (1 ... 99), for which file or port was opened. If no device number is given, #1 is assumed (usually =CO:, the console port).
string	If *string* is given, it will be echoed to the device as a prompt, if the device is a serial or TCP port.
{,|;}	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 read values to (string or numerical expressions). When reading from file, the numerical data much be ASCII numbers with a comma (,) separator (CSV). Reading a string will return the contents of the rest of the line, i.e the text from the file pointer (PTR) to the next end of line <cr>.
[;]	If there is a semi-colon in the end of the command, no line feed (cr+lf) is echoed after reading from a port.

When INPUT is used to read from a serial or TCP port, prompts can be used and input is echoed back to the port to facilitate manual data input from devices such as data terminals.

When reading from files, no data is written to the file.

McBasic program task, where an INPUT command is encountered, stops until necessary data is received from device nn. If other tasks exist, they may continue to be executed while INPUT is waiting.

In cases where it is necessary to be able to observe whether data is available for reading from the device, other means, such as the BYTE(#n) function, should be used for reading.

```
INPUT #4,X,Y,A$
INPUT #3,A$
' semi-colon at the end
' prevents line feed
INPUT "ENTER A NUMBER",N;
PRINT " number was ";N
' semi-colon
' prevents echoing question mark
INPUT "ENTER ANOTHER NUMBER ";N
PRINT N
RUN

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

8.3.2 PRINT

Command	Output to a device.
Syntax	PRINT[#*nn*,][*expression*][{,|;}...{,|;}*expression*][,|;]
nn	Device number (1 ... 99), for which the file or port has been opened. If no device number is given, #1 is assumed (usually CO:, the console).
expression	Numerical or string *expressions* to be output. Without *expressions* the command can be used for line feed. The values of numerical *expressions* are automatically converted to strings for output using the current DIGITS setting.
{,|;}	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 a 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

DIGITS=3                        ' set 3 decimals
PRINT 2+5

7.000
```

Generally it is good practice to read from and output to a device from only one task. 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 can be accomplished by using flag variables to time the operation of tasks or by locking tasks with the PRIOR setting.

When using PRINT to output data to various devices, it may be necessary to take care that data output is not modified unintentionally. PRINT assumes that the output is text, and therefore removes <nul> and <lf> characters ($00 and $0A) from the data when printing to file or TCP port. This is done to preserve the text file format used in McDos. To output binary data, the BLOCK$ command is recommended instead.

8.3.3 LIST

Command	List program.
Syntax	LIST [#nn,][*address1*][,[*address2*]]
nn	By giving device number *nn* program will be listed to device #*nn*, otherwise to device #1.
address1	Address of the first line to be listed. Default is start of program. Can be an address expression such as Label+n, the address of the n:th line after Label. 0 represents the start of the program.
address2	The last line to be listed. Default is 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 Lab1,Lab2      ' from line Lab1 to line Lab2 (including)
LIST Lab1,          ' from line Lab1 to the next empty line
LIST 0+5,Lab2+6     ' from 5th line from program start
                    ' to 6th line after Lab2
```

8.3.4 DIGITS

Command	Set the number of decimals used when printing or making numeric to string conversions.
Syntax	DIGITS=*expression*
expression	Number of decimals (0...20) 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.

Function	Read current DIGITS setting.
Syntax	DIGITS
Type	Number (0…20.5)
Value	Current DIGITS setting in the task.

Number of decimals used in printing is defined by the value of expression. Default number of decimals when starting an McBasic program is 2 decimals. DIGITS setting is local in tasks and is inherited from the creator of a task. Thus changing DIGITS in another task does not affect printing or conversion operations in the current task.

```
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    3.141592
7.0000000   3.1415926
8.00000000  3.14159265
9.0000000003.141592653
```

8.3.5 BYTE(#nn)

Command	Output a (8 bit) 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.

Function	Read a (8 bit) byte from a device.
Syntax	BYTE(#*nn*)
Type	Integer (0 .. 255)
nn	Device number to read from. Can also be a variable or expression.
Value	Value of the received byte. When text, the ASCII code of the character. If there are no bytes (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 working with ASCII text, bytes are generally visible or control characters and their value is the ASCII code of the character.

When writing numerical values to files using BYTE, WORD, FLOAT, REAL or IEEE.. commands, they are saved as binary data 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.

```
BYTE(#3)=66 ' write 66 (ASCII code for "B")


A=BYTE(#4)
IF A<>67 THEN GOSUB Subroutine ' call if "C" received
```

8.3.6 WORD(#nn)

Command	Write a (16 bit) 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 (16 bit) 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.

In WORD the most significant byte is assumed to be the first byte.

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).

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

8.3.7 LONG(#nn)

Command	Write a (32 bit) integer (4 bytes) to a device.
Syntax	LONG(#*nn*)=*expression*
nn	Device number. Can also be a variable or an expression.
expression	Value to be written, integer (0 ... 4294967295).

Function	Read a (32 bit) integer from a device.
Syntax	LONG(#*nn*)
Type	Integer (0 .. 4294967295)
nn	Device number to be read from. Can also be a variable or an expression.
Value	Value (0 ... 4294967295) of the received integer. If there were not 4 bytes available in device buffer or the file did not contain 4 more characters, the function returns value -1.

In LONG the most significant byte is assumed to be the first byte.

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

8.3.8 FLOAT(#nn)

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

Function	Read a (4 byte) floating point number from a device.
Syntax	FLOAT(#*nn*)
Type	Real number
nn	Device number to 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 bytes, 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).

```
FLOAT(#3)=1000*PI
FLOAT(#4)
```

8.3.9 REAL(#nn)

Command	Write a floating point number to a device.
Syntax	REAL[MC|MC32](#*nn*)=*expression*
MC	Legacy Arlacon MC 64 bit floating point number format indicator. If omitted, IEEE64 format is assumed
MC32	Legacy Arlacon MC 32 bit floating point number format indicator. If omitted, IEEE64 format is assumed
nn	Device number to write to. Can also be a variable or an expression.
expression	Value to be written, real number.

Function	Read a floating point number from a device.
Syntax	REAL[MC|MC32](#*nn*)
MC	Legacy Arlacon MC 64 bit floating point number format indicator. If omitted, IEEE64 format is assumed
MC32	Legacy Arlacon MC 32 bit floating point number format indicator. If omitted, IEEE64 format is assumed
Type	Real number
nn	Device number to read from. Can also be a variable or an expression.
Value	Received real number. If enough data (8 or 4 bytes) was not available from device buffer or file, the function returns 0.

When writing to or reading from a file, the file pointer PTR(#nn) is automatically incremented by 4 (32 bit) or 8 (64bit).

Because REAL requires 4 or 8 character ready for reading, it can mainly be used with files. When using serial ports, the use of BYTE function is recommended.

8.3.10 IEEE

Command	Write an IEEE format floating point number (4 or 8 bytes) to a device.
Syntax	IEEE*nn*[I](#*devicenr*)=expression
nn	bits in format, 32 (4 bytes) or 64 (8 bytes)
I	PC style format indicator. If omitted, unix style is assumed
devicenr	Device number to 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 (4 bytes) or 64 (8 bytes)
I	PC style format indicator. If omitted, unix style is assumed
Type	Real number
nn	Device number to 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 allows input and output in four different popular floating point number formats according to the IEEE 754 standard. IEEE64 is the binary64 format in the standard and is the format used by McBasic 3.3 internally.

8.3.11 BLOCK$

Function	Write a string to device
Syntax	BLOCK$(#*nn*)=*string*
nn	Device number to write to. Can also be a variable or an expression.
string	String to be written to the device

Function	Read a string from a device
Syntax	BLOCK$(#*nn,length*)
Type	String
nn	Number of device to read from
length	Length of string to read (bytes, characters)
Value	String containing the specified number of bytes in the same order they were read. If less than *length* bytes were available for reading from the device, the length of the string will be shorter, accordingly.

BLOCK$ is useful for reading and writing data from and to devices especially when working with ports such as serial ports or TCP ports and binary data. This might be the situation when writing binary communications protocols, for example.

Unlike PRINT or INPUT, BLOCK$ does not convert data formats or otherwise modify data in any way.

8.3.12 DATE$

Function	Convert date in number format to string.
Syntax	DATE$(*a*)
Type	String
a	Date in number format (as produced by DATE function) to be converted to string.
Value	Converted number format date in form *yyyymmddhhmmss* (4 digit year).

DATE$ function provides means to convert date calculation results from number format back to string format and to convert 2 digit year formats to 4 digits. Notice that the DATE$ function is also used for reading system real time clock or file dates (see 8.6.6).

Example:

```
PRINT DATE$(DATE(DATE$(#1)))
20110825135344
```

8.3.13 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
```

8.3.14 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 devices #2 and #3 to #10
LINK#11,10,7     'link devices #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 devices #2 and #3 are empty, loop

CLOSE #3 : OPEN #3, "CN:" 'change device #3 to CN:
PRINT #11,"STOP"
CLOSE #10 : CLOSE #11          'break links #10 and #11
CLOSE #2 : CLOSE #3 : CLOSE #7 'closing primary devices
```

8.4 CURSOR CONTROL FUNCTIONS

Cursor control functions are provided for controlling text output to display terminals or files etc. They are compatible with MC300 and MC400 displays and some standard terminals and terminal programs.

8.4.1 TAB

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

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

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

8.4.2 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.

The LINE command sets an automatic change of line at the specified lenght.

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

8.4.3 CURS$(column,row)

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

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

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

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

8.4.4 ANSICURS$(column,row)

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

Like CURS$(nn,nn), but uses ANSI standard escape sequence.

8.5 SERIAL COMMUNICATIONS

In ACN systems, all serial ports are available for freely programmable asynchronous serial communications during program execution. One of the ports, the CN: or console port, is acting also as a programming/monitoring port by default. When the McBasic program is not running (McBasic command mode) or when in McDos, the console port provides the connection to control the system. The logical console (called CO:) can be redirected from CN: to other ports including TELNET using the McDos CNTO command (see McDos User's Manual), leaving the CN: port free for other use by the application program..

During program execution, also the console port can be used just like any other port. Only differences are that a ctrl-X received from the console causes the program execution to stop and in case the program stops for any reason, such as might be an error in program execution, the system will enter the command mode and all messages, prompts and eventual interaction with the system will happen through the current logical console.

While any applicable terminal device or program can be used to connect to the console port, the McBench programming workbench for Windows provides the best tools for working with the system and developing McBasic programs. McBench also uses the console connection to connect to the system.

Another logical device name PR: is reserved for the logical printer. By default PR: is connected to the LP: port, but it can be redirected to other ports or file services using the McDos PRTO or PRTONET commands (see McDos User's Manual).

8.5.1 OPEN

Command	Open a serial port.
Syntax	OPEN #*nn*,*string*
nn	Device number (1 ... 99), for which the serial port is opened.

string	serial port name and transfer parameters in the form:
	"*device:[parameters]*"
	
	`OPEN #2, "LP:9K68E11"`
	
	`where`
	`device name`
	`CN:, LP:..`
	
	`baud rate`
	`300,600,1K2,`
	`2K4,4K8,9K6,`
	`19K,38K`
	
	`number of data bits`
	`7,8`
	
	`parity`
	`0 space`
	`1 mark`
	`E even`
	`O odd`
	`N no`
	`number of stop bits`
	`1,2`
	`xon/xoff handshake`
	`0 not used`
	`1 used`
	`R xon repeated every 5s`

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

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

A serial 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)

After opening, data can be read from ports and written to them with applicable commands and functions described in chapter 8.3 DATA INPUT AND OUTPUT.

For example :

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

At the time of system start up (before opening files and ports) all device numbers refer to the logical console CO: which by default is CN: (to redirect see McDos CNTO command).

Systems running under McDos 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.

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:.

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.

8.5.2 ACN serial ports

The following serial devices are available in SKS Control ACN systems:

SKS Control ACN MPU3 processor module has 2 serial ports and 4 ports that can be used either as McWay I/O loops or serial ports.

physical name	device
CN:	console port, USB
LP:	second serial port
S0:	W0: McWay I/O loop, usually used for local I/O modules
S1:	W1: McWay I/O loop, alternatively auxiliary serial port
S2:	W2: McWay I/O loop, alternatively auxiliary serial port
S3:	W3: McWay I/O loop, alternatively auxiliary serial port

ACN MPU serial port S0:-S3: operation is defined automatically using either the OPEN command to select serial port mode or the WAYMOS$ command to select McWay mode.

8.5.3 CLOSE

Command	Close a serial port.
Syntax	CLOSE #*nn*
nn	Device number (1 ... 99), for which the serial port was opened.

To finish using a serial port. When closing serial ports the communication parameters remain as set, input and output for device *nn* are redirected to CN:.

8.5.4 SIZE

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

The size of the serial port output buffer is specific for any physical device. ACN MPU3 serial devices have a buffer size of 255 bytes, so if no output is in the queue, SIZE(#*nn*) will return 255. A smaller value will indicate, that there is data waiting to be sent.

To achieve the best possible timing consistency, it is advisable to output to a port when the whole buffer is empty (SIZE(#*nn*)=255). Thus the data will be sent with a minimum delay after putting it in the queue.

Generally, it is a good idea to avoid putting data in the queue when the buffer is full (SIZE(#*nn*)=0), because this will cause the program task to stop at the output command (such as PRINT, BYTE, etc.) and wait for space to become available in the buffer.

For example:

```
DO
 IF SIZE(#2)>254 THEN PRINT #2,POSX,POSY
LOOP
```

8.5.5 STATUS

The STATUS function allows studying the statuses of various communication connections in conjunction with serial ports or Ethernet. The following describes the use of STATUS in connection with serial ports.

Command	Read serial port status.
Syntax	STATUS(#*nn,i)*
nn	Device number of the port
Type	Integer
Value	Status information of device #*nn* as specified below.
STATUS (#*nn*,0)	type of device, 1 serial device CN: LP: S0: S1:
For serial ports STATUS (#*nn*,*i*)	(where STATUS (#*nn*,0)=1) *i* 1 device: 0=CN:, 1=LP:, 2=S0:, 3=S1 .... n+2=Sn 2 transfer rate [bit/s] 6 rx, number of received unread bytes in buffer 7 tx, number of unsent bytes in buffer

Use STATUS (#*nn*,0) to determine if the device *nn* is a serial device. Use other values of *i* to obtain information about the device status.

8.6 MEMORY DEVICES AND FILE OPERATIONS

Memory devices in ACN systems are mass memory devices holding data that can be accessed sequentially or randomly. They can be located in FLASH or RAM memory or on a file server through Ethernet network. Depending on the memory type and system, there are specific properties concerning writing and accessing the memory.

There is a default memory device configuration for the system, initialised automatically when starting the system or using the McDos RESET command. Additional memory devices can be mounted and configured using the McDos SET command (see McDos User's Manual).

Memory devices have a file system, allowing data to be organised as files in directories. Depending on the device, McDos McFS or industry standard FAT systems may be used.

Default memory device configuration for ACN MPU3:

Device	
D1:	
D2:	
D3:	External flash memory card if inserted (SD/SDHC, FAT16/32 or McFS).
D4:	Internal flash memory partition for applications etc.
D5:	
D6:	
D7:	512K RAM drive. Cleared at power-off
D8:	Internal flash memory partition for system files.

Use the McDos SET command to view or change the current device configuration.

Any device, especially those without default function, can be connected to a Netbios file server share through Ethernet using the McDos NETUSE command.

Please refer to the McDos Operating System User's Manual for details.

8.6.1 OPEN

Command	Open a file
Syntax	OPEN #*nn*,*string*
nn	Device number (1 ... 99), for which the file is opened.
string	File name in the form [*memorydevice*:][*path*]*name*[.*extension*] default *memorydevice* is D0:, the current device default *extension* is .TX default *path* is the current path

After opening, files 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 the logical console CO: which by default is CN: (to redirect see McDos CNTO command).

Most important extensions :

.BA	McBASIC program file
.EX	McDos batch file
.DT	binary data file
.TX	text file
.CK	MC300, MC400 CPU5 generation command file
.CF	ACN MPU[2], MC400 CPU6 generation command file
.C4	ACN MPU3 generation command file

8.6.2 CLOSE

Command	Close a file.
Syntax	CLOSE #*nn*
nn	Device number (1 ... 99), for which the file was opened.

To finish using a file. The file is closed and possible data in buffer is written to the memory device. The device number *nn* is released and will point to CO: (the logical console) until otherwise opened.

8.6.3 PTR

Command	Set 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 or writing data from/to any part of a file and thus use the file as a random access file.

```
OPEN#3,"D1:\FILE.DT"
PTR(#3)=0            'first character
A=BYTE(#3)           'is read to A
PTR(#3)=SIZE(#3)-1   'last character
B=BYTE(#3)           'is read to B
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 initialized, so it may contain empty (zero value) bytes or some data that has been deleted before.

8.6.4 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 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.

SIZE command is also used to flush UDP and TCP packets that have been written to. Setting SIZE(#*nn*)=0 for an UDP or TCP device forces the current packet to be sent. When flushing, the contents of the packet is sent to its destination. In a TCP connection, packets are flushed automatically if they get full. To complete sending the rest of the written data, the last TCP packet has to be flushed.

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 the end part of a file
CLOSE(#3)              'beginning from PTR
```

```
 OPEN#4,"FILE.TX"
 SIZE(#4)=0 : CLOSE(#4) 'delete a whole file

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

Prnt(M$)
 'sub-routine, prints M$ to #2 which has an output buffer
 'of 255 bytes, when buffer is empty
 DO WHILE SIZE(#2)<255 : LOOP
 PRINT#2,M$
 RETURN
```

8.6.5 DIR$

Function	Read 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 "nnnnnnnneee" (11 characters). nnnnnnnn file name eee extension A name of a subdirectory is an entry in form "nnnnnnnneee/" (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,"D8:"
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
 n=n+1
 LOOP

RUN
```

```
40       .C4     BAS100   .C4     BASE     .C4     DATE     .BA     WAX2     .BA
MCDOS7   .C4     VDEMO    .BA     WAX2A    .BA     BASIC    .C4     ZM       .C4
SER      .BA     TX       .C4     WAKEUP   .EX     WMS2     .BA     CPU6     .BA
NBER     .TX     KELLO    .BA     BAS32    .C4     DISK     .C4     SETCLOCK.BA
```

8.6.6 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
Syntax	DATE$(#*nn*)
Type	String
nn	Device number, for which the file has been opened.
Value	Date in form *yymmddhhmmss* (see above). If #*nn* refers to the console device, the value is the current date/time from the real time clock.

Device CN: (console), usually #1, is the real time clock of the system (see chapter 10.1 REAL TIME CLOCK)

Files are automatically dated according to the console date when written to a disk (closing), if they have been modified.

For example string 110224123456 represents the date February 02, 2011 and the time 12:34:56

```
DATE(#1)="110224123456"
PRINT DATE$(#1)
```

```
110224123456
```

8.7 NETWORK

Network communications is available in McBasic to work with a TCP/IP Ethernet network connected to the system. McDos provides services for McBasic to use TCP and UDP transport protocols from within McBasic programs. McDos can also connect to NetBios servers to use file and printer shares and thus provide access to these services for McBasic application programs as well (see McDos User's Manual).

ACN MPU3 has 2 Ethernet controllers, E1: and E2:. E1: is the default active IP network while E2: can be used as an EtherCat fieldbus master connection.

Setting the IP address of the system and connecting to NetBios shares is done in McDos and thus they are usually initialised in the WAKEUP.EX (see 8.2.2) while starting the application or from within the McBasic program using the SYSTEM command (see 3.4).

After setting the IP address and connecting the possible shared services to devices, NetBios services can be connected to device numbers as any other device and accessed accordingly like memory devices or printer connected to serial port.

TCP and UDP transport services can be used with the following commands:

8.7.1 OPEN

Command	Open a network port.
Syntax	OPEN #*nn*,*string*
nn	Device number (1 ... 99), for which the file or port is opened.
string	**if TCP port:** Open for listen, another system can establish a connection: "*port*://0.0.0.0:0" Open connection to another system listening: "//*nnn.nnn.nnn.nnn:targetport* *where* *nnn.nnn.nnn.nnn* is the ip adress and *targetport* the port number of the device to connect to **if UDP port** Open read socket: "UDP:*port*://[*maxpackets*]" Open transmit packet: "UDP://*nnn.nnn.nnn.nnn*:*targetport*"
port	ACN MPU port to start listening for connection.
nnn.nnn.nnn.nnn	Target system IP address (each *nnn*=0...255)
targetport	Target system port to connect or send to.
maxpackets	Maximum number of ethernet packets the system will queue. (Default = all available buffers). Limit the number of packets to avoid running out of buffer space if UDP packets are not handled as they are received.

After opening, ports can be read and written with applicable commands and functions (see 8.3). When using TCP protocol, the data must be written to a socket and read from a socket sequentially. While packets are sent as they get full when writing to the port, it is necessary to issue a command SIZE(#*nn*)=0 for the devicenumber of the port to release (send) the last packet.

UDP packets can be worked with like files. The maximum SIZE of each packet is 1499 but can in practice be limited by the network infrastructure. Typically at least 1400 byte size packets can be used. The current size of a packet can be read using the SIZE(#nn) function.

When opening an UDP packet for transmit, the SIZE(#*nn*) of the packet is initially zero. When writing to the packet, the SIZE(#*nn*) reflects the size of the contents of the packet and PTR(#*nn*) is incremented as when wiriting to files. By setting PTR(#*nn*), the packet can also be random accessed as a file.

Closing the UDP packet with CLOSE(#*nn*) send the contents of the packet and sets SIZE(#*nn*) and PTR(#*nn*) to zero. Also setting SIZE(#*nn*)=*b* can be used to send *b* bytes from the packet and reset it. Using this method to send avoids the need to use the OPEN command before preparing the next packet.

For more information on Ethernet usage refer to the McDos user's manual.

8.7.2 CLOSE

Command	Close a port.
Syntax	CLOSE #*nn*
nn	Device number (1 ... 99), for which the file or port was opened.

To close a TCP socket or UDP port. A UDP packet will be sent when closing, if its SIZE is greater than zero. A TCP socket will be closed according to the TCP closing sequence, including sending any pending data. Device number #*nn* is released and can be used again immediately.

8.7.3 STATUS

The STATUS function allows studying the statuses of various communication connections in conjunction with serial ports or Ethernet. The following describes the use of STATUS in connection with TCP or UDP transport.

Command	Read TCP or UDP port status.	
Syntax	STATUS(#*nn,i)*	
nn	Device number of the port	
Type	Integer	
Value	Status information of device #*nn* as specified below.	
For any any type of device STATUS (#*nn,i*)	where *i*=	
	-3	number of free TCP connections
	-2	number of free ethernet buffers
	-1	number of free file buffers
	0	type of device
		0 XX: waste basket
		1 serial device CN: LP: S0: S1: …
		2 TR: Terminal in MC300
		3 McNet (legacy communications serial network)
		4 Dx: file
		5 R1: legacy RAM file
		6 TCP TCP/IP socket
		7 UDP UDP socket

For TCP sockets STATUS (#*nn*,*i*)	(where STATUS (#*nn*,0)=6) where *i*=	
	1	own IP address (decimal value)
	2	own port number
	3	target IP address (decimal value)
	4	target port number
	5	connection status values: 0 closed 1 listen 2 syn send 3 syn received 4 established 5 fin-wait1 6 fin-wait2 7 close wait 8 closing 9 last acknowledge 10 timeout wait 11 closed, device still reserved
	6	rx, number of received unread bytes in buffer
	7	tx, number of unsent bytes in buffer
	8	rx pointer
	9	tx pointer
	10	last+1
	11	tx acknowledged
	12	rx, number of urgent unread bytes in buffer
For UDP sockets STATUS (#*nn*,*i*)	(where STATUS (#*nn*,0)=7) where *i*=	
	1	own IP address (decimal value)
	2	own port number
	3	target IP address (decimal value)
	4	target port number
	5	packets in input queue
	6	rx, number of received unread bytes in buffer

For example passive establish and study the status of a TCP socket:

```
OPEN #10,"10000://0.0.0.0:0" ' start listening at port 10000
PRINT STATUS(#10,0)          ' type of device
PRINT STATUS(#10,5)          ' connection status
DO : UNTIL STATUS(#10,5)=4   ' wait for connection
PRINT STATUS(#10,5)          ' connection status

6                            ' TCP socket
1                            ' listening
4                            ' established
```

8.7.4 SIZE

Command	Flush TCP or UDP packet.
Syntax	SIZE(#*nn*)=*x*
nn	Device number, for which the port has been opened.
x	To flush a TCP socket, i.e. send all pending data in transmit buffer, set SIZE(#*nn*)=0. To send a UDP packet, set SIZE(#nn) to the size of the packet to be sent, for example: SIZE(#*nn*)=SIZE(#*nn*) When sent, the size of the UDP packet is reset to zero automatically. To discard a received UDP packet, set its size to zero SIZE(#*nn*)=0. This will delete the packet and give access to the next received packet.

Function	Read UDP packet size
Syntax	SIZE(#*nn*)
Type	Integer
nn	Device number, for which the socket has been opened.
Value	Size of data in the packet (bytes). When using for a TCP port the function returns 255.

The size of an Ethernet package limits the maximum data size in a single packet. Generally, the maximum packet size in McDos is 1499 bytes, although it can be limited by the infrastructure of the network. Normally it is safe to use up to 1400 bytes packets.

With TCP sockets, transmit packets are automatically sent when they get full and read packets are accessed sequentially as they come in, so the only thing to worry about is sending the last packet. In cases, where a message is sent by TCP, it is therefore typical to flush the last packet (set SIZE(#*nn*)=0) after writing all data to the port. To observe the number of bytes in TCP transmit and receive buffers, please use the STATUS(#*nn*,6) and STATUS(#*nn*,7) functions.

With UDP packets, every transmit packet is always prepared and sent before working on the next one, so it is necessary to control the sending of each packet. Similarly, received packets are handled each in turn, so it is also necessary to control moving on to the next packet. Again, the STATUS(#*nn*,5) and STATUS(#*nn*,6) functions can be used to observe the status of the received packets.

9. FIELDBUSES

9.1 MODBUS

ModBus is a standard fieldbus protocol that can be used either in conjunction with serial or Ethernet communications. ModBus protocol support for operation as ModBus RTU, UDP or TCP slave (server) is available in McBasic firmware. For master operation please use the McBasic ModBus master library software available from manufacturer.

ModBus RTU is designed for use with serial communications and thus occupies a serial port from the system. It can be used either as point to point between one master and one slave device, or in a multidrop configuration with one master and several slave nodes. A selection of physical connections can be used. The following table illustrates the possible (x) combinations of ModBus RTU functionality with physical connections.

	RS-232	RS-422	RS-485 4-wire	RS-485 2-wire	optical fibre
Point-to-point master or slave	x	x	x	x	x
Multidrop master	n/a	x	x	x	n/a
Multidrop slave	n/a	n/a	x	x	n/a

ModBus RTU physical signal compatibility

ModBus UDP and ModBus TCP are protocols designed for use with Ethernet TCP/IP networks.

ModBus UDP operates between a master (client) and slave (server) node(s) by sending UDP packets through the network. The master sends messages containing read or write functions to slaves and slaves respond by answering to the master. Because a slave sends its answer to the IP address of the master, multiple masters can access one slave thus making a multimaster configuration possible.

ModBus TCP relies on the master establishing TCP sockets between itself and all slaves in the network. The master can then send function messages to the slaves through these sockets and the slaves answer to the master accordingly. For any slave to serve multiple masters, more than one instance of the server must be running. While TCP has the advantage of being able to detect whether a socket is operable and is able to automatically resend data in case it is lost in transmission, it is more complicated to maintain than UDP. Thus in simple local communications, where the network operation is usually quite deterministic, UDP is often preferred when available in all devices.

9.1.1 MBOPEN()

Function	Start ModBus server
Syntax	*id*=MBOPEN(*device,addr,conf*)
id	The number of the started server (1...10) is returned in the variable *id*.
device	Serial port, UDP or TCP port. For example: “LP:9K68N20” ' serial port LP: “UDP:502://-2” ' UDP socket with two receive buffers “502://” ' TCP socket
addr	Node address
conf	0 Basic configuration: Addressing: 0 based (registers 0 .. 65535) 32 bit word order: lsw frst, msw last +1 Addressing: 1 based (registers 1 .. 65535) +2 32 bit word order: msw first, lsw last

The MBOPEN command is used for starting any type of ModBus server for operation as a slave node in a ModBus network. Variable *id* can be any variable accepting integer values and it must be declared before MBOPEN. It can later be used to monitor the Modbus slave status with MBREG or close the server with MBCLOSE.

The node address *addr* is used especially with Modbus RTU, where it defines the slave in a multidrop RS-485 network. With Ethernet (TCP and UDP), the node is already defined by the IP address and *addr* can usually be set to zero.

Parameter *conf* defines some details of the protocol and the values show are added for the desired combination.

For example, open a Modbus UDP server with 1 based addressing at port 502 of the controller.

```
DIM Mb
Mb=MBOPEN("UDP:502://-2",0,1) ' open ModBus UDP server
```

9.1.2 MBCLOSE

Command	Close all Modbus servers
Syntax	MBCLOSE[(*id*)]
id	Server number (1..10). If omitted, all Modbus servers are closed.

This command is used to end the operation of selected or all Modbus servers.

9.1.3 MBDATA()

Command	Set ModBus data
Syntax	MBDATA(*type,index*) = *value*
type	Data type 0 general addressing for all data types: 0..$3FFF word (16 bit) $4000..$7FFF long (32 bit) $8000..$BFFF float (32 bit floating point) $C000..$FFFF word (16 bit) 1 word registers (16-bit integer) 2 long registers (32-bit integer) 3 floating point registers (32-bit floating point) 4 coils (bit) 5 discrete inputs (bit)
index	0 .. 65535, address of the register/coil/input
value	Numerical value that fits the format of the register: 16 bit unsigned integer 0 .. 65535 16 bit signed integer: -32767 ... 32768 32 bit signed integer: -2147483647 .. 2147483648 32 bit floating point: Any real number (single precision) bit: 0 .. 1

Data for Modbus servers is organised in holding registers and coils/inputs. This data can be accessed from within the controller program or from the Modbus master connected to a server. The data is common for all servers. Some of the data areas (holding registers) are available for free program use, while others correspond to specific system resources. Especially binary I/O can be accessed as inputs/coils by the Modbus master using the coil/discrete input funtions. All binary and analog I/O together with a selection of servo axis related values can be accessed using the holding register commands. Data type 0 can be used for accessing all holding register data. Data types 1-3 can be used to access the different type user registers with alternate zero based addressing. Data types 4 and 5 can be used to access binary i/o INP() and OUT().

Function	Read Modbus data
Syntax	MBDATA(*type,index*)
type	As in MBDATA command
index	0..65535, adress of the register/input/coil
value	As in MBDATA command

The MBDATA function allows the application program to read Modbus register values. All values including those holding system and I/O information can be read.

The addressing of MBDATA can be done in two ways. Using *type* 0 allows access to all data within a single address (*index*) space (0 .. 65535 decimal or $0000 .. $FFFF hexadecimal).

Using *type*s 1 .. 3 allows access to user holding register areas according to data type. In this case the addressing (*index*) for each data type starts from zero. In case of 32bit data types (2 and 3) this type of addressing allows using contiguous addresses (0..511) for each pair of registers.

For example MBDATA(0,16384) is equal to MBDATA(2,0) and MBDATA(0,16386) to MBDATA(2,1) respectively.

The following table shows the allocation of address space for MBDATA:

MODBUS addressing for MBDATA

MBDATA(0,index), holding registers, global addressing			
index (decimal)	index (hex)	format	destination
0 .. 511	$0000 .. $01FF	16 bit integer	16-bit registers for free use. Also accessible as MBDATA(1,0..511)
3072+*a*	$0C00+*a*	16 bit signed integer	INPA(*a*) analog inputs, *a*= 0 .. 511
3584+*a*	$0E00+ *a*	16 bit signed integer	OUTA(*a*) analog outputs, *a*= 0 .. 511
4096+*a*	$1000+*a*	16 bit integer	INP(*i*), 16 inputs / register, *a*= 0 .. 2047, *b*= bit (0 .. 15), *i*=*a**16+*b*
6144+*a*	$1800+*a*	16 bit integer	OUT(*i*), 16 outputs / register, *a*=0 .. 2047, *b*=bit (0 .. 15), *i*=*a**16+*b*
8192 .. 8959	$2000 .. $22FF $2000+*a* $2080+*a* $2100+*a* $2180+*a* $2200+*a* $2280+*a*	16 bit 16 bit signed integer 16 bit signed integer 16 bit signed integer 16 bit signed integer 16 bit signed integer 16 bit signed integer	Axis(*a*) data (*a*=0 .. 127) POS(*a*) [encoder counts] (16 lsb) RPOS(*a*) [encoder counts] (16 lsb) FPOS(*a*) [encoder counts] (16 lsb) POSERR(*a*) [encoder counts] (16 lsb) MAXERR(*a*) [encoder counts] (16 lsb) OFFSET(*a*) [encoder counts] (16 lsb)
9216+*a*	$2400+*a*	16 bit integer, bit: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	Axis a status, meaning of bits: MOVE command in progress CREEP command in progress MOVEPROF command in progress HOME command in progress n/a FOLLOW command in progress OPWR command in progress disabled filter busy (reference filter not empty) n/a n/a MOVEBUFFER(*a*)>0 MOVEBUFFER(*a*)>1 MOVEBUFFER(*a*)>2 MOVEBUFFER(*a*)>3 motion in progress (bit 0-3 or bit 8 true)

9344+*a*	$2480+*a*	16 bit integer, bit: 0 1 2 3 4 5 6 7 8 9-14 15	Axis *a* error status disabled, PWR=0 NLIM, negative limit activated PLIM, positive limit activated EMRG, emergency stop MAXERR exceeded WAYERR, excessive McWay errors ENCERR, excessive encoder errors n/a tripped by TRIPGROUP member n/a error, some of bits 1-6 true
16384 .. 17407	$4000 .. $43FF	32 bit integer	32-bit registers for free use (even addresses). Also accesible as MBDATA(2,0..511)
24576 .. 28671	$6000 .. $6FFF $6000+*a*2* $6100+*a*2* $6200+*a*2* $6300+*a*2* $6400+*a*2* $6500+*a*2*	32 bit 32 bit signed integer 32 bit signed integer 32 bit signed integer 32 bit signed integer 32 bit signed integer 32 bit signed integer	Axis(*a*) data (*a*=0 .. 127) (even addresses) POS(*a*) [encoder counts] RPOS(*a*) [encoder counts] FPOS(*a*) [encoder counts] POSERR(*a*) [encoder counts] MAXERR(*a*) [encoder counts] OFFSET(*a*) [encoder counts]
32768 .. 33791	$8000 .. $83FF	32 bit floating point (IEEE 754)	Single precision floating point registers for free use (even addresses). Also accessible as MBDATA(3,0..511)
40960 .. 45055 40960+*a*2* 41216+*a*2*	$A000 ... $AFFF $A000+*a*2* $A100+a*2	32 bit floating point (IEEE 754)	Axis(*a*) data (*a*=0 .. 127) (even addresses) POS(*a*) [programming units] RPOS(*a*) [programming units]
49152 .. 65535	$C000 .. $FFFF	16 bit integer	16-bit registers for free use.

9.1.4 MBREG()

Command	Set Modbus status
Syntax	MBREG(*id*,*index*)=*value*
id	Server number (1..10)
index	20 configuration 0 registers 0..65535, 32-bit number lsw first, msw last +1 registers 1..65535 +2 32-bit number msw first, lsw last +4 32 bit address increment 2 21 node address 22 error counter 23 message counter 24 character counter

The MBREG command allows altering some ModBus server setting while it is running. Setting MBREG(*id*,20) alters the same settings as *conf* when starting the server with MBOPEN. It is also possible to change the node address the server recognizes, or preset some counters.

For example, setting MBREG(id,23)=0, would reset the message counter and make it easy to count the number of ModBus messages received by the server during some period.

Example:

```
' ModBus watchdog
DO ' while the master sends messaged at least every 1 second
  MBREG(1,23)=0
  DELAY 1
  UNTIL MBREG(1,23)=0
LOOP
'
' message timeout, stop machine
STOPMOVE          ' stop motion
FOR n=100 TO 131
  OUT(n)=0        ' reset outputs
NEXT n
```

Function	Read ModBus status
Syntax	MBREG(*id,index*)
id	Server number (1..10)
index	0..19 similar to STATUS(#*nn,i*) (see 8.5.5 STATUS, 8.7.3 STATUS) 20 configuration as in MBREG command 21 node address 22 error counter 23 message counter 24 character counter 25 last error: 1 = illegal function 2 = illegal data address (register, coil or input address) 3 = illegal value (data out of range) 4 = illegal format 26 last error fn: The ModBus function number causing the error. 27 last error addr: Node address in the command causing the error. 28 last error index: Last data address before the error. 29 last error data: Last data value before the error.

The MBREG function shows the communications status for a ModBus server, like the STATUS function would show for an opened port. It also allows reading the items that can be set using the MBREG command and provides some data on ModBus errors.

9.2 ETHERCAT

EtherCat fieldbus master functionality is available in ACN MPU3 for connecting to I/O and drives. McBasic commands and functions are available to setup the fieldbus configuration and to control it.

9.2.1 ETHERCAT

Use the ETHERCAT command to stop and start EtherCat.

Command	Control EtherCat master
Syntax	[*x*=]ETHERCAT(*n*,*m*[,*t*])
x	Optional return value. Use only when starting Ethercat. Number of EtherCat nodes found
n	Ethernet controller. 1 E1: (Ethernet port 1) 2 E2: (Ethernet port 2) 0 close EtherCat master
m	Mode: 0 stop 1 start
t	Optional timeout. Use only when starting EtherCat. Time to wait for nodes to respond. Normally program continues after all specified nodes have been found. After timeout program continues with uncomplete configuration (*x* less than number of specified nodes).

9.2.2 ECMOD$

Use the ECMOD$ command to set the configuration of the EtherCat fieldbus. Before using ECMOD$, use ETHERCAT(*n*,1) to set the fieldbus in configuration mode. After all settings have been made, use ETHERCAT(*n*,2) to start fieldbus operation.

Command	Set EtherCat configuration
Syntax	ECMOD$(*node,slice*)=*string*
node	Number of EtherCat node (0...511)
slice	Number of slice in modular I/O. 0 if fielbus coupler or drive (single device).
string	Configuration entry for *node*/*slice.* For valid devices, see appendix 1.

The list of valid devices is amended from time to time. Therefore it is maintained as an appendix to this manual (Appendix 1). A sample EtherCat configuration with some I/O and 2 servo drives:

```
ETHERCAT(2,0)                                ' stop EtherCat in port 2
ECMOD$(0,0)="NA-9286"                        ' Crevis I/O coupler
ECMOD$(0,1)="ST-1218 INP(1000)"              ' 8 input slice INP(1000..1007)
ECMOD$(0,2)="ST-2328 OUT(1000)"              ' 8 output slice OUT(1000..1007)
ECMOD$(0,3)="ST-2744 OUT(1008)"              ' 4 output slice OUT(1008..1011)
ECMOD$(1,0)="UNIDRIVE/M700 PWR(0) POS(0)" ' X axis servo
ECMOD$(2,0)="UNIDRIVE/M700 PWR(1) POS(1)" ' Y axis servo
ETHERCAT(2,1)                                ' start EtherCat in port 2
ECAX(0,-1,1000,1008,-1,-1)                   ' X axis I/O settings
ECAX(1,-1,1004,1009,-1,-1)                   ' Y axis I/O settings
```

Axis I/O setting are such that limit switch/emergency/status inputs are INP(1000..1003) for X axis and INP(1004..1007) for Y axis. OUT(1008) is configured as hardware enable for X axis and OUT(2009) for Y axis.

It is often good programming practice to stop EtherCat before configuring and starting it. This allows restarting EtherCat always when restarting the program when testing, for example. Otherwise EtherCat is only stopped when exiting McBasic or reseting/ power cycling the system.

9.2.3 ECPAR

Read and write EtherCat parameters. These parameters are normally set by ECMOD$ when configuring Ethercat. For special debugging/setup purposes ECPAR provides a way to access them. Writeable values should only be written with EtherCat configuration mode, ETHERCAT(*n,*1).

Command	Write EtherCat parameter
Syntax	ECPAR(*node,par*)=*expression*
node	Number of EtherCat node (0...47)
par	Parameter number (0...73). Parameters currently in use: par function 0 device address 0..65535 1 vendor id (32-bit number) 2 product code (32-bit number) 3 product revision (32-bit number) 4 serial number (32-bit number) 5 axis number, default -1=no, 0=X 1=Y .. 6 output message box base address 0.. 7 input message box base address 0.. 8 output message box size (bytes) 0.. 9 input message box size (bytes) 0.. 10 output process data offset, slice 0 11 input process data offset, slice 0 12 output process data offset, slice 1 13 input process data offset, slice 1 . 10+2*n output process data offset, slice n 10+2*n+1 input process data offset, slice n . 72 output process data offset, slice 31 73 input process data offset, slice 31

Function	Read EtherCat parameter
Syntax	ECPAR(*node,par*)
node	Number of EtherCat node (0...511)
par	0..72 as in ECPAR command.

9.2.4 ECAX

Command	Configure EtherCat axis I/O
Syntax	ECAX(*axis,aout,inp,ena,nrun,prun*)
axis	Number of axis
aout	Address of analog output used as reference output for *axis*
inp	Adress of first input of the 4 inputs block for *axis* INP(*inp*) drive status as defined by LIMITTYPE(*axis*) INP(*inp*+1) negative limit switch as defined by LIMITTYPE(*axis*) INP(*inp*+2) positive limit switch as defined by LIMITTYPE(*axis*) INP(*inp*+3) emergency swich (1=ok)
ena	Drive enable output
nrun	When DRIVETYPE(*axis*)=2, run negative signal
prun	When DRIVETYPE(*axis*)=2, run positive signal

When using ECAX, the EtherCat must be in operating mode, ETHERCAT(*n,*2).

If any of the *aout,inp,ena,nrun,prun* are not used, they can be given the value -1.

9.2.5 ECCO

Read and write device registers using CAN-over-EtherCat protocol. ECCO allows single read and write operations to device registers to access device data and settings such as drive parameters etc.

Command	Write Ethercat CoE register
Syntax	ECCO(*nodeaddr*,*index*,*subindex*,*bytes*)=*integer*
nodeaddr	Address of EtherCat node
index	EtherCat register index as defined for the device in question.
subindex	EtherCat register subindex as defined for the device in question.
bytes	Length of *integer* for the register [bytes].

Function	Read Ethercat CoE register
Syntax	ECCO(*nodeaddr*,*index*,*subindex*)
nodeaddr	Address of EtherCat node
index	EtherCat register index as defined for the device in question.
subindex	EtherCat register subindex as defined for the device in question.
value	Value in the registed, integer.

When using ECCO, the EtherCat must be in operating mode, ETHERCAT(*n*,2).

For example, write parameter 3.10 (Speed controller P gain, 16 bit value) in Unidrive SP drive as node 5 on the EtherCat fieldbus:

```
ECCO(5,$2000+3,10,2)=100
```

Parameter addresses in the drive map to the EtherCat register addresses so that the *index* will be $2000+*menunumber* and the *subindex* will be equal to the parameter number. The length of the parameter is 16 bits, so *bytes* is 2. Note that it is necessary to define *bytes* only when writing the parameter. The actual value of the gain will be 0.0100 as it is defined in the drive with 4 decimal resolution.

For example, read parameter 3.10:

```
PRINT ECCO(5,$2000+3,10)

100
```

9.2.6 ECSERNUM

Specify EtherCat device serial number for verification of configuration.

Command	Specify EtherCat device serial number
Syntax	ECSERNUM(*node*)=*integer*
node	Address of EtherCat node
integer	32 bit integer

After specifying the serial number of an EtherCat device the system only accepts a device of the type specified with ECMOD$ and with the serial number specified with ECSERNUM to be connected as device number *node.*

ECSERNUM can also be read with the corresponding function to check whether it has been set.

Function	Read EtherCat serial number specification.
Syntax	ECSERNUM(*node*)
node	Number of EtherCat node
value	Current EtherCat serial number specification for the device, 32 bit integer.

The EtherCat serial number specification can be used to identify the device and to prevent unwanted configuration. This may be particularly handy in large installations with redundant fieldbus topologies are used and device fielbus wiring may be rerouted in case of malfunction.

While some EtherCat devices have their serial numbers set by the manufacturer, some allow writing to the serial number register and may in fact have a zero serial number by default.

To read and write the serial number of an EtherCat device from McBasic, use the ECPAR(*node*,4) command and function (see 9.2.3 ECPAR).

9.3 FIELDBUS SLAVE OPTION

A fielbus slave option is available for the ACN MPU3 controller for connection to further popular fieldbuses, such as Profinet and Profibus, as a slave node. The option uses an Anybus module for this connection. Different fieldbuses use the same commands and functions for configuring and using the option.

9.3.1 ANYBUS

The ANYBUS command controls the Anybus fieldbus slave option operation.

Command	Control Anybus module operation
Syntax	ANYBUS=*integer*
integer	0 set init/configuration mode 1 start fieldbus operation

To stop the fieldbus and enter configuration mode set ANYBUS=0. To start fieldbus operation set ANYBUS=1.

9.3.2 ABCONF$

ABCONF$ command is used to configure fieldbus process data objects. Setting object properties defines the data frame for the fieldbus and creates a fieldbus data buffer for the ACN controller. Various object types are available for the data frame. It is important to configure the data frame to match the fieldbus master settings. Depending on the master and fieldbus type, only some of the object types may be applicable.

Most of the data types need to be connected to input or output registers (INPREG, OUTREG) from where they can be accessed by the McBasic program. Bit type objects can also be connected to i/o bit registers (INP,OUT) so that one of more bits long data can be directed to consequtive i/o addresses.

When using INP/OUT i/o registers, take care not to cause address conflicts with other i/o connected to the system.

Function	Read Anybus module object configuration.
Syntax	ABCONF$(*a*)
a	1..65535 Number of adi (application data instance) -1 Default device type description (read only) -11 User configurable device type description (r/w)
Value	String *a*>0: Current object configuration for adi. *a*<0: Current value of description *a*.

Command	Set Anybus module object configuration
Syntax	ABCONF$(*a*)=*string*
a	*a*>0: Number of adi (application data instance) *a*=-11: Set user configurable device type description
string	*a*>0: process data configuration string elements, "*objtype*[**count*] io(*addr*)", where *objtype*: object type BOOL boolean 0/1 BITS8 8 bits BITS16 16 bits BITS32 32 bits CHAR 8-bit character FLOAT 32-bit ieee floating point number SINT8 signed 8-bit integer -128..127 SINT16 signed 16-bit integer -32768..32767 SINT32 signed 32-bit integer -2147483648..2147483647 SINT64 (signed 64-bit integer), precise upto 52-bits OCTET 8-bit data UINT8 unsigned 8-bit integer 0..255 UINT16 unsigned 16-bit integer 0..65535 UINT32 unsigned 32-bit integer 0..4294967295 UINT64 (unsigned 64-bit integer 0..), precise up to 52-bits
count	Number of similar fields, if more than 1
io(*addr*)	first McBasic reference (*addr* is address of first i/o or register) INP*(addr*) bit input OUT(*addr*) bit output INPREG(*addr*) register input OUTREG(*addr*) register output
string	a=-11 User configurable device type description string

ABCONF

ABCONF is used to access the Anybus fieldbus module numeric parameters.

Function	Read Anybus module parameters.
Syntax	ABCONF(*n,d*)
n	-1 module status data -2 user settings
n=-1	*d* (read only values) -2 physical network status (binary) bit 0 link sensed bit 1 ip address ok bit 3 port1 link sensed bit 4 port2 link sensed -1 anybus status (integer) 0 Setup 1 NW_Init 2 Wait process 3 Idle 4 Process active 5 Error 7 Exception 0 device type (Anybus module type specific) 5 Profibus 135 Ethercat 137 Profibus IRT 143 Modbus TCP 150 Profinet IO 2-Port 1 manufacturer's vendor id ($10C, HMS Industrial Networks) 2 firmware version 3 ip address 4 subnet mask 5 gateway address 6 cycle time
n=-2	d (user values, R/W) 0 device type 1 vendor id
Value	Current object configuration for n,d. Integer.

Command	Set Anybus module parameter
Syntax	ABCONF(*-2,d*)=*expression*
d	0 device id 1 vendor id
expression	New value for parameter.

For example:

```
ANYBUS=0                                ' init

ABCONF(-2,0)=ABCONF(-1,0)+$100          ' set device id
ABCONF(-2,1)=$4D43                      ' set vendor id
ABCONF$(-11)="ACN MPU3 with Anybus" ' device description

' process data in
ABCONF$(1)="UINT8*2 INPREG(100)"    ' 2 8bit unsigned integers
ABCONF$(2)="SINT16  INPREG(102)"    ' 1 16bit signed integer
ABCONF$(3)="UINT16  INPREG(103)"    ' 1 16bit unsigned integer
ABCONF$(4)="BOOL    INPREG(104)"    ' 1 boolean (bit)

' process data out
ABCONF$(5)="SINT16  OUTREG(100)"    ' 1 16bit signed integer
ABCONF$(6)="UINT32  OUTREG(101)"    ' 1 32bit unsigned integer
ABCONF$(7)="SINT16  OUTREG(102)"    ' 1 16bit signed integer

ANYBUS=1                                ' start fieldbus master
```

Fieldbus masters such as Profibus and Profinet master usually need a configuration file called the GSD file for configuring the master for the desired process data configuration. These files are available from SKS Control. Each GSD file contains a specific process data configuration. For correct operation, the ACN Anybus must be configured to match the configuration described in the GSD file.

10. TIMING AND REAL TIME CLOCK

Modified files are dated automatically according to the current time of the MPU3 real time clock, when closed. 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).

10.1 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

```
DATE$(#1)="110224123456"
```

sets the date to 24.02.2011 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 8.3.12 - 8.3.13.

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

```
110919161354
```

```
D$=DATE$(#1)
PRINT MID$(D$,5,2)+"."+MID$(D$,3,2)+".20";
PRINT MID$(D$,1,2)+"  at ";
PRINT MID$(D$,7,2)+":"+MID$(D$,9,2);
PRINT " o'clock"
```

```
19.09.2011  at 16:13 o'clock
```

10.2 TIME MEASUMENTS

10.2.1 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.

```
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 Delay1:

```
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
```

10.2.2 CLOCK

Function	Read system on -time
Syntax	CLOCK
Type	Real number
Value	Time elapsed from power-on [s].

CLOCK function provides means to read system uptime in [s].

10.2.3 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 passes 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
```

11. OTHER COMMANDS

11.1 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.

11.1.1 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.

```
DataBlock1
 DATA 10,13,15,"SKS CONTROL","MANUFACTURER"
 DATA 3,2,7,"AUTOMATIC MACHINE LTD","CLIENT"
```

11.1.2 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
```

11.1.3 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 the 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
```

11.1.4 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.

11.2 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.

11.2.1 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.

Notice that the DEF command has to be the first command on a program line.

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

```
68.00
```

11.2.2 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
```

11.3 COMMENTS

11.3.1 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
```

11.3.2 '

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
```

12. 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=, JCOMP= 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, stiffness, 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 the program even if some of the default values could be used in the application. This makes the program easier to read and modify.

12.1 ENCODER OPERATION

12.1.1 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
```

12.1.2 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.

12.1.3 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.

12.1.4 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 when connected to modules such as AXi, AXa, WAX2 and WAX2A .

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

In conjunction with absolute encoders ENCERR reports errors occurred for example in SSI transmission. 3 successive errors automatically result in a position loop error (MOVEREADY..=-64) and drive disable.

12.2 POSITION CONTROL SETTINGS

12.2.1 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: 1 ±10V reference (WAX[2][A], or AXi/AXa module) with enable at ENA2 only 2 0...10V reference with direction output at ENA1 3 as 1 but enable both at ENA1 and ENA2 Additional setting can be added to *type* as bits set in DRIVEYPE. Thus each of the following addition (powers of 2) applies the described setting to *axis*: add +8 disable automatic PWR on for MOVx commands +16 disable limit switches, MAXERR and EMRG intervention +32 disable encoder counter +64 disable motion commands (MOVE,MOVER,MOVC,CREEP etc.) +128 disable position controller +256 do not stop logging at PWR<1 +1024 invert REF output

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

```
DRIVETYPEXY=1
```

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

```
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.

12.2.2 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 (add values marked with + to 0,1,2 or 3 as applicable). *type* function 0 or 1 no limit switches 2 NLIM and PLIM, normal closed 3 LIM (n.c.) and MASK (open in negative end) +4 invert limit switch signals +8 use index pulse (CLKX, edge-activated) +16 STAT normal open +32 invert STAT 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 STAT external trip switch STAT

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.

12.2.3 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 500 in standard McBasic versions for MPU3. 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.

PIDFREQ also sets the refresh rate of real time I/O systems, such as McWay and Ethercat.

12.2.4 RAMPTIME

RAMPTIME can be used to control active braking of a servo axis.

Command	Set stop ramp time for axis.
Syntax	RAMPTIME[{*axes*...|(*n*,...,*m*)}]=*expression*
{*axes*...|(*n*...)}	List of axes (or (*n*...) - axes numbers), whose ramp time is set.
expression	Time [s] for REF to go from 100% to zero (0...60). Default 0.

Function	Read stop ramp time of axis.
Syntax	RAMPTIME{*axis*|(*n*)}
{*axis*|(*n*)}	Identification letter or number of *axis*.
Value	Current stop ramp time [s] of *axis.*

By default, the REF output (usually speed reference), goes to 0 when the axis is disabled by setting PWR=0 or by emergency/limit switch or an error condition. Using the RAMPTIME setting, the behaviour of the REF output can be adjusted to allow a suitable limited deceleration if the drive is kept enabled during the stop ramp. If ENA outputs are used they will go off in the beginning of the ramp to reflect that stop sequence has started and can be used to activate a delayed disable/emergency power off etc.

During the stop ramp MOVEREADY will reflect the stopping status with a negative value according to the stopping reason (see chapter 12.6) deducted with a further -512.

When using EtherCat connected drives, the drive will be disabled by the control word after the ramp has finished.

12.2.5 BRAKETIME

BRAKETIME can used to insert a futher delay after stop ramp before disabling an EtherCat drive. This may be useful to allow for a brake system to activate before torque is removed from the motor.

Command	Set brake delay time for axis.
Syntax	BRAKETIME[{*axes*...|(*n*,...,*m*)}]=*expression*
{*axes*...|(*n*...)}	List of axes (or (*n*...) - axes numbers), whose brake delay time is set.
expression	Time [s] from REF reaching zero to disable (0...1). Default 0.

Function	Read brake delay time of axis.
Syntax	BRAKETIME{*axis*|(*n*)}
{*axis*|(*n*)}	Identification letter or number of *axis*.
Value	Current brake delay time [s] of *axis.*

12.2.6 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. Real number. 0 prevents operation of the feedback system.

Function	Read proportional gain of position control.
Syntax	GAIN{*axis*|(*n*)}
{*axis*|(*n*)}	Identification letter or number of axis.
Type	Real number.
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=40
GAIN(2,12,14)=0.3
DIGITS=3
PRINT GAINX,GAINY,GAINZ
PRINT GAIN(12),GAIN(14)


40       40       40
0.300    0.300
```

McBasic 3.3 uses a floating point counter for position error in position control. Therefore the maximum position error is equal to the maximum operating area of the position measurement system. However, the proportional area, when GAIN is set to 1, 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).

12.2.7 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. Real number. 0 prevents integration.

Function	Read integrating factor of position control.
Syntax	INTG{*axis*|(*n*)}
{*axis*|(*n*)}	Identification letter or number of axis.
Type	Real number
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 using INTG. Because the integrator increases the order of the control system, it also increases the tendency to oscillation. Therefore, the use of INTG often 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=3.5
INTGZ=0

PRINT INTG(1),INTGY,INTGZ

4        3.5      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.

12.2.8 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. Real number. 0 prevents operation of derivation.

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

DERV defines the part of reference caused by quick change in position error. 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.

DERV is mostly needed in conjunction with control setups with direct torque control or low gain velocity loop. Most servo systems with a high gain velocity loop in the servo drive do not need DERV.

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. In systems with low resolution position measurement, using DERV can be limited by quantization noise.

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)=0.6
PRINT GAINX,INTGX,DERVX
PRINT GAIN(3), INTG(3), DERV(3)

10        0         7
10        0         0.6
```

12.2.9 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.

12.2.10 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.

Setting ACOMP to zero removes the operation of acceleration compensation.

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

12.2.11 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.

12.2.12 JCOMP

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

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

Jerk feedforward or jerk compensation for position control. Value of expression represents the time in [s] it takes for the acceleration (torque/force) to settle the set value. Usual values with industrial brushless servo are in the range of 0.001 to 0.02 (1-20ms). Generally, values less than 1/PIDFREQ have no effect on the control.

JCOMP parameter can be used to add to the control (reference) output a part depending on the latency of the change of current of the servo drive. It allows the reference signal to compensate for the latency.

JCOMP setting is called critical, when the controlled axis follows the generated motion with minimum position error behaviour during positive and negative changes in acceleration.

JCOMP value 0 removes the operation of jerk compensation.

12.2.13 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 (-255..255). 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 filter operation.

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 20ms period for a system with PIDFREQ=500 (1s / 500 = 2ms). 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 20ms filter at 1000mm/s would cause a 10mm 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).

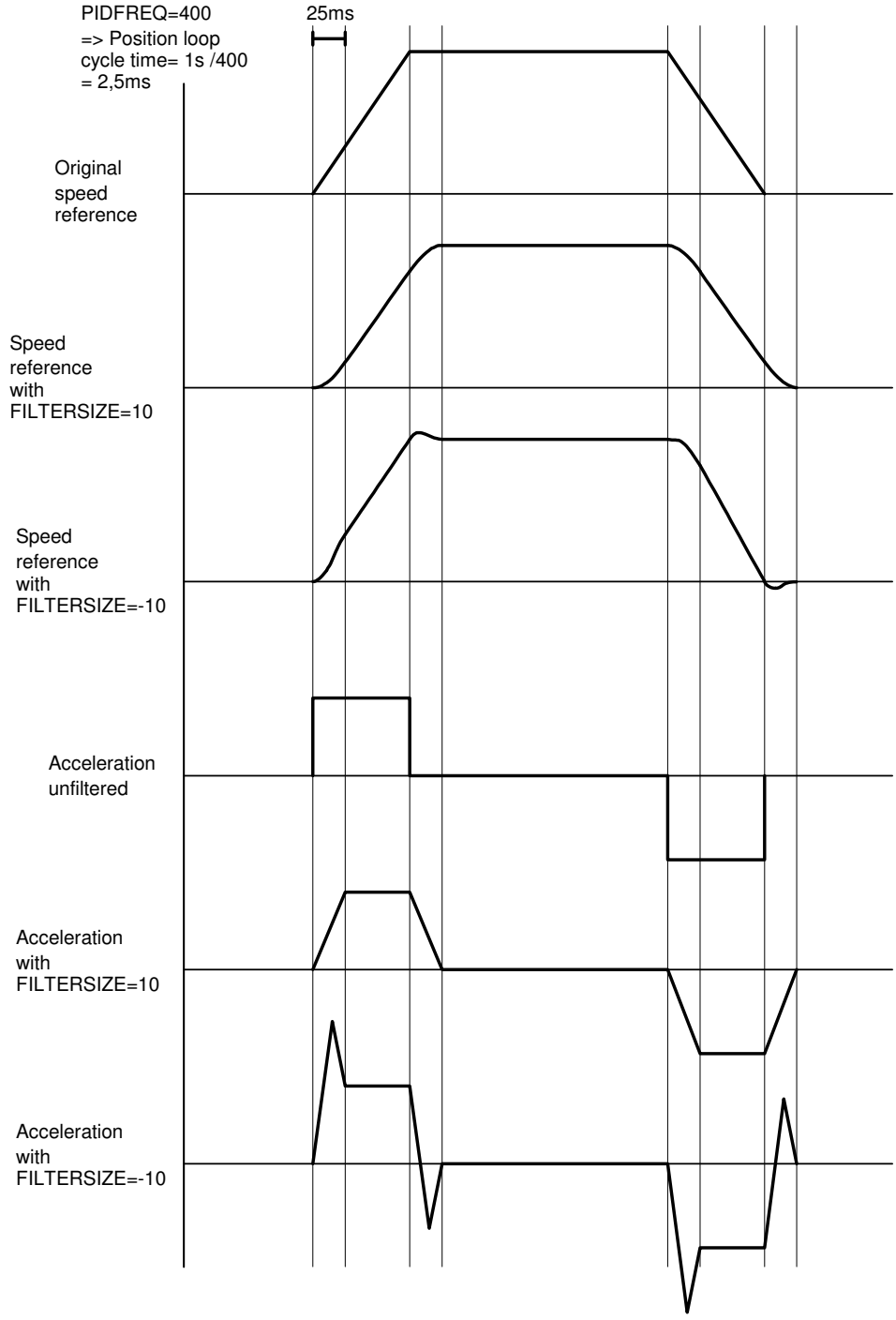


Fig. 11.2.10, The effect of FILTERSIZE

12.2.14 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

$$v_{xyz} = \sqrt{v_x^2 + v_y^2 + v_z^2}$$

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.

12.2.15 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.

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

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 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.

12.2.16 OVERRIDE

Command	Scale speed of motion generated with MOV.. commands.
Syntax	OVERRIDE[{*axes*...|(*n*,...,*m*)}]=*expression*
{*axes*...|(*n*...)}	List of axes (or (*n*...) - axes numbers), whose speed is adjusted.
expression	Scale factor for speed, 0..10 (default = 1).

Function	Read speed scale setting of axis.
Syntax	OVERRIDE{*axis*|(*n*)}
{*axis*|(*n*)}	Identification letter or number of *axis*.
Type	Real number.
Value	Current OVERRIDE factor of *axis*. During changes in OVERRIDE, where OVERRIDERATE limits the rate of change of the scale factor, gives the actual momentary value.

OVERRIDE is a method for scaling programmed paths and motion generated with MOVE, MOVC, MOVER, MOVCR, CIRCLEMOVER and CIRCLEMOVCR commands. It also scales the accererations of the motion to preserve the programmed path and shape of the motion. OVERRIDE can be adjusted from complete standstill to max. 10x speed. Value 1 represents the original programmed speed.

12.2.17 OVERRIDERATE

Command	Set rate of change for OVERRIDE.
Syntax	OVERRIDERATE[{*axes*...|(*n*,...,*m*)}]=*expression*
{*axes*...|(*n*...)}	List of axes (or (*n*...) - axes numbers), whose OVERRIDERATE are set.
expression	Value for rate of change [1/s].

Function	Read rate of change for OVERRIDE.
Syntax	OVERRIDERATE{*axis*|(*n*)}
{*axis*|(*n*)}	Identification letter or number of axis.
Type	Real number.
Value	Current OVERRIDERATE.

With OVERRIDERATE the rate change of OVERRIDE can be limited to allow suitable time for the change so as not to exceed the force available for the extra acceleration for the change. The value represents the rate of change in [1/s], so a value for 1, for example allows the speed factor to

change from its initial value to a value of 1 lower of higher in 1s. With setting OVERRIDERATE to 0.1 the same change would take 10s.

12.2.18 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.

12.3 POSITION CONTROL FUNCTIONS

12.3.1 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 3.3 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..

12.3.2 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

12.3.3 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.

12.3.4 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
```

12.3.5 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.

12.3.6 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

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

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

12.4 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.

```
' RUNNING HOME
HOMEXYZ
IF NOT MOVEREADYXYZ THEN 190
RETURN
```

12.5 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.

```
ACCELXZ=900 : STOPMOVEXZ

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.

12.6 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)
	-128	external trip from WAX stat input
	-256	tripped by another axis in a TRIPGROUP

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.

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

12.6.1 TRIPGROUP

Set group of axes to trip together trigged by any of the group members. TRIPGROUP applies to servo errors generated for axes as described in MOVEREADY (negative values). Each group is distinguished by its number, so several groups can exist simultaneously. It is possible to add and remove axes to and from groups using the TRIPGROUP= command.

Command	Set trip group of axes.
Syntax	TRIPGROUP[{axes…|(n,…,m)}]= *expression*
{*axes*...|(*n*...)}	List of axes (or (*n*...) - axes numbers), whose trip group are set.
expression	Group number (0 .. 255). If set to 0, the axes are no longer a member of a trip group.

Function	Read trip group of axis.
Syntax	TRIPGROUP{*axis*|(*n*)}
{*axis*|(*n*)}	Identification letter or number of axis.
Type	Integer number.
Value	Axis trip group number. Default 0 (no group).

The axis that caused the trip has its MOVEREADY value set as described before. Axes that have tripped because of another axis in the group, have their MOVEREADY values set at the MOVEREADY value of the axis that has caused the trip -256. Thus, for example, if X and Y axes belong to the same TRIPGROUP and X axis exceeds its MAXERR, both X and Y axes will trip simultaneously and MOVEREADYX will be -16 and MOVEREADYY -272.

12.7 TRANSLATIONS

12.7.1 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.

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

12.7.2 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.

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

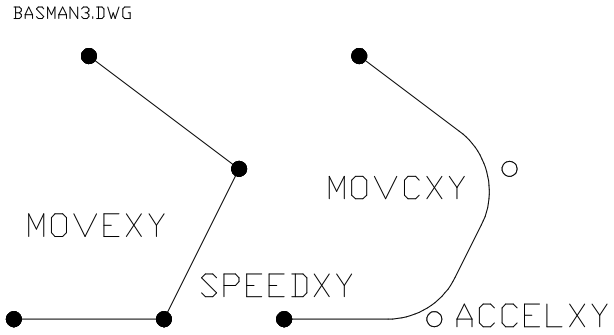


Fig. 11.7.2, two axis translations

12.7.3 CIRCLEMOVER

The CIRCLEMOVER command allows generating circular motion with two axes forming a cartesian two axes frame of reference, like the XY plane. Optionally, one or more additional axes can be included to be linearly interpolated together with the circular motion. The motion obeys speeds, accelerations and decelerations set for the axes so that for the circular motion the axis with the lower parameter sets the limits that are then combined with the limits of the linear interpolation similar to other linear motion commands.

Command	Relative circular move.
Syntax	CIRCLEMOVER(*aa,axis1:xx,axis2:yy*[,*nz:zz,.. nw:ww*])
aa	Angle to move [radians].
axis1	Number of first axis
xx	Center point offset in the direction of *axis1*
axis2	Number of second axis
yy	Center point offset in the direction of *axis2*
nz	Number of first optional axis to perform linear interpolation.
zz	Relative move length.
.	.
.	.
nw	Number of last optional axis to perform linear interpolation.
ww	Relative move length

The circular motion starts from the current position of axes *axis1* and *axis2* forming the cartesian frame of reference, such as the XY plane. The centerpoint of the arc to be performed is defined by *xx* and *yy* as offsets from the starting position in the directions of *axis1* and *axis2* respectively. The length of the circular motion is defined as an angle *aa* in radians (π radians equals 180 degrees). Thus, for example a full circle is about 6.28 radians. The sign of *aa* defines the direction of the motion. A positive value of *aa* moves the vector position of (*axis1, axis2*) in the positive angular direction (ccw) and a negative value of *aa* in the negative angular direction (cw) respectively.

Examples:

```
CIRCLEMOVER(2*PI,0:5.0,1:5.0)    'full circle (ccw)
CIRCLEMOVER(-PI,0:0,1:5.0,3:6)   'half circle (cw) with W-ax
                                 'linear 6 unit move
```

12.7.4 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.

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

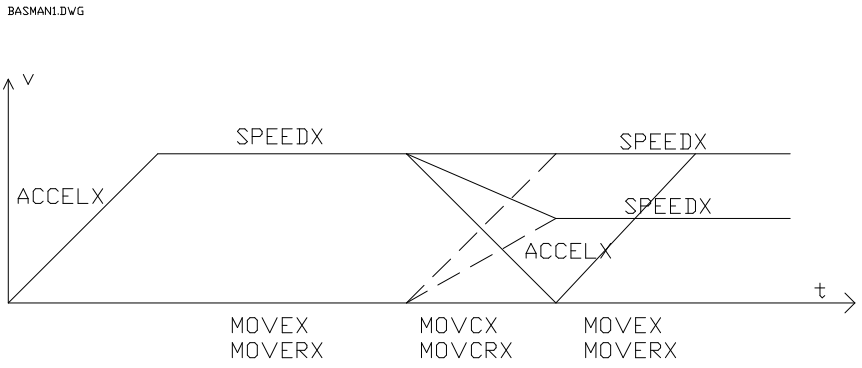


Fig. 11.7.3, Speed profiles

12.7.5 CIRCLEMOVCR

The CIRCLEMOVCR operates the same way as the CIRCLEMOVER command, with the difference that it can start before the previous motion with the same axes has stopped. The joining of

consecutive motion commands operates in the same way as with other continuous motion commands like MOVC etc., additionally considering the angular ramps that are generated during the circular motion.

Command	Relative continuous circular move.
Syntax	CIRCLEMOVCR(*aa,nx:xx,ny:yy*[*,nz:zz,.. nw:ww*])
aa	Angle to move [radians].
axis1	Number of first axis
xx	Center point offset in the direction of *axis1*
axis2	Number of second axis
yy	Center point offset in the direction of *axis2*
nz	Number of first optional axis to perform linear interpolation.
zz	Relative move length.
.	.
.	.
nw	Number of last optional axis to perform linear interpolation.
ww	Relative move length

12.7.6 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.

Example: Move along a polygon approximation of a circle:

```
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*PI STEP PI/8
DO : UNTIL MOVEREADY(1,2)<4
 IF MOVEREADY(1,2)<0 THEN STOP
LOOP
MOVC(1:Xorig+Radius*COS(Ang),2:Yorig+Radius*SIN(Ang))
NEXT Ang
```

12.8 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)
ACCELX=100    'see Fig. 11.8
CREEPX100
DELAY 3
ACCELX=150
CREEPX70
DELAY 1
ACCELX=50
CREEPX30
DELAY .5
ACCELX=100
DELAY .75
ACCELX=75
CREEPX100
DELAY.2
CREEPX0
```

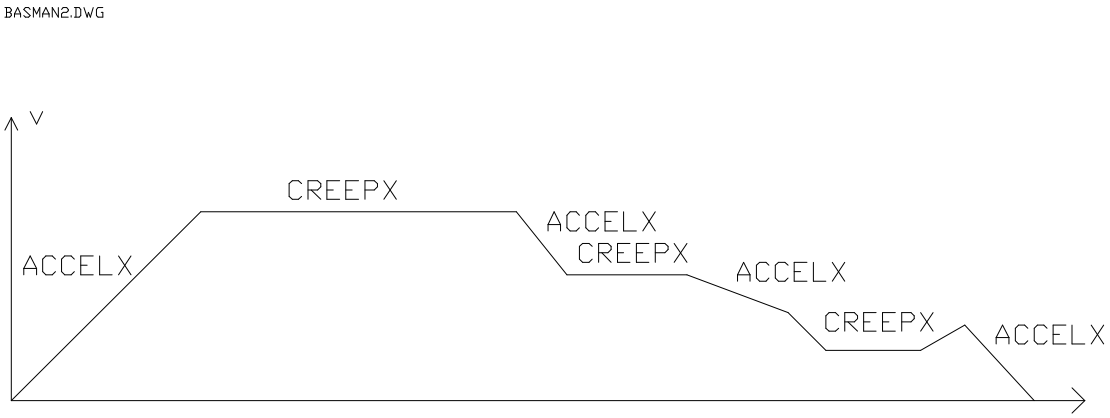


Fig. 11.8, motion with CREEP -command

12.9 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 16 bit integer and 32 bit decimal part, the following limitations are valid regarding its operation:

Maximum value of ratio i is $\pm 2^{15}$ counts/count with accuracy of $1/2^{32}$ counts.
To follow continuous reference motion such as that of a of a master encoder without inaccuracy caused by rounding error, the reference (or master) encoder must be selected to have a pulse number so that the cycle of the slave axis can be defined as an exact real number (less than 14 significant digits) or a rational number of counts from the reference encoder.

12.10 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.

12.11 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.

```
PWR=0
FOR N=0 TO 1 STEP .1
PWRX=N
TIMER(0)=.1
IF TIMER(0) THEN 240
NEXT N
```

12.12 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
```

12.13 FAST POSITION CAPTURE

12.13.1 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 AXi or 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 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 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
```

12.13.2 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 AXi or 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 using the index channel is approximately ±1 encoder pulse edge, and . ±0.1 millisecond using inp0.

12.14 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.

12.14.1 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.

12.14.2 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)

```
FOR N=0 TO 511
PROFX(N)=SIN(2*PII*N/512)
PROFY(N)=COS(2*PII*N/512)
NEXT N
PROFX(512)=PROFX(0)
PROFY(512)=PROFY(0)   'Note! PROF as function
```

12.14.3 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:

```
REST=512 : ACCELT=2
PWRXYT=1
MOVEPROFXY(T)
CREEPT(2)
```

12.15 POSITION CONTROL LOG

12.15.1 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.

12.15.2 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. Starting logging automatically clears all log data in the log array.

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
```

No data will be logged if PWR(*axnr)* <= 0. Therefore logging stops also if a servo error occurs for an axis or the axis is disabled. This can be prevented by adding 256 to the DRIVETYPE of the axis (see chapter 12.2.1).

12.15.3 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
-1	PID count	32bit integer
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()

additionally with some axes using EtherCat connected drives such as Unidrive M, some of the following data may be available:

8:	ethercat status word (6041.0)
9	ethercat control word (6040.0)
10	motor current (6077.0) (M700 series)
11	drive bus voltage Vdc (2005.5) (M700 series)
12	drive i/o (2008.20) (M700 series)
13	actual velocity (6043.0) (M300)
14	target velocity (6042.0) (M300)

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.

13. I/O CONNECTIONS

The digital and analog inputs and outputs in ACN control systems are available for programming with dedicated McBasic commands and functions.

Inputs and outputs exist in the control system in various devices connected to the system either in the ACN chassis, through external McWay I/O system or EtherCat fieldbus.

13.1 McWay I/O configuration

McWay is the I/O connection system used for ACN I/O modules installed in the ACN chassis or other McWay I/O modules connected to ACN external McWay loops.

Before a McWay loop can be used, it must be initialised using the McBasic WAYMOD$ command.

13.1.1 WAYMOD$

Command	McWay I/O system configuration.
Syntax	WAYMOD$(*n,m*)=*string*
n	Loop number (0...3). Loop 0 is the ACN chassis internal McWay loop.
m	Module number (0...120). Module 0 is the first module in the loop.
string	specification and information of module: “END” end of modules in loop “EMPTY(*n*)” no module in current position, reserve n bits of I/O space in the loop "AXi INP(*n1*) [OUT(*n2*)] IO(*n3*)" AXi axis connection module (incr.enc., 32bits) "AXa INP(*n1*) [OUT(*n2*)] IO(*n3*)" AXa axis connection module (abs.enc., 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 "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) "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 on module specific syntaxes refer to chapter 6.8 of "ACN Motion Control System User's Manual" or chapter 3 of "McWay I/O - system user’s manual".

13.1.2 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
```

13.1.3 WAYSLAVE

McWay I/O system provides functionality to build hierarchical motion controller systems using several ACN MPU3 controllers. In such systems, a master controller can distribute position and status information through McWay loops to slave controllers connected in the loop. To set a loop to function as a slave, use the WAYSLAVE command.

Command	Set McWay I/O slave mode.
Syntax	WAYSLAVE=*n*
n	Number of McWay connection to be used as slave. Set to -1 to exit slave mode.

Function	Read slave mode status.
Syntax	WAYSLAVE
Type	Integer -1 ... 3
Value	-1 slave mode off 0 .. 3 number of loop in slave mode

To use slave mode for axes, data must be configured to the master loop and slaves as virtual i/o modules. 56bit data objects are available for this. The objects transfer 32bit position information and 16bit status information for axis master/slave operation. To configure, use the WAYMOD$ command to set master loop:

Command	Set McWay axis output virtual module.
Syntax	WAYMOD$(*n*,*m*)="WMC POS(*a*)"
n	Number of master McWay loop.
m	Position of slave in the loop.
a	Number of master axis.

Command	Set McWay axis input virtual module.
Syntax	WAYMOD$(*n*,*m*)="WMCIN POS(*a*)"
n	Number of slave McWay loop.
m	Position of master virtual module in the loop.
a	Number of slave axis.

In a master controller, one or more WMC modules can be configured in any of the available McWay loops. Slave controller(s) must then be connected in the loop(s) at the correct location with WAYSLAVE set to the McWay connection used and WMCIN modules configured at the corresponding locations. It is also possible to configure WMC modules in the slave and WMCIN modules in the master to read position data from the slave to the master.

When set, the position(s) are transferred in real time and can be used as references for axes motion. The axes that are used are typically configured as virtual axes to be able to build axes groups to be moved using FOLLOW or MOVEPROF, for example. Axis status data is also copied between master and slave axes so that if one detects an error condition, the other one will also trip. TRIPGROUP can be used to further distribute error reaction to cover axes in several controllers.

13.1.4 MOTION CONTROL I/O LOGICAL ADDRESSES

When connecting axis I/O to the system, motion control related inputs and outputs 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

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 upto 31 or 99 depending on the McBasic version used. The first I/O address for each axis is its number multiplied by 4. This address is also be used in conjunction with axis module settings (WAYMOD$, ECMOD$) 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.

13.1.5 I/O LOGICAL ADDRESSES

Also other I/O devices connected to the ACN system are configured using the WAYMOD$ function for ACN McWay modules and ECMOD$ for Ethercat connected modules. Please refer to the ACN User's manual and McWay User's manual for more information on McWay configuration and chapter 9.2 in this manual for information on EtherCat configuration.

13.2 DIGITAL I/O

13.2.1 INP

Function	Read status of input.
Syntax	INP(*a*[,*n*])
Type	Truth value.
a	Input address. A numerical expression (integer).
n	Number of inputs to read (-32...32). Default 1. When *n* is positive, INP(a) is the LSB. When *n* is negative, INP(a) is the MSB.
Values if	0 inputs not active (off) 1 input INP(*a*) active (on) (*n*=1) x when n>1, x= INP(*a*)+...+2^{n-1}*INP(*a+n-1*) when n<-1, x= 2^{-n-1}*INP(*a*)+...+INP(*a-n-1*) -1 Communications error -2 Missing module

INP function is used for reading the status of a binary input or *n* inputs.

```
DO UNTIL INP(3)=0 AND INP(4)=1 : LOOP

PRINT INP(100),INP(101),INP(102),INP(103)
PRINT INP(100,4)
PRINT INP(100,-4)

1       0       1       0
5
10
```

13.2.2 OUT

Command	Control an output.
Syntax	OUT(*a*[,*n*])=*expr*
a	Address of output.
n	Number of consequtive outputs to set. Default 1.
expr	Value to set. 1 or 0 (ON or OFF). 0 output(s) off 1 output(s) on x when *n*>1, OUT(*n*)= LSB of *expr* , OUT(*n*+1)=the next bit etc. when *n*<-1,OUT(*n*)=bit *n*-1 of expr, OUT(*n*+1)=bit *n*-2 etc.

Set a binary output on or off. For example:

```
OUT(35)=1 : OUT(36)=ON
```

or

```
OUT(35,2)=3
```

sets on outputs 35 and 36.

```
OUT(100,8)=%10011001
```

sets on output 100,103,104,107 and sets off outputs 101,102,105,106.

Function	Read output status.
Syntax	OUT(*a*)
Type	Truth value.
a	Address of output.
Values	Output status(es) as in INP function.

OUT function is used for reading statuses of outputs. If an 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
```

13.3 ANALOG I/O

Analog I/O may be available as McWay analog I/O modules or as EtherCat fieldbus connected analog I/O modules. Analog I/O is accessed using the INPA and OUTA commands and functions.

13.3.1 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 every i/o cycle. Thus the maximum sample rate is determined by PIDFREQ.

13.3.2 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.

13.4 STATUSOUTS

The STATUSOUTS command can be used to configure some outputs in the system I/O to reflect system status.

Command	Configure system status outputs.
Syntax	STATUSOUTS(*run*,*noerr*[, *timeout*])
run	Number of run output. On (=1) when application program is running normally.
noerr	Number of no error output. On (=1) when no runtime error has been detected and
timeout	Timeout parameter [s]. During running McBasic checks the console connection for ctrl-X characters. If *timeout* is exceeded between consecutive checks, both *run* and *noerr* outputs go off. Default value 0.25.

STATUSOUTS can be used to add to system safety by using some outputs to stop the system in case of system failure or program error. To achieve this, an output can be connected to operate emergency stop, for example.

Giving a value of -1 for *run* or *noerr* cancel the configuration for the respective output.

14. 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
```

14.1 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
```

14.2 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
```

14.3 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
```

14.4 ERR

Function	Number of the last occurred error.
Syntax	ERR
Type	Integer (0 .. 127)

14.5 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.

```
PRINT "Error #";ERR;
PRINT " on line ";ERL
```

14.6 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$
```

14.7 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
```

14.8 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)
```

14.9 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.

Appendix 1, list of EtherCat device configuration strings for ECMOD$

addr is the address (number) of the first input or output in the device of subnode.
axis is the number of the axis used to refer to a drive output PWR(*axis*) or position input POS(*axis*)

Configuration string	description
Generic devices	
"INP2 INP(*addr*)"	2 bit binary input
"INP4 INP(*addr*)"	4 bit binary input
"INP8 INP(*addr*)"	8 bit binary input
"INP16 INP(*addr*)"	16 bit binary input
"OUT2 OUT(*addr*)"	2 bit binary output
"OUT4 OUT(*addr*)"	4 bit binary output
"OUT8 OUT(*addr*)"	8 bit binary output
"OUT16 OUT(*addr*)"	16 bit binary output
"DRIVE PWR(*axis1*)[POS(*axis2*)]"	drive, *axis1* is the axis to control, *axis2* is the encoder to measure
"UNKNOWN"	unknown ethercat device
SKS Control devices:	
"ACN/EIO IO(*addr*)"	Base module with 32 bit binary input/outputs and 4 option slots
subnodes (options):	
"ENC1/INC POS(*axis*) PWR(*axis*) OUT(*addr*)"	Axis connection with incremental encoder
"ENC1/ABS POS(*axis*) PWR(*axis*) OUT(*addr*)"	Axis connection with SSI absolute encoder
Crevis NA devices:	
"NA-9186" ethercat coupler	
"NA-9286" ethercat coupler	
subnodes (i/o slices):	
"ST-1114 INP(*addr*)"	4 bit binary input, 5V DC
"ST-111F INP(*addr*)"	16 bit binary input
"ST-1124 INP(*addr*)"	4 bit binary input, source, 5V DC
"ST-112F INP(*addr*)"	16 bit binary input
"ST-1214 INP(*addr*)"	4 bit binary input, sink, 12/24V DC
"ST-1218 INP(*addr*)"	8 bit binary input
"ST-121F INP(*addr*)"	16 bit binary input
"ST-1224 INP(*addr*)"	4 bit binary input, source, 12/24V DC
"ST-1228 INP(*addr*)"	8 bit binary input,
"ST-122F INP(*addr*)"	16 bit binary input
"ST-1314 INP(*addr*)"	4 bit binary input, sink, 48V DC
"ST-1318 INP(*addr*)"	8 bit binary input
"ST-131F INP(*addr*)"	16 bit binary input
"ST-1324 INP(*addr*)"	4 bit binary input, source, 48V DC
"ST-1328 INP(*addr*)"	8 bit binary input,
"ST-132F INP(*addr*)"	16 bit binary input
"ST-1804 INP(*addr*)"	4 bit binary input, 120V AC (AC 85V~132V)
"ST-1904 INP(*addr*)"	4 bit binary input, 240V AC (AC 170V~264V)
"ST-2114 OUT(*addr*)"	4 bit binary output, TTL inverting, 5V DC/20mA
"ST-2118 OUT(*addr*)"	8 bit binary output
"ST-221F OUT(*addr*)"	16 bit binary output, sink, 24V DC/0.5A
"ST-222F OUT(*addr*)"	16 bit binary output, source, 24V DC/0.5A
"ST-2314 OUT(*addr*)"	4 bit binary output, sink, 24V DC/0.5A
"ST-2318 OUT(*addr*)"	8 bit binary output
"ST-2324 OUT(*addr*)"	4 bit binary output
"ST-2328 OUT(*addr*)"	8 bit binary output
"ST-2414 OUT(*addr*)"	4 bit binary output, sink, diagnostics, 24V DC/0.5A
"ST-2418 OUT(*addr*)"	8 bit binary output
"ST-2514 OUT(*addr*)"	4 bit binary output, sink, diagnostics, 24V DC/2A
"ST-2518 OUT(*addr*)"	8 bit binary output,
"ST-2614 OUT(*addr*)"	4 bit binary output, sink, 24Vdc/2A
"ST-2624 OUT(*addr*)"	4 bit binary output, source, 24Vdc/2A
"ST-2742 OUT(*addr*)"	2 bit binary output

"ST-2744 OUT(*addr*)"	4 bit binary output
"ST-2748 OUT(*addr*)"	8 bit binary output
"ST-2792 OUT(*addr*)"	2 bit binary output, source, 240Vac/2A, Manual Type
"ST-3114 INPA(*addr*)"	4 channel analog input, 0~20mA, 12Bit, RTB
"ST-3118 INPA(*addr*)"	8 channel analog input
"ST-3134 INPA(*addr*)"	4 channel analog input, 0~20mA, 14Bit, RTB
"ST-3214 INPA(*addr*)"	4 channel analog input, 4~20mA, 12Bit, RTB
"ST-3218 INPA(*addr*)"	8 channel analog input
"ST-3234 INPA(*addr*)"	4 channel analog input, 4~20mA, 14Bit, RTB
"ST-3274 INPA(*addr*)"	4 channel analog input, 4~20mA, 12Bit, status
"ST-3424 INPA(*addr*)"	4 channel analog input, 0~10Vdc, 12Bit, RTB
"ST-3428 INPA(*addr*)"	8 channel analog input
"ST-3444 INPA(*addr*)"	4 channel analog input
"ST-3474 INPA(*addr*)"	4 channel analog input, 0~10V DC, 12Bit
"ST-3524 INPA(*addr*)"	4 channel analog input, -10~+10Vdc, 12Bit, RTB
"ST-3544 INPA(*addr*)"	4 channel analog input, -10~+10Vdc, 14Bit, RTB
"ST-3624 INPA(*addr*)"	4 channel analog input, 0~5Vdc, 12Bit, RTB
"ST-3644 INPA(*addr*)"	4 channel analog input, 0~5Vdc, 14Bit, RTB
"ST-4112 OUTA(*addr*)"	2 channel analog output, 0~20mA, 12Bit, RTB
"ST-4114 OUTA(*addr*)"	4 channel analog output
"ST-4212 OUTA(*addr*)"	2 channel analog output, 4~20mA, 12Bit, RTB
"ST-4214 OUTA(*addr*)"	4 channel analog output
"ST-4274 OUTA(*addr*)"	4 channel analog output, 4~20mA, 12Bit
"ST-4422 OUTA(*addr*)"	2 channel analog output
"ST-4424 OUTA(*addr*)"	4 channel analog output
"ST-4474 OUTA(*addr*)"	4 channel analog output, 0~10V, 12Bit
"ST-4522 OUTA(*addr*)"	2 channel analog output
"ST-4622 OUTA(*addr*)"	2 channel analog output, 0~5V, 12Bit, RTB
"ST-5101 POS(*axis*)"	incremental encoder input
"ST-5351 POS(*axis*)"	ssi encoder input

Crevis RT devices

"RN-9286" ethercat control device

subnodes (i/o slices):

"RT-1238 INP(*addr*)"	8 bit binary input
"RT-2328 OUT(*addr*)"	8 bit binary output
"RT-12DF INP(*addr*)"	16 bit binary input
"RT-226F OUT(*addr*)"	16 bit binary output
"RT-1218 INP*(addr)*"	8 bit binary input, sink, 12V / 24Vdc
"RT-1228 INP*(addr)*"	8 bit binary input, source, Terminal, 12V / 24Vdc
"RT-1238 INP*(addr)*"	8 bit binary input, sink/source, 24Vdc
"RT-12DF INP*(addr)*"	16 bit binary input
"RT-1804 INP*(addr)*"	4 bit binary input, 110Vac (AC 85V ~ 132V)
"RT-1904 INP*(addr)*"	4 bit binary input, 220Vac (AC 170V ~ 264V)
"RT-225F OUT*(addr)*"	16 bit binary output, sink, 24Vdc / 0.5A
"RT-226F OUT*(addr)*"	16 bit binary output
"RT-2318 OUT*(addr)*"	8 bit binary output, sink, 24Vdc / 0.5A
"RT-2328 OUT*(addr)*"	8 bit binary output, source, 24Vdc / 0.5A
"RT-2428 OUT*(addr)*"	8 bit binary output, source, self -Diagnostic, 24Vdc / 0.5A
"RT-2734 OUT*(addr)*"	4 bit binary output, MOS Relay,220V,110V, AC/DC, 0.5A
"RT-2744 OUT*(addr)*"	4 bit binary output, 230Vac / 2A, 24Vdc/2A
"RT-2748 OUT*(addr)*"	8 bit binary output, 230Vac / 2A, 24Vdc/2A
"RT-2772 OUT*(addr)*"	2 bit binary output, 24Vdc / 220Vac/2A
"RT-2944 OUT*(addr)*"	4 bit binary output, MOS Relay, AC /DC Output, 24V/2A
"RT-3114 INPA*(addr)*"	4 channel analog input, 0~20mA, 12Bit, status
"RT-3118 INPA*(addr)*"	8 channel analog input
"RT-3134 INPA*(addr)*"	4 channel analog input, 0~20mA, 14Bit, status
"RT-3138 INPA*(addr)*"	8 channel analog input
"RT-3154 INPA*(addr)*"	4 channel analog input, 0~20mA, 15Bit, status
"RT-3158 INPA*(addr)*"	8 channel analog input
"RT-3214 INPA*(addr)*"	4 channel analog input, 4~20mA, 12Bit, status
"RT-3218 INPA*(addr)*"	8 channel analog input
"RT-3234 INPA*(addr)*"	4 channel analog input, 4~20mA, 14Bit, status
"RT-3238 INPA*(addr)*"	8 channel analog input
"RT-3254 INPA*(addr)*"	4 channel analog input, 4~20mA, 15Bit, status
"RT-3258 INPA*(addr)*"	8 channel analog input
"RT-3424 INPA*(addr)*"	4 channel analog input, 0~10Vdc, 12Bit, status

```
"RT-3428 INPA(addr)"                8 channel analog input
"RT-3444 INPA(addr)"                4 channel analog input, 0~10Vdc, 14Bit, status
"RT-3448 INPA(addr)"                8 channel analog input
"RT-3464 INPA(addr)"                4 channel analog input, 0~10Vdc, 15Bit, status
"RT-3468 INPA(addr)"                8 channel analog input
"RT-3624 INPA(addr)"                4 channel analog input, 0~5Vdc, 12Bit, status
"RT-3628 INPA(addr)"                8 channel analog input
"RT-3644 INPA(addr)"                4 channel analog input, 0~5Vdc, 14Bit, status
"RT-3648 INPA(addr)"                8 channel analog input
"RT-3664 INPA(addr)"                4 channel analog input, 0~5Vdc, 14Bit, status
"RT-3668 INPA(addr)"                8 channel analog input
"RT-3704 INPA(addr)"                4 channel analog input, RTD, status
"RT-3804 INPA(addr)"                4 channel analog input, Thermocouple
"RT-3914 INPA(addr)"                4 channel analog input, 12bit diff., 0~20mA, 4~20mA, -20~20mA
"RT-3924 INPA(addr)"                4 channel analog input, 12bit diff., 0~10V, 0~5V, -10~10V, -5~5V
"RT-3934 INPA(addr)"                4 channel analog input, 15bit diff., 0~20mA, 4~20mA, -20~20mA
"RT-3944 INPA(addr)"                4 channel analog input, 15bit diff., 0~10V, 4~5V, -10~10V, -5~5V
"RT-4114 OUTA(addr)"                4 channel analog output, 0~20mA, 12Bit
"RT-4118 OUTA(addr)"                8 channel analog output
"RT-4134 OUTA(addr)"                4 channel analog output, 0~20mA, 14Bit
"RT-4138 OUTA(addr)"                8 channel analog output
"RT-4154 OUTA(addr)"                4 channel analog output, 0~20mA, 15Bit
"RT-4158 OUTA(addr)"                8 channel analog output,
"RT-4214 OUTA(addr)"                4 channel analog output, 4~20mA, 12Bit
"RT-4218 OUTA(addr)"                8 channel analog output,
"RT-4234 OUTA(addr)"                4 channel analog output, 4~20mA, 14Bit
"RT-4238 OUTA(addr)"                8 channel analog output,
"RT-4254 OUTA(addr)"                4 channel analog output, 4~20mA, 15Bit
"RT-4258 OUTA(addr)"                8 channel analog output,
"RT-4424 OUTA(addr)"                4 channel analog output, 0~10V, 12Bit
"RT-4428 OUTA(addr)"                8 channel analog output,
"RT-4444 OUTA(addr)"                4 channel analog output, 0~10V, 14Bit
"RT-4448 OUTA(addr)"                8 channel analog output,
"RT-4464 OUTA(addr)"                4 channel analog output, 0~10V, 15Bit
"RT-4468 OUTA(addr)"                8 channel analog output,
"RT-4524 OUTA(addr)"                4 channel analog output, -10~10V, 12Bit
"RT-4544 OUTA(addr)"                4 channel analog output, -10~10V, 14Bit
"RT-4564 OUTA(addr)"                4 channel analog output, -10~10V, 15Bit
"RT-4624 OUTA(addr)"                4 channel analog output, 0~5V, 12Bit
"RT-4628 OUTA(addr)"                8 channel analog output,
"RT-4644 OUTA(addr)"                4 channel analog output, 0~5V, 14Bit
"RT-4648 OUTA(addr)"                8 channel analog output,
"RT-4664 OUTA(addr)"                4 channel analog output, 0~5V, 15Bit
"RT-4668 OUTA(addr)"                8 channel analog output,
```

Wago devices

```
"750-354" ethercat coupler
```

subnodes (i/o slices):

```
"750-431 INP(addr)"                 8 bit binary input
"750-454 INPA(addr)"                2 bit analog input
"750-459 INPA(addr)"                4 bit analog input
"750-461 INPA(addr)"                2 bit analog input
"750-469 INPA(addr)"                2 bit analog input
"750-476 INPA(addr)"                2 bit analog input
"750-512 OUT(addr)"                 2 bit binary output
"750-530 OUT(addr)"                 8 bit binary output
"750-556 OUTA(addr)"                2 bit analog output
"750-630 POS(axis)"                 ssi encoder input
```

Beckhoff devices

"EK1100"	ethercat coupler

subnodes (i/o slices):

"EL1002 INP(*addr*)"	2 bit binary input	24vdc 3ms typ3 pnp
"EL1004 INP(*addr*)"	4 bit binary input	24vdc 3ms typ3 pnp
"EL1008 INP(*addr*)"	8 bit binary input	24vdc 3ms typ3 pnp
"EL1012 INP(*addr*)"	2 bit binary input	24vdc 10us typ3 pnp
"EL1014 INP(*addr*)"	4 bit binary input	24vdc 10us typ3 pnp
"EL1018 INP(*addr*)"	8 bit binary input	24vdc 10us typ3 pnp
"EL1024 INP(*addr*)"	4 bit binary input	24vdc 3ms typ2 pnp
"EL1034 INP(*addr*)"	4 bit binary input	24vdc 10us typ1 pnp
"EL1084 INP(*addr*)"	4 bit binary input	24vdc 3ms typ3 npn
"EL1088 INP(*addr*)"	8 bit binary input	24vdc 3ms typ3 npn
"EL1094 INP(*addr*)"	4 bit binary input	24vdc 10us typ3 npn
"EL1098 INP(*addr*)"	8 bit binary input	24vdc 10us typ3 npn
"EL1124 INP(*addr*)"	4 bit binary input	5vdc 10uS
"EL1134 INP(*addr*)"	4 bit binary input	48vdc 10us typ1
"EL1144 INP(*addr*)"	4 bit binary input	12vdc 10uS
"EL1252 INP(*addr*)"	2 bit binary input	24vdc 1us typ3 npn
"EL1702 INP(*addr*)"	2 bit binary input	230v
"EL1712 INP(*addr*)"	2 bit binary input	120v
"EL1722 INP(*addr*)"	2 bit binary input	230v
"EL1862 INP(*addr*)"	16 bit binary input	24vdc 3ms typ3 pnp
"EL1872 INP(*addr*)"	16 bit binary input	24vdc 10us typ3 pnp
"EL2002 OUT(*addr*)"	2 bit binary output	24vdc 0.5A
"EL2004 OUT(*addr*)"	4 bit binary output	24vdc 0.5A
"EL2008 OUT(*addr*)"	8 bit binary output	24vdc 0.5A
"EL2022 OUT(*addr*)"	2 bit binary output	24vdc 2.0A
"EL2024 OUT(*addr*)"	4 bit binary output	24vdc 2.0A
"EL2042 OUT(*addr*)"	16 bit binary output	24vdc 0.5A
"EL2084 OUT(*addr*)"	4 bit binary output	24vdc 2.0A npn
"EL2088 OUT(*addr*)"	8 bit binary output	24vdc 2.0A npn
"EL2602 OUT(*addr*)"	2 bit binary output	230v 5.0A relay
"EL2612 OUT(*addr*)"	2 bit binary output	230v 2.0A relay
"EL2622 OUT(*addr*)"	2 bit binary output	230v 2.0A relay
"EL2624 OUT(*addr*)"	4 bit binary output	230v 2.0A relay
"EL2652 OUT(*addr*)"	2 bit binary output	230v 1.0A relay
"EL2712 OUT(*addr*)"	2 bit binary output	12-230v 0.5A triac
"EL2722 OUT(*addr*)"	2 bit binary output	12-230v 1.0A triac
"EL2732 OUT(*addr*)"	2 bit binary output	12-230v 0.5A triac
"EL2828 OUT(*addr*)"	8 bit binary output	24vdc 2.0A
"EL2872 OUT(*addr*)"	16 bit binary output	24vdc 2.0A
"EL4001 OUTA(*addr*)"	1 analog output	12-bit 0..10V
"EL4002 OUTA(*addr*)"	2 analog output	12-bit 0..10V
"EL4004 OUTA(*addr*)"	4 analog output	12-bit 0..10V
"EL4008 OUTA(*addr*)"	8 analog output	12-bit 0..10V
"EL4031 OUTA(*addr*)"	1 analog output	12-bit -10..10V
"EL4032 OUTA(*addr*)"	2 analog output	12-bit -10..10V
"EL4034 OUTA(*addr*)"	4 analog output	12-bit -10..10V
"EL4038 OUTA(*addr*)"	8 analog output	12-bit -10..10V
"EL5001 POS(*axis*)"	1 SSI encoder interface	
"EL5002 POS(*axis*)"	2 SSI encoder interface	

Control Techniques devices (drives)

"COMMANDER/SK PWR(*axis*)"	
"UNIDRIVE/SP [PWR(*axis1*)] POS(*axis2*)"	*axis1* is number of axis, *axis2* is number of encoder
"UNIDRIVE/SP POS(*axis*)"	drive, *axis* is number of axis and encoder
"UNIDRIVE/M300 PWR(*axis*)"	drive, *axis* is number of axis, no encoder
"UNIDRIVE/M400 PWR(*axis*)"	drive, *axis* is number of axis, no encoder
"UNIDRIVE/M600 PWR(*axis*)"	drive, *axis* is number of axis, no encoder
"UNIDRIVE/M700 [PWR(*axis1*)] POS(*axis2*)]"	*axis1* is number of axis, *axis2* is number of encoder
"UNIDRIVE/M701 [PWR(*axis1*)] POS(*axis2*)]"	*axis1* is number of axis, *axis2* is number of encoder
"UNIDRIVE/M702 [PWR(*axis1*)] POS(*axis2*)]"	*axis1* is number of axis, *axis2* is number of encoder