



Danaher**Motion**

---

# **Motion Controller MC-600**

*Programmable in MotionBasic*

*Add-on card for the servo amplifier  
SERVOSTAR™ 600*

**User's Manual**

September 2002

# Index

Safety information	4
Installation	5
Block diagram	7
Technical data	7
Connections	8
Serial interface RS-232 / PC	8
Servobus	9
Msetsb(mode)	9
Mcantx(id,data)	9
Mcanrx(id,data)	10
Digital Inputs	11
Digital Outputs	12
How the MC-600 works	13
The “Motion planner”	14
The “Motion planner” Units	16
The Motion functions	17
Mspeed(vel)	17
Mmove(pos)	18
Mimove(ipos)	18
Mmerge(pos)	19
Mimerge(ipos)	19
Mgear(type)	20
mcam(table)	23
mspline(xy%,np%,camtab%)	25
Mvirtualax(pos)	26
Mvspeed(vel)	26
Mphase(pos)	26
Midle()	27
Mhome(cmd)	27
The Interrupt functions	29
Source	29
The Drive related functions	31
The I/O functions	32
The other system variables of the “Motion planner”	34

Introduction	36
The operating system	36
Basic programming rules	38
CHARACTER	38
NAME and DESCRIPTION	38
Programming numbers and variables	39
<i>Integer, floating point and string constants</i>	39
<i>Integer floating point and string variables</i>	42
<i>Integer, floating point and string arrays</i>	42
<i>Expressions and operators</i>	44
<i>Arithmetic expressions</i>	44
<i>Arithmetic operators</i>	44
<i>Relational operators</i>	46
<i>Logical operators</i>	47
<i>Hierarchy of operations</i>	49
<i>String operations</i>	50
<i>String expressions</i>	50
<i>Programming techniques</i>	51
<i>The BASIC Filesystem</i>	52
Description of BASIC keywords	54

# Safety information

Using the motion controller MC-600 it is possible change on line system parameters whilst motor is running in order to increase the performances of the system.

Before to use this MC-600 capability be sure that does give not rise to dangerous situations that could endanger the safety of people close to the moving part of the machine or lead to damage to the machine itself.

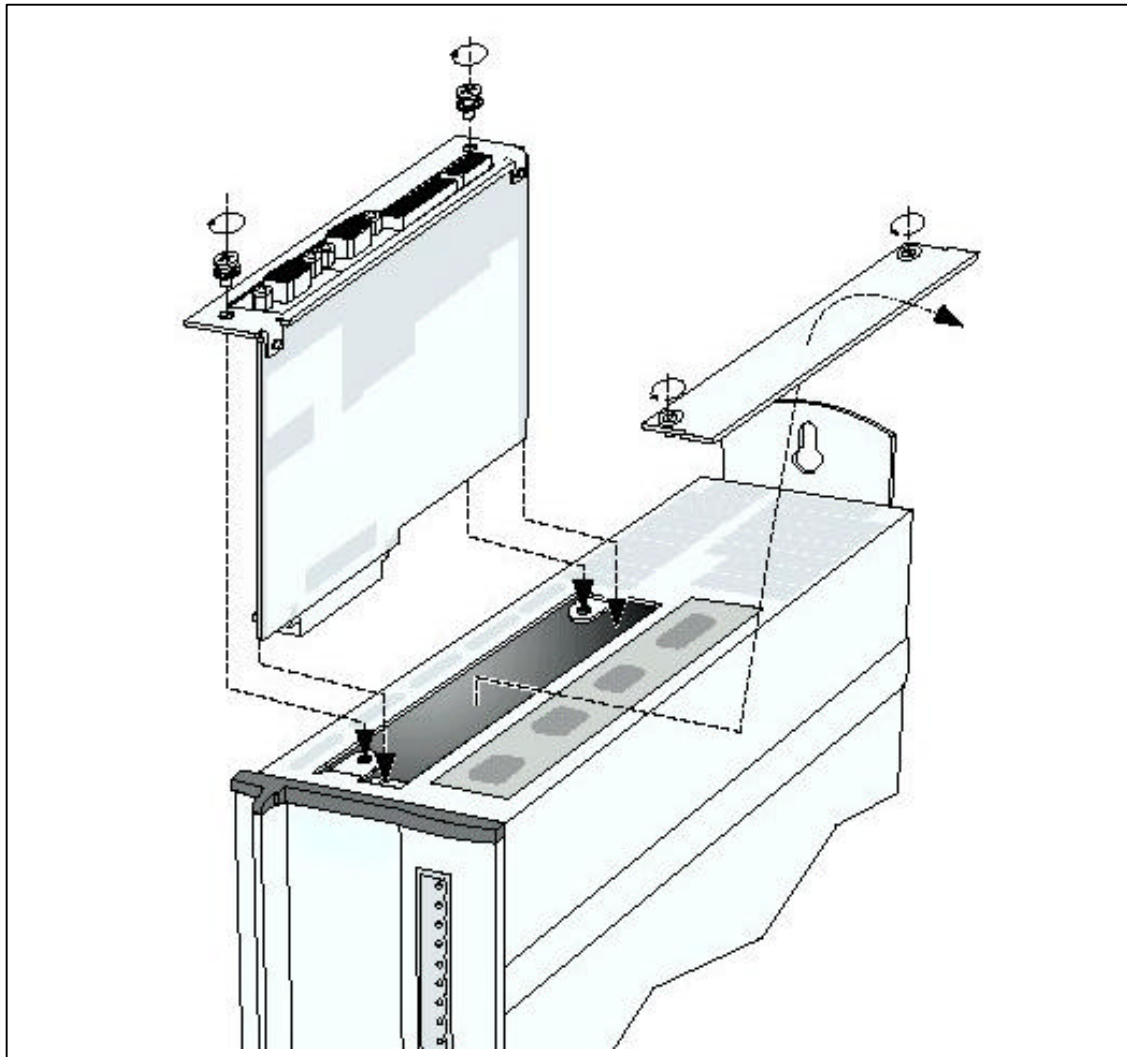


To avoid any risk take in consideration the following basic rules:

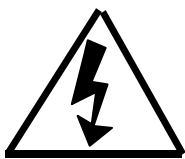
- Before to change the value of any motion variable check that it is correct and evaluate what the possible effect could be.
- Take adequate electrical safety precautions such as fitting limit switches and emergency off switches.
- Only qualified and trained personnel with a good knowledge of electronics, motion programming and converter technology is allowed to commissioning.

The add-on card MC-600 is intend for operating with the servo amplifier SERVOSTAR 600 manufactured by Kollmorgen Seidel gmbh, please refer to the supplied installation manual for the SERVOSTAR 600. There pay attention to the safety notes at the start of the manual as well as the information given about directives and standard.

# Installation



Power-off the **SERVOSTAR 600** before installing or removing the MC-600.



Remove the blanking plate from the SERVOSTAR 600.

Insert the card into the guide rails in the slot and without tilting it push it in all the way, you must be able to fill the bus connector clipping into place.

Fit the two screws using them to fix the card in place.

Perform the electrical connections.

Download the application software.

## Servostar 600

It is assumed that the servo amplifier is already fitted into the cubicle and all the electrical connections have been done and checked. If this isn't the case **please do it now with the MC-600 unplugged**. To do this, refer to the assembly and installation instructions and the quickstart manual as well as the commissioning software of the SERVOSTAR 600.

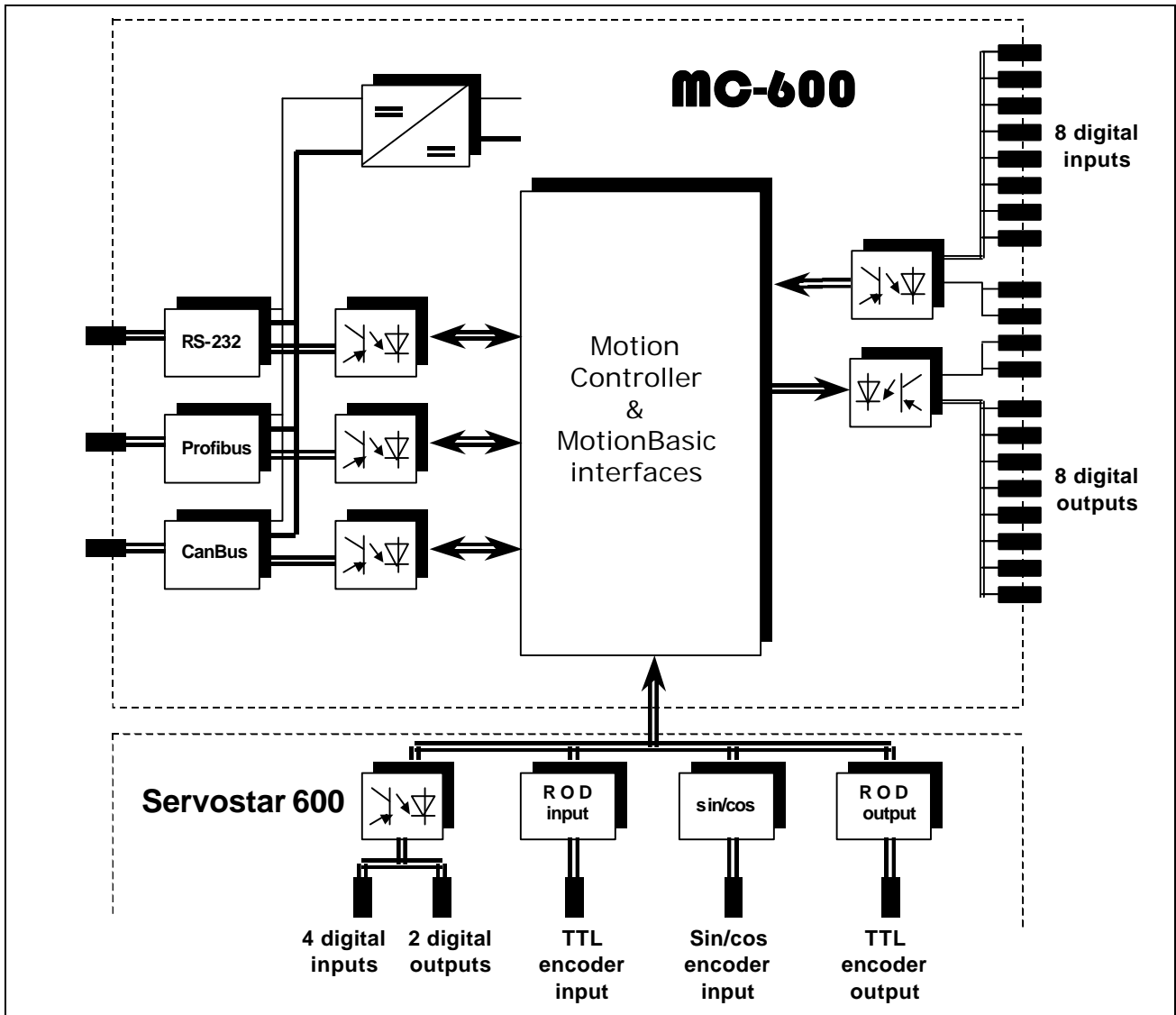


The firmware installed in the Servostar 600 must be greater or equal to 4.94

A few parameters of the SERVOSTAR 600 have a special meaning in conjunction with the MC-600 (set them using the commissioning software):

- OPMODE (screen page "amplifier") setting is made automatically from the MC-600.
- If used chose the encoder type for the master axis.
- Use of the amplifier's own output by the MC-600:
  - Set code 6 under 'I/O analog' for the 'MONITOR 1/2 ' variables.
  - Set code 23 under 'I/O digital' for the 'DIGITAL-OUT 1/2 ' variables.
- Load one of the standard setup file in the Servostar
- Set the name of the application to run in the Servostar ASCII command ALIAS, or set the password to start in command mode. The default password is PASSWORD, refer to the basic command **password** to change it.

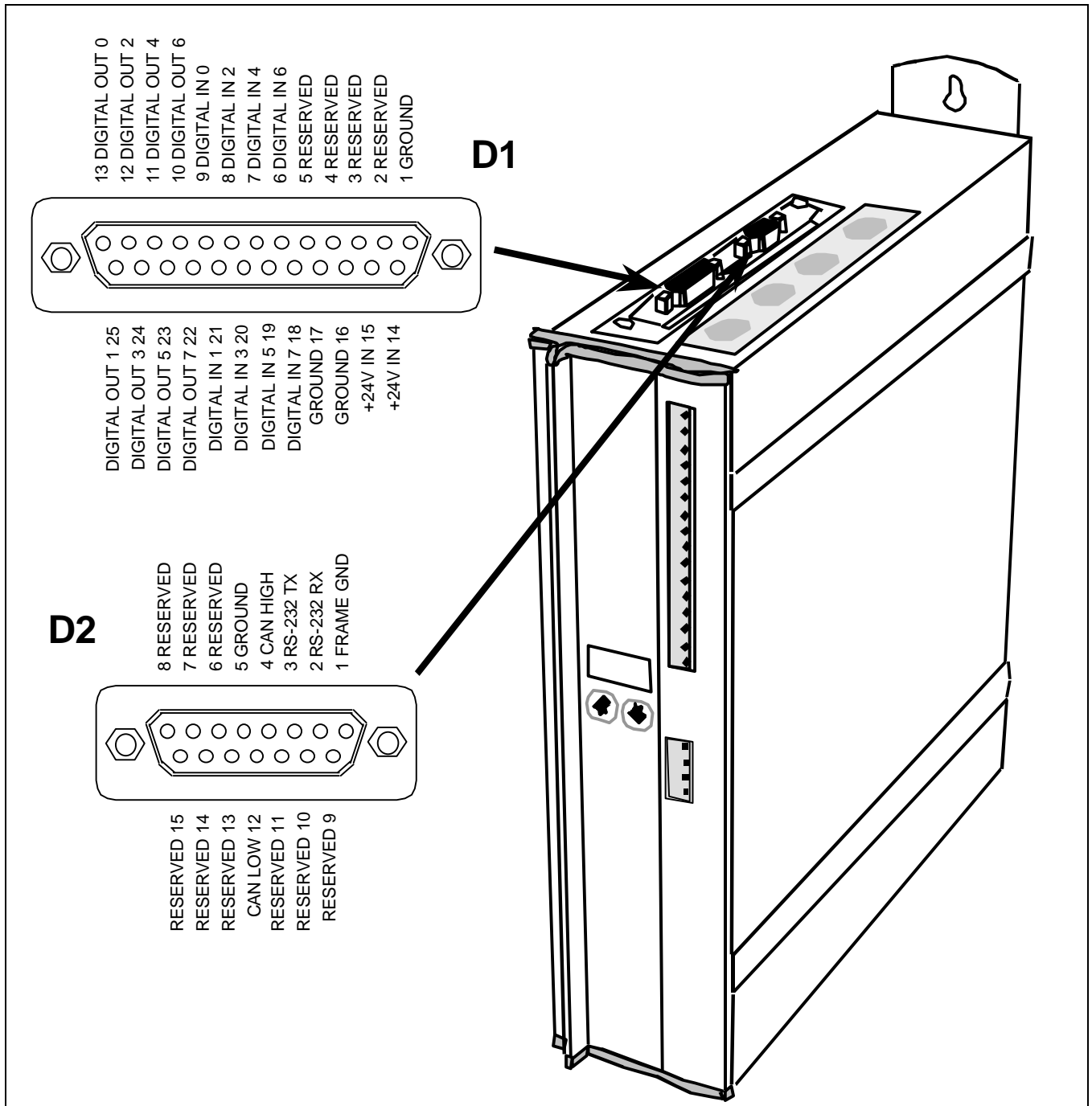
# Block diagram



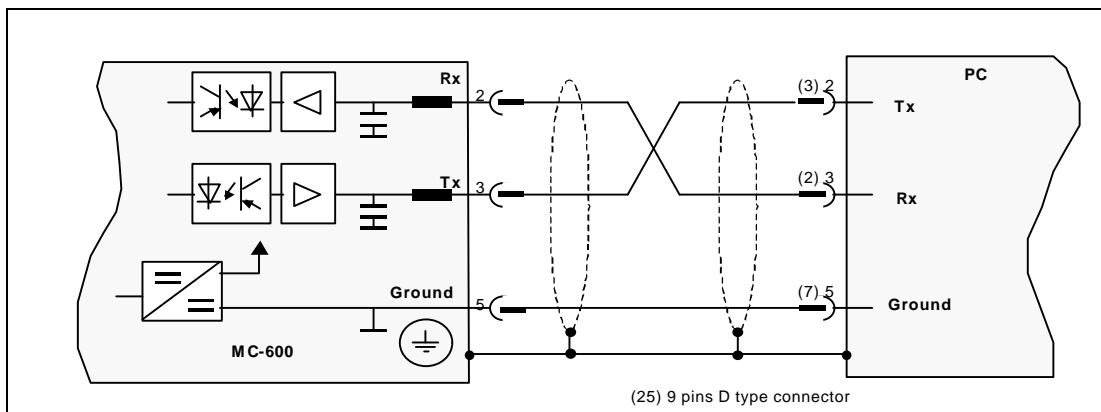
## Technical data

Motion controller	Processor	SAB C167
	Flash memory	1 Mbyte
	RAM	1 Mbyte
	E <sup>2</sup> PROM	2 Kbyte
Digital inputs	8	electrically isolated
	Input impedance	>2 Kohm
	Low logic level	0 to 5V
	High logic level	15 to 30V
Digital outputs	8	electrically isolated
	Surge current	20mA
Profibus-dp	up to 12 Mbps	
CanBus	Application layers	ServoBus DeviceNet
RS-232	9600 bauds	
Programming tool	Windows Hyper Terminal	

# Connections



## Serial interface RS-232 / PC



# Servobus

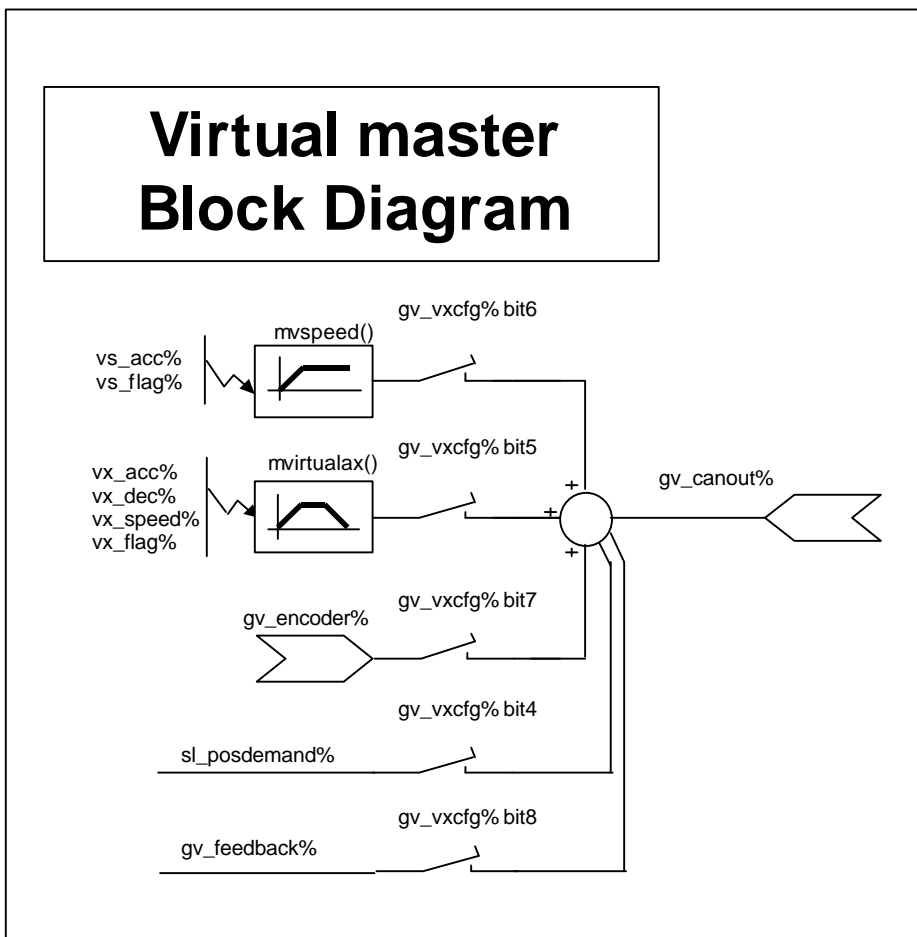
Using the ServoBus it is possible synchronize up to 16 MC-600 motion cards. When is used the synchronization at every sampling time the MC-600 master send the sync command to any MC-600 slave, along with the sync command is also sent one of the position variables which can be selected as master position from any slave unit. With the ServoBus it is also possible exchange MotionBasic variables among the MC-600 cards connected via the ServoBus using the MotionBasic functions MCantx() and MCanrx().

## Msetsb(mode)

When the function **msetsb(1)** is performed the MC-600 is set as master otherwise by default or with the function **msetsb(0)** it's set has slave.

The data sent from the master can be read in the variable gv\_canout%.

The data received from the slave can be read in the variable gv\_canin%.



## Mcantx(id,data)

The function **mcantx(id,data)** send the data **data** with the identifier **id**. The identifier must be between 1 and 2047.

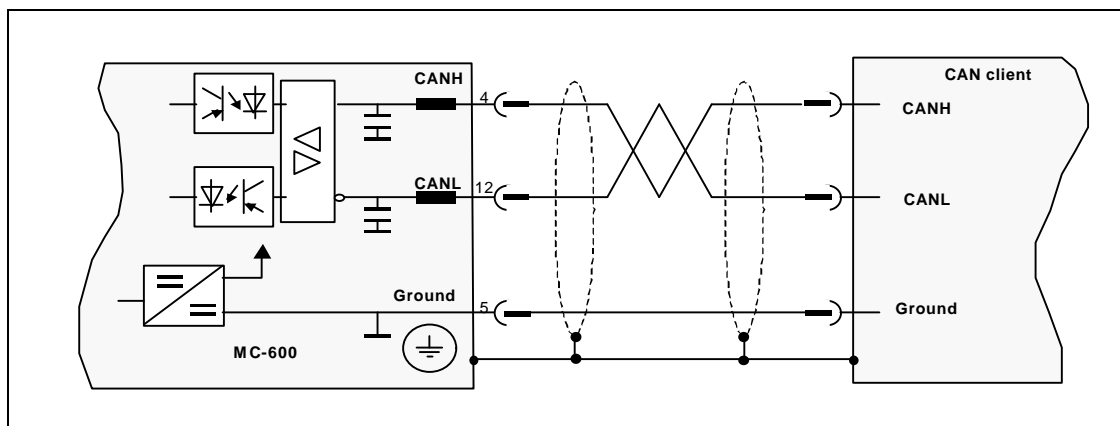
## Mcanrx(box,id)

The function **mcanrx(box,id)** program the mailbox **box** in order to receive the telegram with the identifier **id**. The box must be between 0 and 7, the identifier between 1 and 2047. The data received will be stored in the array `can_data%`. The variable `can_flag%` signal if the mailboxes have been updated (bit 0 mailbox 0, bit 7 mailbox 7).

If the MC-600 has to be a CanOpen slave it is possible to use the Can interface built in the **SERVOSTAR 600** through the 8 user variables of the amplifier.



If the module is **the first or the last unit** of the Bus, the line must be terminated with an external resistor of 120 ohm (soldered into the plug).



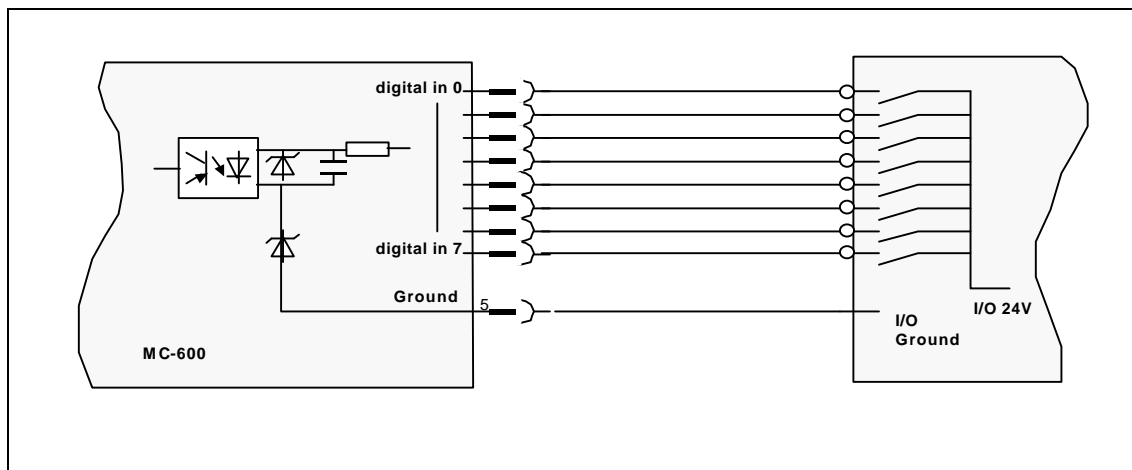
# Digital Inputs

All 8 Digital inputs are electrically insulated via optocouplers. The function of any input can be defined using the application program written in MotionBasic.

The Digital input 2 and 3 can be used as dimension capture inputs.

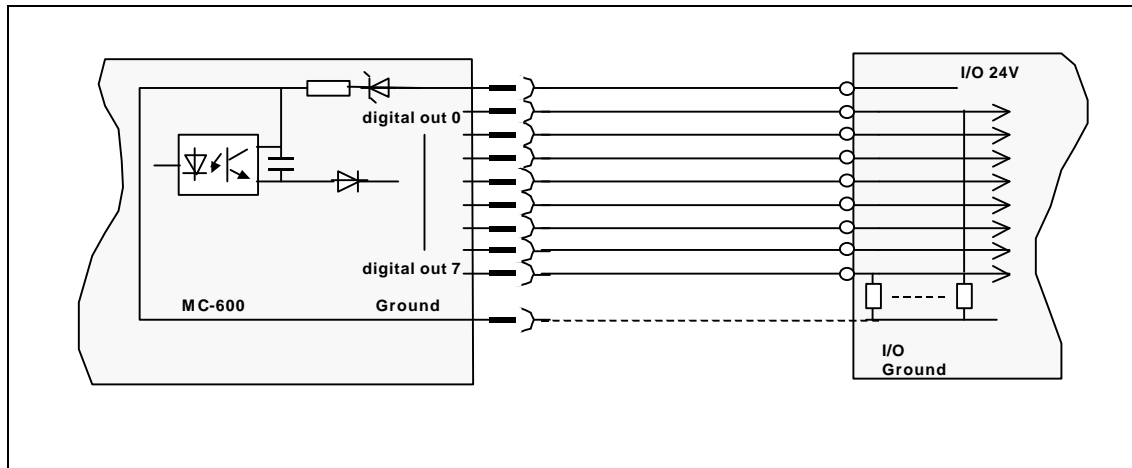
(The same function is available on D1 and D2 on the SERVOSTAR 600).

The 4 digital inputs on the SERVOSTAR 600 can be used from the application program, this capability expand the number of digital inputs available to 12.



# Digital Outputs

On the MC-600 8 floating digital outputs are available; their function is defined by the application program running on the card.



The ground connection shown as a dashed line is only required if the same external auxiliary power supply is used for the digital inputs.

The 2 digital outputs on the SERVOSTAR 600 can be used by the application program, this capability expand the number of digital inputs available to 10.



If in your application you are going to use the digital outputs on the SERVOSTAR 600 pay attention that they are NPN instead of the PNP polarity of the MC-600 digital outputs.

## How the MC-600 works

In the firmware of the MC-600 card are running two main software layers, the first one is called "*motion planner*" and its task is to compute the trajectory of the motor following the command coming from the "*MotionBasic*" layer. The sampling time of the "*motion planner*" is 2.

The MC-600 don't perform any position loop but it use the regulation loop built in the SERVOSTAR 600 which is running every 250uS, refer to the SERVOSTAR 600 documentation to get detail about the position loop.

The hardware resources of the SERVOSTAR 600 can be used by the MC-600 so you can connect different position sensor to the MC-600 to get for instance the main motor position.

The different interfaces for encoders available are:

***Incremental encoder 24V*** – plug X3

Accept both tracks, quadrature A/B and pulse/direction.

Counting frequency 10kHz

Power supply from the auxiliary voltage.

***Incremental encoder 5V*** – plug X5

Max counting frequency 2MHz

***Sin/Cosin encoder*** – plug X1

Max. frequency 250kHz.

Resolution is 1024 steps for sin period.

The application program has to be written in MotionBasic, the MotionBasic of the MC-600 is a standard BASIC interpreter so any programmer with a few experience in this language will find the MC-600 environment familiar. To use the MotionBasic you need only the HyperTerminal program supplied with any PC system, and then you are ready to program your first instruction without reading the manual, for instance typing

```
>10 print "Hello I'am the MC-600"  
>20 end  
>run [return]
```

it will work in few seconds.

The interface with the "*motion planner*" is done through MotionBasic functions and system variables.

# The “Motion planner”

The “Motion Planner” block diagram is shown in the figure below. The 4 trajectory compute blocks are:

**Velocity** – It computes a position trajectory so the motor will be running at constant speed, the acceleration, deceleration are limited at the set value.

**Move** – Is computed a trajectory to have a point to point positioning, acceleration, deceleration and cruise speed can be changed on fly.

**Gearing** – The controlled motor will follow the master encoder; the ratio and the limits of acceleration can be defined as well as the engage and release methods.

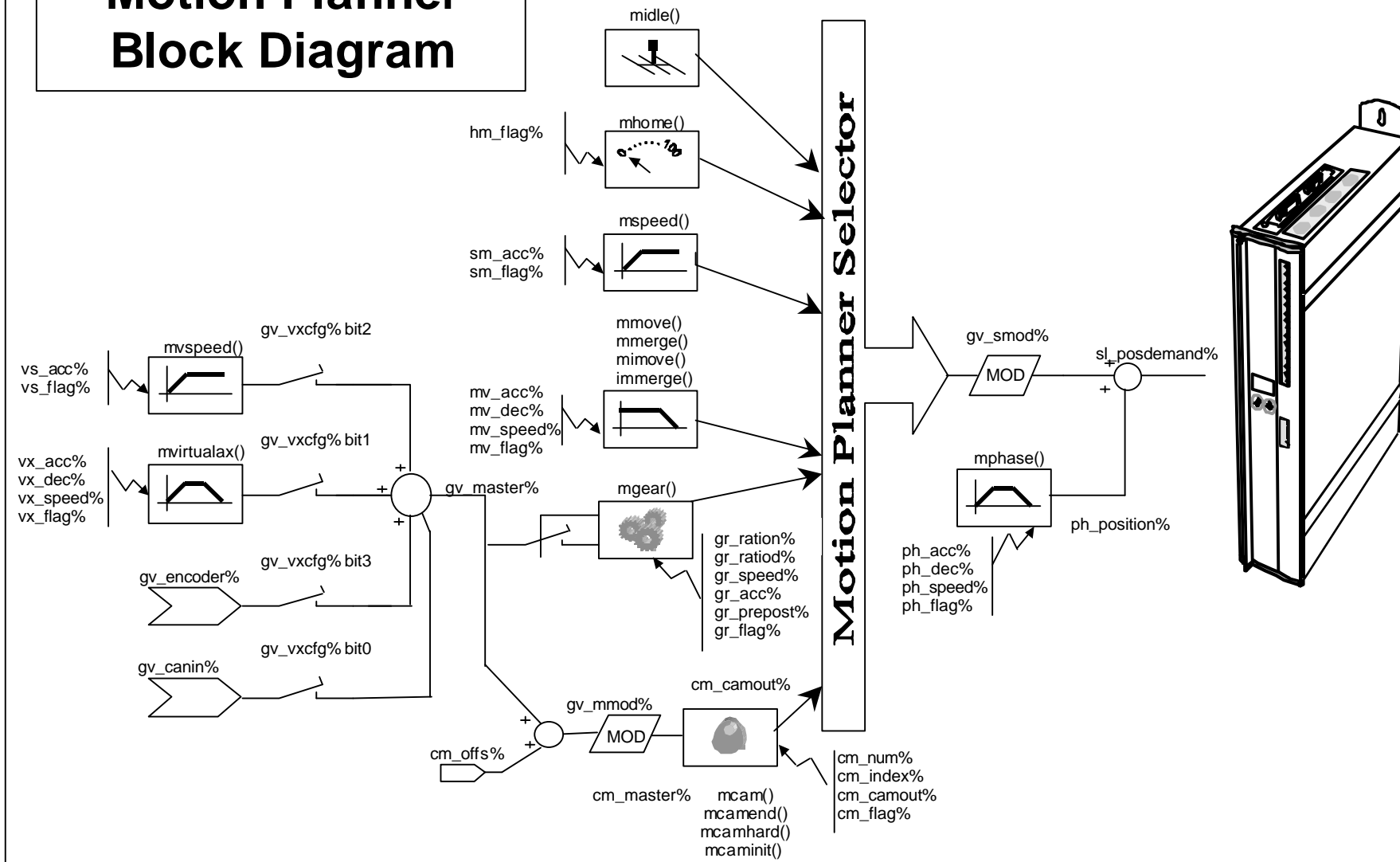
**Camming** – Using the camming function it is possible any kind of motion profile as function of the master position, the master position could be both a real axis and a virtual one.

**Master position** – Added to the master position is a move generator block using it is possible to adjust the master encoder position or, disabling the master encoder, to use it as a virtual axis.

**Phasing** – At the position demand, before to be sent to the SERVOSTAR 600, is added a move block generator to give the possibility to adjust the phase of the controlled axis.

**Homing** – The MC-600 use the homing procedures available on the SERVOSTAR 600. Please refer to the SERVOSTAR 600 if details are needed.

# Motion Planner Block Diagram



## The “Motion planner” Units

All the variables in the “motion planner” have the same format, they are 32 bits signed.

The unit of any position variable is count considering that one motor revolution is 4096 counts (**mres%** = 0) or 65536 counts (**mres%** = 1). This variable must not be changed after the homing.

The unit of any speed variable is  $(\text{count} * 256) / \text{Sampling time}$ . Considering the typical sampling time equal to 2mS, to get the right speed unit you have to multiply the speed in r.p.m. by 34.952.

The unit for any acceleration variable is  $(\text{count} * 256) / (\text{Sampling time})^2$ . Considering the typical sampling time equal to 2mS, to get the right acceleration unit you have to multiply the acceleration in rpm variation for mS by 69.904.

# The Motion functions

## Mspeed(vel)

When the function **mspeed(vel)** is performed the motor will be driven generating a position trajectory to reach the velocity "vel" as a constant speed.

The acceleration / deceleration limits are defined via the system variable **sm\_acc%**.

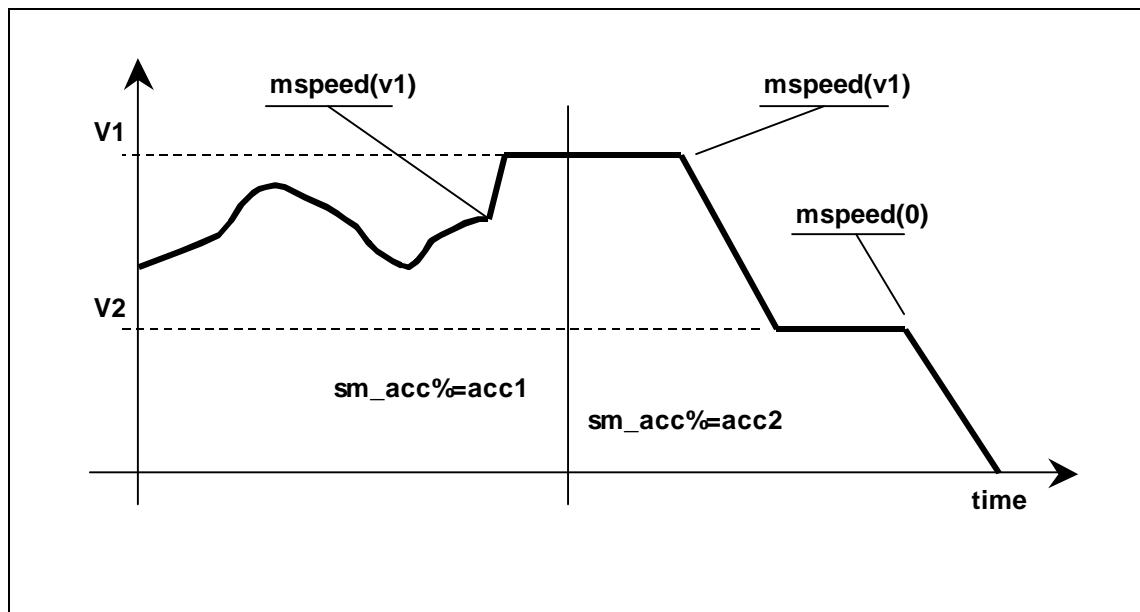
The variable **sm\_flag%** has in general utility flags. **Bit 0** is 1 when the speed mode does not contribute to the motor movement, otherwise it is 0. **Bit 2** is 1 when a reference is generated such that a speed equal to the reference and no limits to the active acceleration are required.

When the function **mspeed(vel)** is performed if a trajectory generator is active it will be aborted.

It is possible change on fly the variable **sm\_acc%** or invoking the speed function with a different vel parameter, the new speed demand will become active immediately.

If the "speed" trajectory generator has to be released the function **midle()** has to be invoked.

The actual speed demand can be read in the variable **sm\_speed%**



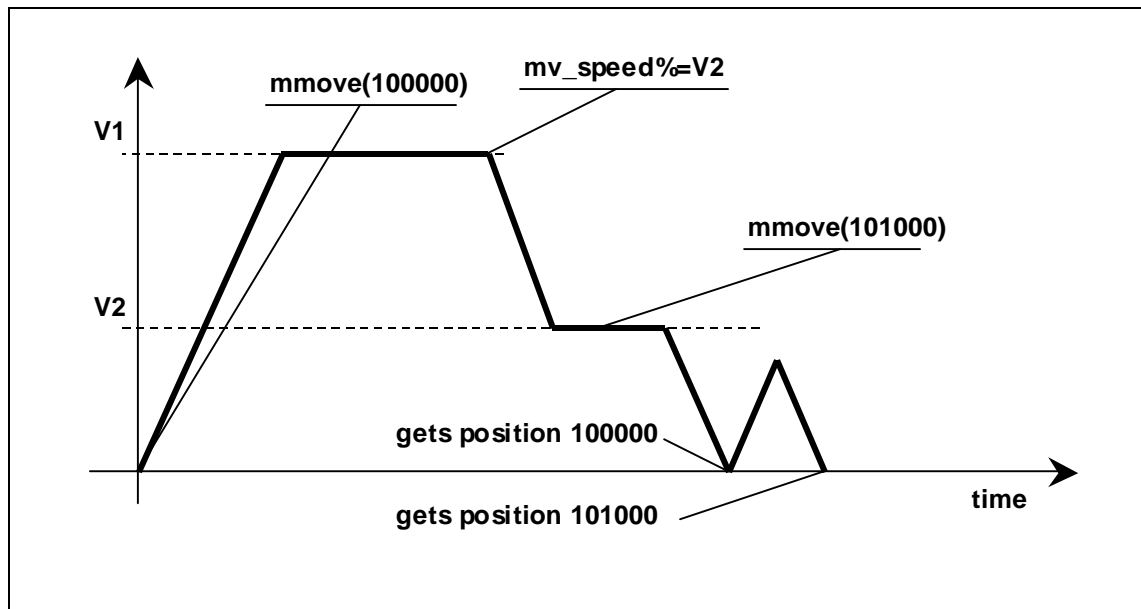
## Mmove(pos)

The **mmove(pos)** function start the generation of a move trajectory to reach the position pos starting from the actual position and the actual speed.

The move trajectory will be generated according to the acceleration limit in **mv\_acc%**, the deceleration limit in **mv\_dec%** and the maximum velocity in **mv\_speed%**.

**Mv\_flag%** is a general utility flags variable. **Bit 0** is 1 when the target position is reached therefore no moving operation is active.

Setting the bit 4 of **mv\_flag%** the movement direction will be always positive. Setting the bit 5 of **mv\_flag%** the movement direction will be always negative. Setting the bit 6 of **mv\_flag%** the motor will follow the shortest path.



## Mimove(ipos)

The function **mimove(ipos)** works as the function **mmove(pos)** with the difference that instead to perform an absolute move it perform an incremental move. The **ipos** parameter will be added to the target position reference so if an **mimove(ipos)** is invoked while a move is in progress the target position will be increased of **ipos**.

## Mmerge(pos)

The **mmerge(pos)** function start the generation of a move trajectory to reach the position pos starting from the actual position and the actual speed.

If the actual speed is smaller of **minspped%** the trajectory speed will minspped%.

The move trajectory will be generated according to the acceleration limit in **mv\_acc%**, the deceleration limit in **mv\_dec%** and the maximum velocity during the trajectory will be the motor speed when the **mmerge(pos)** function has been invoked.

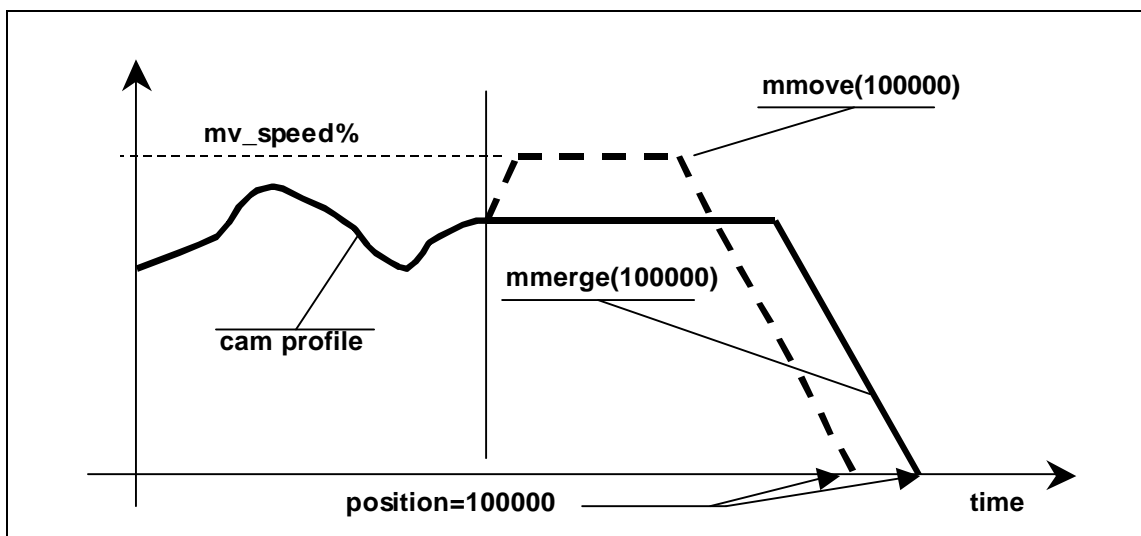
**Mv\_flag%** is a general utility flags variable. **Bit 0** is 1 when the target position is reached therefore no moving operation is active.

## Mimerge(ipos)

The function **mimerge(ipos)** works as the function **mmerge(pos)** with the difference that instead to perform an absolute move it perform an incremental move. The **ipos** parameter will be added to the target position reference so if an **mimerge(ipos)** is invoked while a move is in progress the target position will be increased of **ipos**.



If it is invoked either the **mmerge(pos)** or **mimerge(ipos)** while the motor is stopped at zero speed will be not possible to reach the target position, the move command will stay active till an other trajectory generator will be invoked.

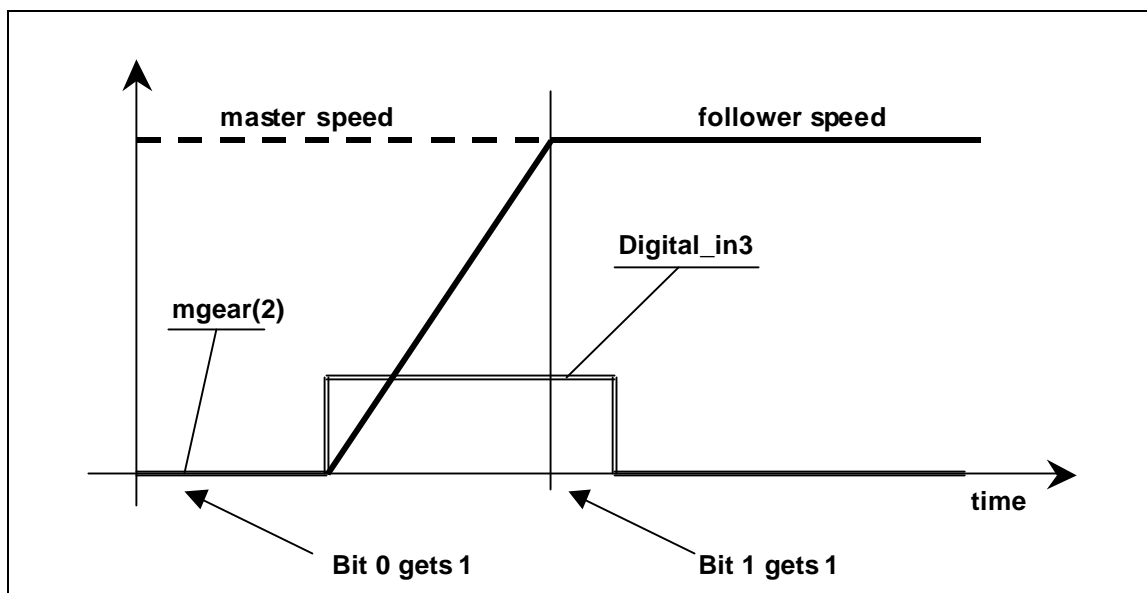


The figure is showing the difference between the functions merge and move for example to release a cam profile reaching the position 100000.

## Mgear(type)

The **mgear(type)** function generates a position trajectory for the position loop in such a way that the controlled motor accurately follows the master encoder selected on the input, the master/follower speed ratio can be programmed via the system variables *gr\_ration%* and *gr\_ratioid%*, they are respectively the numerator and denominator of the intended ratio. Different engage modes are available and which is used can be selected through the *type* parameter

If **type** is 2 will be selected the hardware engage with no phase recovering. After invoking **mgear(2)** no action will be taken till a positive edge on Digital input 3 is detected. The motor will start to accelerate according to the acceleration limit in *gr\_acc%*, when the motor will get the master speed the controlled axis will follow the master unit according to the gear ratio selected. The bit 0 of *gr\_flag%* gets 1 when the motor is performing the engage procedure; Bit 1 gets 1 when the master and the follower axes are digitally locked.

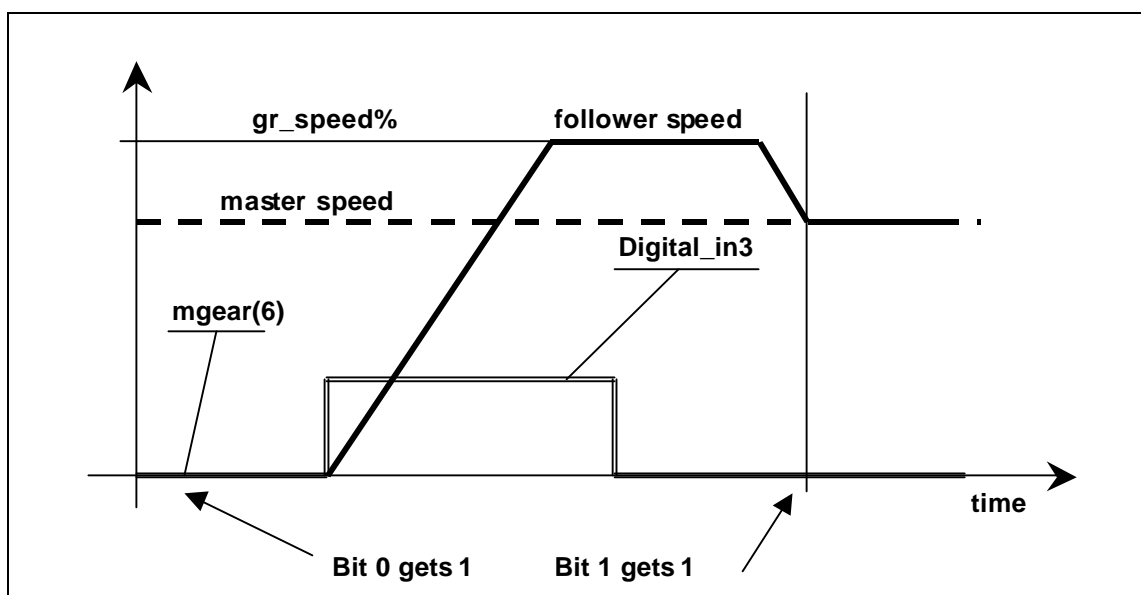


With **type=1** the **mgear()** function works as the function with *type = 2* but the motor will start accelerating to reach the master speed immediately regardless of the status of *digital input3*.

With **type=3** the **mgear()** function works as the function with *type = 2* but the motor will wait a negative edge of *digital input3*.

With **type=4** the **mgear()** function works as the function with *type = 2* but the motor will wait the first edge of *digital input3*.

If **type** is 6 will be selected the hardware engage with phase recovering. After invoking **mgear(6)** no action will be taken till a positive edge on Digital input 3 is detected. The motor will start to accelerate according to the acceleration limit in *gr\_acc%*, when the motor will get the master speed, in order to recover the space lost during the acceleration, a recovering trajectory will be generated in accordance with both *gr\_acc%* and *gr\_speed%*, then the controlled axis will follow the master unit according to the gear ratio selected. The bit 0 of *gr\_flag%* gets 1 when the motor is performing the engage procedure; Bit 1 gets 1 when the master and the follower axes are digitally locked.



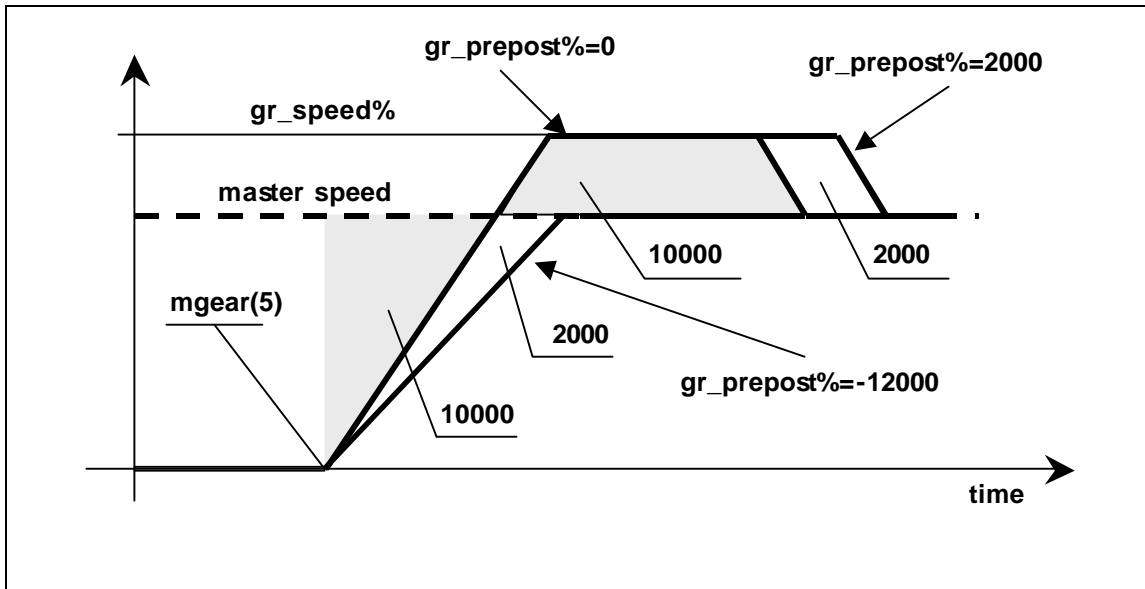
With **type=5** the **mgear()** function works as the function with **type = 6** but the motor will start accelerating to reach the master speed immediately regardless of the status of *digital input3*.

With **type=7** the **mgear()** function works as the function with **type = 6** but the motor will wait a negative edge of *digital input3*.

With **type=8** the **mgear()** function works as the function with **type = 6** but the motor will wait the first edge of *digital input3*. If the engage methods 5 thru 8 are used then become active the pre-post triggering variable *gr\_prepost%*.

Usually, the interlock phase is divided into two actions; in the first action the follower-axis accelerates in order to reach the master axis speed, loosing space; in the second action the follower exceeds the master axis speed in order to recover space lost during acceleration. In some applications, it is necessary that the space recovered and the space lost does not coincide. To do this, the *gr\_prepost%* variable will add to the space to be recovered.

If the space which the axis is going to lose during the acceleration is less of the pre-trigger space (P.I. space which is going to lose 10000,  $gr\_prepost\%=-12000$ ), automatically the trajectory generator will use a lower acceleration rate in order to get the master speed with no space to recover so no overshoot will be generated.



The  $gr\_prepost\%$  variable is used mainly to solve two typical needs:

If you set  $gr\_acc\%$  with high acceleration rate (over the physical limit) the engage action will take always the space in  $gr\_prepost\%$  and no overshoot will be got.

It is possible to use  $gr\_prepost\%$  to adjust via software the position of the engage sensor.



Pay attention of the sign of the variable  $gr\_prepost\%$ , if the motor will run clockwise a positive value means post trigger, a negative value pre trigger. If the motor will run counter clockwise it is the opposite.



To release the ***mgear(type)*** function an other trajectory generator has to be invoked as for example ***mmove(pos)***, ***mspeed(vel)*** or ***midle()***.



Any change of the system variable  $gr\_...\%$  will take its effect on fly.

## **mcam(table)**

The **mcam(table)** function provides a tracking function where the motion law of the follower-axis is described through points in a specific table. Each point describing the *follower = f ( master )* function is described by setting only the Y coordinate of the Cartesian coordinate couple in the table; the X coordinates are considered at steady X intervals (equidistant).

The **mcam(table)** function between two x,y coordinates will perform a linear interpolation in order to have the trajectory smoother.

The engage procedure will be done as follow:

The motor accelerates or decelerates as necessary to the speed required by the present motion of the master and the slave position profile defined in the cam table using an implicit **mspeed(vel%)**.

Then, using an implicit **mphase(cm\_camout%)** function the motor gets the position at the output of the camming generator and then it will be directly connected to the output at the camming generator to drive the position loop.

Using the function **mcamhard(table)** the engage procedure will be skipped and the output of the camming generator will be connected to the position loop.

The system variables related to the cam generator are:

**Cm\_camout%** It is the position demand at the output of the trajectory generator, its value is always inside the slave modulus (to refer to the general system variables).

**Cm\_num%** It is the number of point of the table describing the camming profile. Pay attention to set the right number to avoid undesired effects.

**Cm\_index%** It is a read only variable and is reporting the last point in the table used for the calculation of the follower-axis position. For example if the table is 360 points long, **cm\_index%** is the master position in degree.

**Cm\_flag%** The Bit0 gets 1 after invoking **mcamend()**, Bit0 indicates the trajectory generator is going to release the camming function at the end of the present modulus.



To release the **mcam(table)** function an other trajectory generator has to be invoked as for example **mmove(pos)**, **mspeed(vel)** or **midle()**; or it is possible release the **mcam(table)** at the end of the present modulus invoking **mcamend()**.



Any change of the system variable **cm\_...%** will take its effect on fly.



Invoking a ***mcam(table)*** function while is active the trajectory generator, using a different table, cause the on fly change of table used. The application software must be written to handle the change avoiding undesired effects.

## mspline(xy%,np%,camtab%)

The function perform a cubic spline interpolation among **np%** pair points describing the motion profile through cartesian coordinates, in output the function fill a 360 element array to use with the motion function **mcam(camtab%)** In the specific table **XY%**. each point X represents the master position and the Y point represents the follower position.

Filling the XY% table you have to pay attention that:

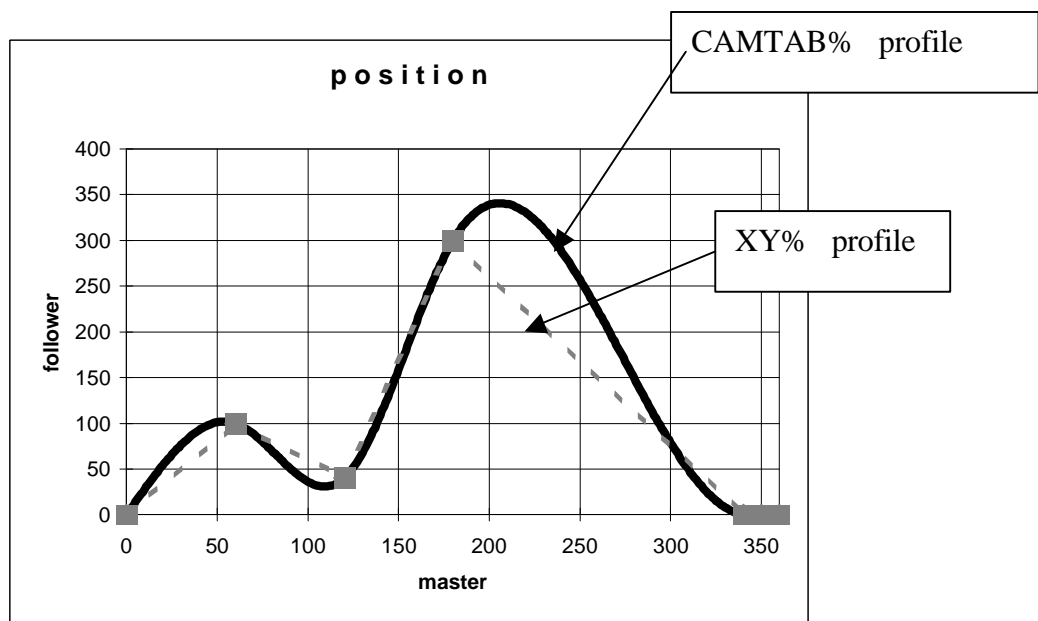
- The X's value must be increasing
- The point XY%(0,0) must be 0
- The point XY%(0,1) must be 0
- The number of pair point must be greater of 7

For example if XY% is:

XY%(0,0) = 0	XY%(0,1) = 0
XY%(1,0) = 0	XY%(1,1) = 0
XY%(2,0) = 60	XY%(2,1) = 100
XY%(3,0) = 120	XY%(3,1) = 40
XY%(4,0) = 180	XY%(4,1) = 300
XY%(5,0) = 340	XY%(5,1) = 0
XY%(6,0) = 350	XY%(6,1) = 0
XY%(7,0) = 360	XY%(7,1) = 0

Invoking the function **mspline(XY%,7,camtab%)**

You will get a camtab% profile better described in the graph below



## Mvirtualax(pos)

The **mvirtualax(pos)** function perform the generation of a move trajectory in order to reach the position pos, but the position demand at the output of the trajectory generator will not drive the position loop, it will be added to the master position coming from the selected encoder.

The move trajectory will be generated in accordance to the limit programmed in vx\_acc% vx\_dec% and vx\_speed%.

Vx\_pos% is a read only variable reporting the position at the output of the virtual axis trajectory generator.

**vx\_flag%** is a general utility flags variable. **Bit 0** is 1 when the target position is reached therefore no moving operation is active.

If **Bit 2** = 1 the selected encoder will not be added to get the master position.

If you need the generation of a virtual axis only, pay attention to set to 1 Bit2 of vx\_flag%.



## Mvspeed(vel)

When the function **mvspeed(vel)** is invoked a speed trajectory will be generated and added to the master position coming from the selected encoder.

The acceleration / deceleration limits are defined via the system variable vs\_acc%.

The variable **vs\_flag%** has in general utility flags. **Bit 0** is 1 when the speed mode does not contribute to the motor movement, otherwise it is 0. **Bit 2** is 1 when a reference is generated such that a speed equal to the reference and no limits to the active acceleration are required.

The actual speed demand can be read in the variable

**vs\_speed%**

It is also possible use the function **mvirtualax(pos)** and **mvspeed(vel)** to adjust the master position (re-phasing).

## Mphase(pos)

The **mphase(pos)** function perform the generation of a move trajectory in order to reach the position pos but the position demand at the output of the trajectory generator will be added to the active trajectory generator before to drive the position loop.

The move trajectory will be generated according to the limit programmed in ph\_acc% ph\_dec% and ph\_speed%.

**ph\_flag%** is a general utility flags variable. **Bit 0** is 1 when the target position is reached therefore no moving operation is active.

The function **phase(pos)** is normally used to adjust the follower position (re-phasing).

**Ph\_position%** return the actual phasing value.

## Midle()

The function **midle** abort the trajectory generator active and drive the motor at zero speed using an implicit invoking of speed(0). Reached zero speed the position demand variable is no more update so the axis stays at zero speed locked in position.

## Mhome(cmd)

The **mhome(cmd)** function perform one of the 8 homing procedure available in the SERVOSTAR 600.

**Cmd** equal 1 start the homing procedure, **cmd** equal 2 abort the homing procedure.

The homing will be executed only if the motion planner is in idle. The variable Hm\_flag% show the status of the homing, BIT0 get 1 when homing is in progress, BIT1 get 1 when homing is normally terminate.

The type of homing will be selected thru the Servostar variable NREF, the type available are the following:

- |          |  |
|----------|--|
| Homing 0 | Sets the reference point to the setpoint position (the following error is lost).   |
| Homing 1 | Traverse to the reference switch with zero-mark recognition.   |
| Homing 2 | Move to hardware limit switch, with zero-mark recognition. The reference point is set to the first zero-crossing of the feedback unit beyond the limit switch. |
| Homing 3 | Move to reference switch, without zero-mark recognition. The reference point is set to the transition of the reference switch.                                 |
| Homing 4 | Move to hardware limit switch, without zero-mark recognition. The reference point is set to the transition of the reference switch.                            |
| Homing 5 | Move to the next zero-mark of the feedback unit. The reference point is set to the next zero-mark of the feedback unit.  |
| Homing 6 | Sets the reference point to the actual position (the following error is lost).   |
| Homing 7 | Move to mechanical stop, with zero-mark recognition. The reference point is set to the first zero-crossing of the feedback unit beyond the limit switch.       |
| Homing 8 | Drives to an absolute SSI position. At the start of the homing run, a position is read in from the SSI input and then used as the target position.             |



The repetition accuracy of the homing procedure that are made without zero-point recognition depends of the traversing speed and the mechanical design of the reference switch or limit switch.



-For a better understanding about the different type of homing procedures work, please refer to **SERVOSTAR 600** setup software manual.

The Servostar variables relatives to the homing procedure are:

VREF	is the traverse speed.
ACCR	is the acceleration limit.
DECR	is the deceleration limit.
REFIP	is the torque limit when the homing is in progress.
DREF	is the direction of the homing.
ROFFS	position offset.

# The Interrupt functions

In order to provide a faster reaction to the external events some interrupt functions has been provided.

The events able to generate the interrupt are:

Event	Source
MC-600 digital input 0	0
MC-600 digital input 1	1
MC-600 digital input 2	2
MC-600 digital input 3	3
MC-600 digital input 4	4
MC-600 digital input 5	5
MC-600 digital input 6	6
MC-600 digital input 7	7
Gv_mflag% bit 0	8
Gv_mflag% bit 1	9
Gv_mflag% bit 2	10
Gv_mflag% bit 3	11
Gv_mflag% bit 4	12
Gv_status% bit 3	13
Gv_status% bit 7	14
Gv_status% bit 8	15
Gv_status% bit 10	16
Timer interrupt	17
CAM Index Interrupt	18
Reserved	19
Reserved	20
Reserved	21
Gv_compare% bit 0	22
Gv_compare% bit 1	23
Gv_compare% bit 2	24
Gv_compare% bit 3	25
Gv_compare% bit 4	26
Gv_compare% bit 5	27
Gv_compare% bit 6	28
Gv_compare% bit 7	29
Input capture on input 2	30
Input capture on input 3	31

If a the programmed transition will be detected the normal program execution will be interrupted, the interrupt source will be disabled and the programmed subroutine will be called.

**Intdisable()**            The Intdisable function disable the interrupt

**Intenable()**            The Intenable function enable the interrupt

### Intsetup(Line%, Source%, Edge%)

The **Intsetup()** function program the interrupt vector. **Line%** define the starting line of the interrupt handling subroutine.

**Source%** select the interrupt source to program.

**Edge%** define the edge of the transition:

- 0 Disable the source
- 1 On 0→1 transition
- 2 On 1→0 transition
- 3 On both transitions

### Reti

The **Reti** statement is used to exit from an interrupt subroutine. The **Reti** restarts the rest of your program at the next executable statement following the interruption and re-enable the interrupt.

### Mquotecmp(Source, Dest, On, Off)

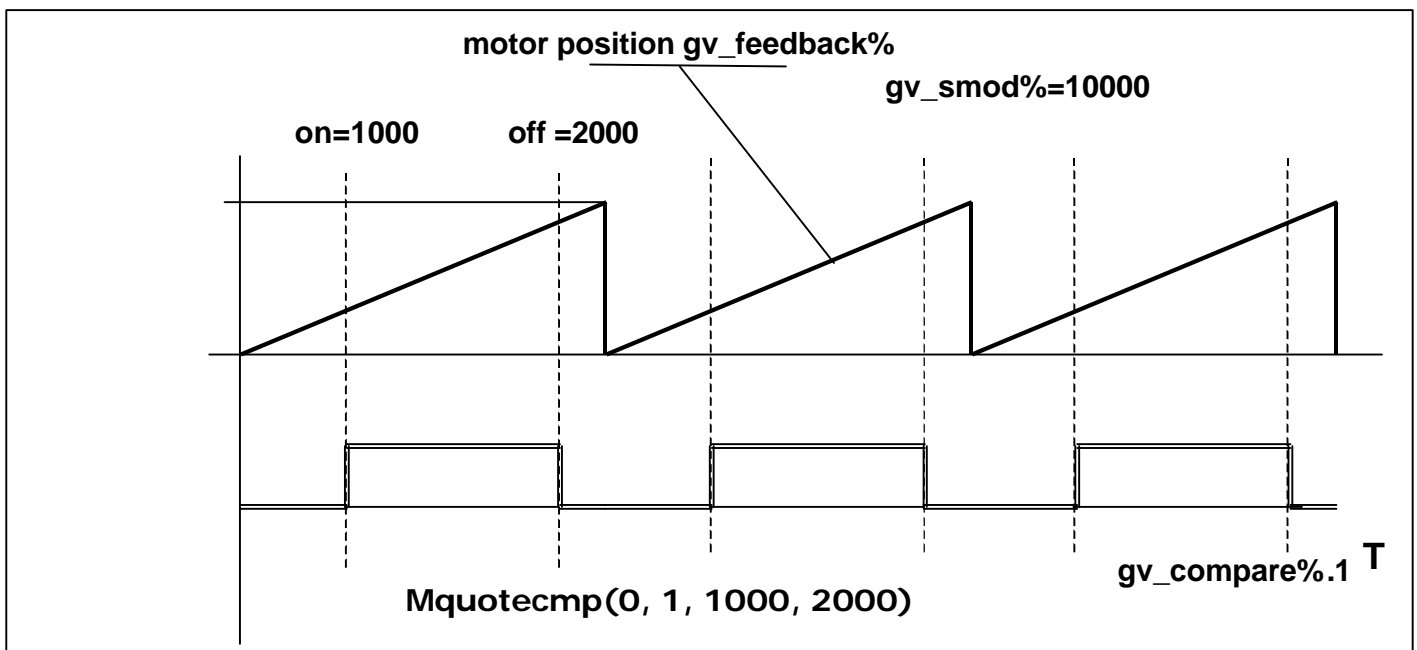
The **Mquotecmp** function set the threshold for the quote comparator.

**Source** selects the variable to be compared:

- 0 gv\_feedback%
- 1 gv\_master%
- 2 cm\_master%

**Dest** selects the number of comparator (0 to 7)

**On** is the first valid position



**Off** is the last valid position

# The Drive related functions

**Mclrfault()** The **mclrfault** function reset the drive fault.

**Mdrven()** The **mdrven** function enable the drive.

**Mdrvdis()** The **mdrvdis** function disable the drive.

**Sdoread(idx)** The **sdoread(idx)** function return the value of the drive parameter with object number idx.

**Sdowrite(idx, Data)** The **sdowrite(idx,data)** function write the value data into the parameter with object number idx.

**Sdoerror()** The **sdoerror()** function return the error code of the last SDO operation.

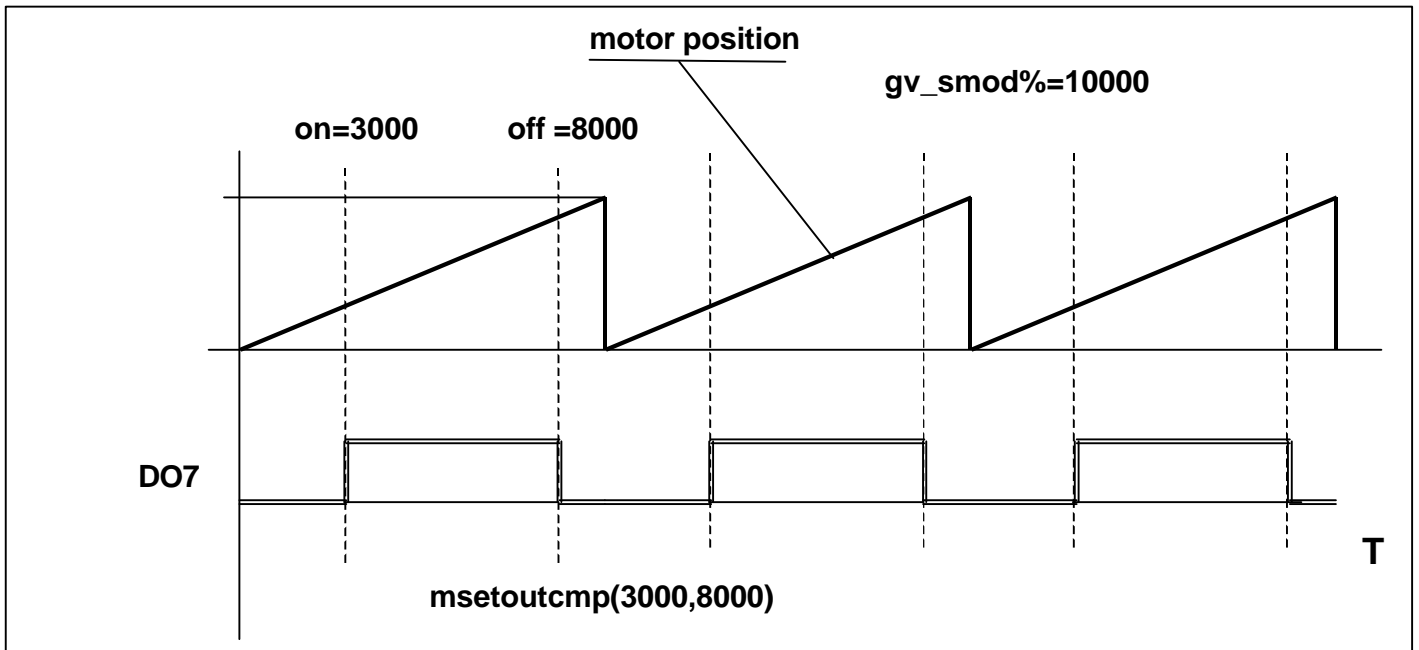
Error code for PLD read/write SDO	
Code	Description
&H00000001	Invalid access on used SDO channel
&H00000002	Drive timeout by SDO reading
&H00000003	Drive internal SDO access error
&H00000004	Drive SDO receipt missing
&H00000005	DPRSTATE HS error
&H00000007	Invalid SDO number
&H00000009	Invalid control code request
&H0000000A	Invalid write SDO index
&H0000000B	Invalid write access (read only)
&H0000000C	Invalid write data
&H0000000D	Invalid drive state (Enable / Opmode)
&H0000000E	Invalid write data < minimum
&H0000000F	Invalid write data > maximum
&H00000010	Invalid write data wrong parameter length

# The I/O functions

## msetoutcmp (on,off)

The **msetoutcmp (on,off)** function enable the MC-600 to drive automatically the digital output 7.

The output *do* will be 1 if the motor position is between *on* and *off* positions.



## Dimension capture

### Msetincap(in, edge, time)

The **Msetincap** function enable the capture of the position on a transition of digital input 2 (**in** = 0) and input 3 (**in** = 1).

The parameter **edge** select the active edge of the input:

- 0 Disable the capture
- 1 On 0→1 transition
- 2 On 1→0 transition
- 3 On both transitions

The parameter **time** is a correction value related to the reaction delay of the sensor (unit 0.8 microseconds).

The input 2 is able to capture the feedback position and return the latched position in the system variable **gv\_capposf%**.

The input 3 is able to capture a programmable position and return the latched position in the system variable **gv\_cappos%**.

The variable **gv\_capsel%** selects the source of the master position as follow:

- 0 master position **gv\_master%**
- 1 cam master position **cm\_master%**
- 2 motor position **sl\_posdemand%**

### **Outw(value)**

The **outw** function write the digital output of the MC-600. A single output can be set or reset thru the variable **gv\_outx%**.

### **Outsta()**

The **outsta** function return the status of the digital output of the MC-600.

The status of a single output can be read thru the variable **gv\_outx%**.

### **Insta()**

The **insta** function return the status of the digital input of the MC-600.

The status of a single input can be read thru the variable **gv\_inx%**.

## The other system variables of the “Motion planner”

Gv_master%	It is the master position. It is the value of the selected encoder added of the value coming from the virtualax trajectory generator
Gv_feedback%	It is the actual position of the controlled axis
Gv_encoder%	It is the value of the selected encoder
Gv_encspeed%	It is the speed value of the selected encoder
Gv_inx% (x=0..12)	They are the status of the digital inputs, from 0 to 7 they are located on the MC-600, from 8 to 12 are located on the <b>SERVOSTAR</b> 600.
Gv_outx% (x=0..9)	They are the status of the digital output, from 0 to 7 they are located on the MC-600, the outputs 8 and 9 are located on the <b>SERVOSTAR</b> 600.
Gv_mmod%	Is the master modulus. If its value is 0 means no modulus correction will be done on the master position. If for example gv_mmod%=4096 the range of the master position will be from 0 to 4095.
Gv_smod%	Is the modulus of the controlled axis. If its value is 0 means no modulus correction will be done on the feedback unit. If for example gv_smod%=4096 the range of the feedback position will be from 0 to 4095.
Sl_posdemand%	Is the reference of the position loop
Sl_spddemand%	Is the speed of the position reference.
Gv_status%	<ul style="list-style-type: none"> <li>Bit 0 – ready for switch on</li> <li>Bit 1 – switched on</li> <li>Bit 2 – operation enabled</li> <li>Bit 3 - error</li> <li>Bit 4 - reserved</li> <li>Bit 5 – not in emergency stop</li> <li>Bit 6 – switch on inhibit</li> <li>Bit 7 - warning</li> <li>Bit 8 – following error</li> <li>Bit 9 - reserved</li> <li>Bit 10 – idle</li> <li>Bit 11 – speed active</li> <li>Bit 12 – gear active</li> <li>Bit 13 – move active</li> <li>Bit 14 – cam active</li> <li>Bit 15 – home active</li> </ul>
Gv_command%	<ul style="list-style-type: none"> <li>Bit 0 – enable</li> <li>Bit 1 – emergency stop and disable (1→0)</li> <li>Bit 2 – emergency stop (1→0)</li> <li>Bit 3 – reference enable</li> </ul>
Gv_error%	Drive error code (See drive documentation ERRCODE)
Gv_warning%	Drive warning code (See drive documentation STATCODE)

Gv_anin1%	Value of drive analog input 1
Gv_anin2%	Value of drive analog input 2
Gv_trjstat%	Drive status information (See drive documentation TRJSTAT)
Gv_latch1%	Drive dimensione capture 1 (positive edge)
Gv_latch1n%	Drive dimensione capture 1 (negative edge)
Gv_latch2%	Drive dimensione capture 2 (positive edge)
Gv_latch2n%	Drive dimensione capture 2 (negative edge)
Gv_monitor1%	Setting of drive analog output 1
Gv_monitor2%	Setting of drive analog output 2
Gv_usvar1%	Setting of drive DPRVAR1 (CANOpen object 36A6h)
Gv_usvar2%	Setting of drive DPRVAR2 (CANOpen object 36A7h)
Gv_usvar3%	Setting of drive DPRVAR3 (CANOpen object 36A8h)
Gv_usvar4%	Setting of drive DPRVAR4 (CANOpen object 36A9h)
Gv_usvar5%	Setting of drive DPRVAR5 (CANOpen object 36AAh)
Gv_usvar6%	Setting of drive DPRVAR6 (CANOpen object 36ABh)
Gv_usvar7%	Setting of drive DPRVAR7 (CANOpen object 36ACh)
Gv_usvar8%	Setting of drive DPRVAR8 (CANOpen object 36ADh)
Gv_usvar9%	Value of drive DPRVAR9 (CANOpen object 36AEh)
Gv_usvar10%	Value of drive DPRVAR10 (CANOpen object 36AFh)
Gv_usvar11%	Value of drive DPRVAR11 (CANOpen object 36B0h)
Gv_usvar12%	Value of drive DPRVAR12 (CANOpen object 36B1h)
Gv_usvar13%	Value of drive DPRVAR13 (CANOpen object 36B2h)
Gv_usvar14%	Value of drive DPRVAR14 (CANOpen object 36B3h)
Gv_usvar15%	Value of drive DPRVAR15 (CANOpen object 36B4h)
Gv_usvar16%	Value of drive DPRVAR16 (CANOpen object 36B5h)
Gv_velmot%	Real motor speed value (1rpm = 33554432/240000 counts)
Gv_curmot%	Torque demand value (Drive peak current = 3280)
Gv_mflag%	Bit 0 – Final position reached (Position mode) Bit 1 – Follower axis locked to the master (Gear mode) Bit 2 – At speed (Speed mode) Bit 3 – Cam locked (Cam mode) Bit 4 – Homing done (Home mode)

# The "MotionBasic"

## Introduction

This chapter is designed as a REFERENCE GUIDE, it is not designed to teach the BASIC programming language. There is, however, an extensive glossary of terms and a "semi-tutorial" approach to many of the sections in the book. If you don't already have a working knowledge of BASIC and how to use it to program, we suggest that you study the BASIC language using other books available on the market.

## The operating system

The Operating System is contained in firmware of the MC-600 and is a combination of six separate, but interrelated, program modules.

- 1) The BASIC Interpreter
- 2) The Filesystem
- 3) The KERNEL
- 4) The Screen Editor
- 5) The Motion planner
- 6) The Field buses activity

- 1) The BASIC Interpreter is responsible for analyzing BASIC statement syntax and for performing the required calculations and/or data manipulation. The BASIC Interpreter has a vocabulary of 65 "keywords" which have special meanings. The upper and lower case alphabet and the digits 0-9 are used to make both keywords and variable names. Certain punctuation characters and special symbols also have meanings for the Interpreter. Table Table 0-1. BASIC Character Set lists the special characters and their uses.
- 2) The Filesystem handles the storage of user's programs and data; it is organized into two solid state disk units, named A and B, residing in RAM and in FLASH memory. A is a RAM disk, and its data is lost upon system power off, while B is a FLASH disk and its data is retained in nonvolatile storage. Because of this different nature, disk units have different usage capabilities, which will be explained in next section.
- 3) The KERNEL handles most of the interrupt level processing in the system. The KERNEL also does the actual input and output of data.
- 4) The Screen Editor controls the output to the terminal screen and the editing of BASIC program text. In addition, the

Screen Editor intercepts keyboard input so that it can decide whether the characters put in should be acted upon immediately, or passed on to the BASIC Interpreter.

- 5) The Motion planner module has been already described.
- 6) The Field buses activity will be described in the next section of this user's manual.

# Basic programming rules

Table 0-1. BASIC Character Set

CHARACTER	NAME and DESCRIPTION
	BLANK - separates keywords and variable names
;	SEMI-COLON - used in variable lists to format output
=	EQUAL SIGN - value assignment and relationship testing
+	PLUS SIGN - arithmetic addition or string concatenation concatenation: linking together in a chain)
-	MINUS SIGN - arithmetic subtraction, unary minus
*	ASTERISK - arithmetic multiplication
/	SLASH - arithmetic division
^	UP ARROW - arithmetic exponentiation
(	LEFT PARENTHESIS - expression evaluation and functions
)	RIGHT PARENTHESIS - expression evaluation and functions
%	PERCENT - declares variable name as an integer
#	NUMBER - comes before logical file number in input/ output statements
\$	DOLLAR SIGN - declares variable name as a string
,	COMMA - used in variable lists to format output; also Separates command parameters
.	PERIOD - decimal point in floating point constants
"	QUOTATION MARK - encloses string constants
:	COLON - separates multiple BASIC statements in a line
?	QUESTION MARK - abbreviation for the keyword PRINT
<	LESS THAN - used in relationship tests
>	GREATER THAN - used in relationship tests
{pi}	the numeric constant 3.141592654

The Operating System gives you two modes of BASIC operation:

- 1) DIRECT Mode
- 2) PROGRAM Mode

1) When you're using the DIRECT mode, BASIC statements don't have line numbers in front of the statement. They are executed whenever the <RETURN> key is pressed.

2) The PROGRAM mode is the one you use for running programs.

When using the PROGRAM mode, all of your BASIC statements must have line numbers in front of them. You can have more than one BASIC statement in a line of your program, but the number of statements is limited by the fact that you can only put 80 characters on a logical screen line. This means that if you are going to go over the 80 character limit you have to put the

entire BASIC statement that doesn't fit on a new line with a new line number.



Always type NEW and hit <RETURN> before starting a new program.

## Programming numbers and variables

### *Integer, floating point and string constants*

Constants are the data values that you put in your BASIC statements. BASIC uses these values to represent data during statement execution. The MC-600 BASIC can recognize and manipulate three types of constants:

- 1) INTEGER NUMBERS
- 2) FLOATING-POINT NUMBERS
- 3) STRINGS

***Integer constants*** are whole numbers (numbers without decimal points).

Integer constants have 32 bits format. Integer constants do not have decimal points or commas between digits. If the plus (+) sign is left out, the constant is assumed to be a positive number. Zeros coming before a constant are ignored and shouldn't be used since they waste memory and slow down your program. However, they won't cause an error. Integers are stored in memory as four-byte binary numbers. Some examples of integer constants are:

-12  
8765  
-3276878  
+44  
0  
-3276767

**Floating-point constants** are positive or negative numbers and can contain fractions. Fractional parts of a number may be shown using a decimal point. Once again remember that commas are NOT used between numbers. If the plus sign (+) is left off the front of a number, the MC-600 assumes that the number is positive. If you leave off the decimal point the computer will assume that it follows the last digit of the number. And as with integers, zeros that come before a constant are ignored. Floating-point constants can be used in two ways:

- 1) SIMPLE NUMBER
- 2) SCIENTIFIC NOTATION

Floating-point constants will show you up to nine digits on your screen. These digits can represent values between -999999999. and +999999999. If you enter more than nine digits the number will be rounded based on the tenth digit. If the tenth digit is greater than or equal to 5 the number will be rounded upward. Less than 5 the number will be rounded downward. This could be important to the final totals of some numbers you may want to work with.

Floating-point numbers are stored (using eight bytes of memory) and are manipulated in calculations with ten places of accuracy. However, the numbers are rounded to nine digits when results are printed. Some examples of simple floating-point numbers are:

1.23	.7777777
-.998877	-333.
+3.1459	.01

Numbers smaller than .01 or larger than 999999999. will be printed in scientific notation. In scientific notation a floating-point constant is made up of three parts:

- 1) THE MANTISSA
- 2) THE LETTER E
- 3) THE EXPONENT

The mantissa is a simple floating-point number. The letter E is used to tell you that you're seeing the number in exponential form. In other words E represents  $\times 10$  (eg.,  $3E3 = 3 \times 10^3 = 3000$ ). And the exponent is what multiplication power of 10 the number is raised to.

Both the mantissa and the exponent are signed (+ or -) numbers. The exponent's range is from -39 to +38 and it indicates the number of places that the actual decimal point in the mantissa would be moved to the left (-) or right (+) if the value of the constant were represented as a simple number.

There is a limit to the size of floating-point numbers that BASIC can handle, even in scientific notation: the largest number is +1.70141183E+38 and calculations which would result in a larger number will display the BASIC error message ?OVERFLOW ERROR. The smallest floating-point number is +2.93873588E-39 and calculations which result in a smaller value give you zero as an answer and NO error message. Some examples of floating-point numbers in scientific notation (and their decimal values) are:

235.988E-3	(.235988)
2359E6	(2359000000.)
-7.09E-12	(-.000000000000709)
-3.14159E+5	(-314159.)

**String constants** are groups of alphanumeric information like letters, numbers and symbols. When you enter a string from the keyboard, it can have any length up to the space available in an 80-character line (that is, any character spaces NOT taken up by the line number and other required parts of the statement). A string constant can contain blanks, letters, numbers, punctuation in any combination. You can even put commas between numbers. The only character which cannot be included in a string is the double quote mark ("). This is because the double quote mark is used to define the beginning and end of the string.

A string can also have a null value-which means that it can contain no character data. You can leave the ending quote mark off of a string if it's the last item on a line or if it's followed by a colon (:). Some examples of string constants are:

""	( a null string)
"HELLO"	
"\$25,000.00"	
"NUMBER OF EMPLOYEES"	

## ***Integer floating point and string variables***

Variables are names that represent data values used in your BASIC statements. The value represented by a variable can be assigned by setting it equal to a constant, or it can be the result of calculations in the program. Variable data, like constants, can be integers, floating point numbers, or strings. If you refer to a variable name in a program before a value has been assigned, the BASIC Interpreter will automatically create the variable with a value of zero if it's an integer or floating-point number. Or it will create a variable with a null value if you're using strings. Variable names can be a max length of 20 characters.

Variable names may NOT be the same as BASIC keywords and they may NOT contain keywords in the middle of variable names. Keywords include all BASIC commands, statements, function names and logical operator names. If you accidentally use a keyword in the middle of a variable name, the BASIC error message ?SYNTAX ERROR will show up on your screen.

The characters used to form variable names are the alphabet and the numbers 0-9. The first character of the name must be a letter. Data type declaration characters (%) and (\$) can be used as the last character of the name. The percent sign declares the variable to be an integer and the dollar sign (\$) declares a string variable. If no type declaration character is used the Interpreter will assume that the variable is a floating-point. Some examples of variable names, value assignments and data types are:

A\$="DRIVE ENABLED"	(string variable)
MTH\$="JAN"+A\$	(string variable)
K%=5	(integer variable)
CNT%=CNT%+1	(integer variable)
FP=12.5	(floating-point variable)
SUM=FP*CNT%	(floating-point variable)

## ***Integer, floating point and string arrays***

An array is a table (or list) of associated data items referred to by a single variable name. In other words, an array is a sequence of related variables. A table of numbers can be seen as an array, for example. The individual numbers within the table become "elements" of the array. Arrays are a useful shorthand way of describing a large number of related variables. Take a table of numbers for instance. Let's say that the table has 10 rows of numbers with 20 numbers in each row. That makes total of 200 numbers in the table. Without a single array name to call on, you would have to assign a unique name to each value in the table. But, because you can use arrays, you

only need one name for the array and all the elements in the array are identified by their individual locations within the array. Array names can be integers, floating-points or string data types and all elements in the array have the same data type as the array name. Arrays can have a single dimension (as in a simple list) or they can have multiple dimensions. Each element of an array is uniquely identified and referred to by a subscript (or index variable) following the array name, enclosed within parentheses ( ).

The maximum number of dimensions an array can have is 4 and the number of elements in each dimension is limited to 32767. If an array has only one dimension and its subscript value will never exceed 10 (11 items: 0 thru 10) then the array will be created by the Interpreter and filled with zeros (or nulls if string type) the first time any element of the array is referred to, otherwise the BASIC DIM statement must be used to define the shape and size of the array.

Value assignments and data types are:

$A(5)=0$  (sets the 5th element in the 1 dimensional array called "A" equal to 0)

$B(5,6)=0$  (sets the element in row position 5 and column position 6 in the 2 dimensional array called "B" equal to 0)

# ***Expressions and operators***

Expressions are formed using constants, variables and/or arrays. An expression can be a single constant, simple variable, or an array variable of any type. It can also be a combination of constants and variables with arithmetic, relational or logical operators designed to produce a single value. How operators work is explained below. Expressions can be separated into two classes:

- 1) ARITHMETIC
- 2) STRING

Expressions are normally thought of as having two or more data items called operands. Each operand is separated by a single operator to produce the desired result. This is usually done by assigning the value of the expression to a variable name. All of the examples of constants and variables that you've seen so far, were also examples of expressions.

An operator is a special symbol the BASIC Interpreter in your MC-600 motion platform recognizes as representing an operation to be performed on the variables or constant data. One or more operators, combined with one or more variables and/or constants, form an expression. Arithmetic, relational and logical operators are recognized by MC-600 BASIC.

## ***Arithmetic expressions***

Arithmetic expressions, when solved, will give an integer or floating point value. The arithmetic operators (+, -, \*, /, ^) are used to perform addition, subtraction, multiplication, division and exponentiation operations, respectively.

## ***Arithmetic operators***

An arithmetic operator defines an arithmetic operation which is performed on the two operands on either side of the operator. If one of the operands is floating point, the arithmetic operations are performed using floating-point numbers. If needed, integers are converted to floating-point numbers before an arithmetic operation is performed. The result is converted back to an integer if it is assigned to an integer variable name.

If all the operands are integer the arithmetic operation will be performed using 32 bits integer mathematics.

**ADDITION (+):** The plus sign (+) specifies that the operand on the right is added to the operand on the left.

EXAMPLES:

2+2  
A+B+C  
X%+1  
BR+10E-2

**SUBTRACTION (-):** The minus sign (-) specifies that the operand on the right is subtracted from the operand on the left.

EXAMPLES:

4-1  
100-64  
A-B  
55-142

The minus can also be used as a unary minus. That means that it is the minus sign in front of a negative number. This is equal to subtracting the number from zero (0).

EXAMPLES:

-5  
-9E4  
-B  
4-(-2) same as 4+2

**MULTIPLICATION (\*):** An asterisk (\*) specifies that the operand on the left is multiplied by the operand on the right.

EXAMPLES:

100\*2  
50\*0  
A\*X1  
R%\*14

**DIVISION (/):** The slash (/) specifies that the operand on the left is divided by the operand on the right.

EXAMPLES:

10/2  
6400/4  
A/B  
4E2/XR

**EXPONENTIATION (^):** The up arrow (^) specifies that the operand on the left is raised to the power specified by the operand on the right (the exponent). If the operand on the right is a 2, the number on the left is squared; if the exponent is a 3, the number on the left is cubed, etc. The exponent can be any number so long as the result of the operation gives a valid floating-point number.

EXAMPLES:

$2^2$       Equivalent to:  $2*2$   
 $3^3$       Equivalent to:  $3*3*3$   
 $4^4$       Equivalent to:  $4*4*4*4$   
 $AB^{CD}$   
 $3^{-2}$       Equivalent to:  $1/3*1/3$

## ***Relational operators***

The relational operators (<, =, >, <=, >=, <>) are primarily used to compare the values of two operands, but they also produce an arithmetic result. The relational operators and the logical operators (AND, OR, and NOT), when used in comparisons, actually produce an arithmetic true/false evaluation of an expression. If the relationship stated in the expression is true, the result is assigned an integer value of - 1 and, if it's false, a value of 0 is assigned. These are the relational operators:

<    LESS THAN  
=    EQUAL TO  
>    GREATER THAN  
<=    LESS THAN OR EQUAL TO  
>=    GREATER THAN OR EQUAL TO  
<>    NOT EQUAL TO

EXAMPLES:

$1 = 5 - 4$       result true (-1)  
 $14 > 66$       result false (0)  
 $15 > = 15$       result true (-1)

Relational operators can be used to compare strings. For comparison purposes, the letters of the alphabet have the order  $A < B < C < D$ , etc.

Strings are compared by evaluating the relationship between corresponding characters from left to right (see String Operations).

EXAMPLES:

"A" < "B"      result true (-1)  
"X" = "YY"      result false (0)  
BB\$ <> CC\$

Numeric data items can only be compared (or assigned) to other numeric items. The same is true when comparing strings, otherwise the BASIC error message ?TYPE MISMATCH will occur. Numeric operands are compared by first converting the values of either or both operands from integer to floating-point form, if

necessary. Then the relationship of the two values is evaluated to give a true/false result.

At the end of all comparisons, you get an integer no matter what data type the operand is (even if both are strings).

Because of this, a comparison of two operands can be used as an operand in performing calculations. The result will be - 1 or 0 and can be used as anything but a divisor, since division by zero is illegal.

## ***Logical operators***

The logical operators (AND, OR, NOT) can be used to modify the meanings of the relational operators or to produce an arithmetic result. Logical operators can produce results other than -1 and 0, though any nonzero result is considered true when testing for a true/false condition. The logical operators (sometimes called Boolean operators) can also be used to perform logic operations on individual binary digits (bits) in two operands. But when you're using the NOT operator, the operation is performed only on the single operand to the right. The operands must be in the integer range of values (floating-point numbers are converted to integers) and logical operations give an integer result.

Logical operations are performed bit-by-corresponding-bit on the two operands. The logical AND produces a bit result of 1 only if both operand bits are 1. The logical OR produces a bit result of 1 if either operand bit is 1. The logical NOT is the opposite value of each bit as a single operand. In other words, it's really saying, "If it's NOT 1 then it is 0. If it's NOT 0 then it is 1."

The exclusive OR (XOR) means that if the bits of two operands are equal then the result is 0 otherwise the result is 1. Logical operations are defined by groups of statements which, taken together, constitute a Boolean "truth table" as shown in Table 1-2.

Table 0-2. Boolean Truth Table

***The AND operation results in a 1 only if both bits are 1:***

1 AND 1 = 1  
0 AND 1 = 0  
1 AND 0 = 0  
0 AND 0 = 0

***The OR operation results in a 1 if either bit is 1:***

1 OR 1 = 1  
0 OR 1 = 1  
1 OR 0 = 1  
0 OR 0 = 0

***The NOT operation logically complements each bit:***

NOT 1 = 0  
NOT 0 = 1

***The exclusive OR (XOR) operation results in a 0 if both bits are equal***

1 XOR 1 = 0  
1 XOR 0 = 1  
0 XOR 1 = 1  
0 XOR 0 = 0

The logical operators AND, OR, XOR and NOT specify a Boolean arithmetic operation to be performed on the two operand expressions on either side of the operator. In the case of NOT, ONLY the operand on the RIGHT is considered. Logical operations (or Boolean arithmetic) aren't performed until all arithmetic and relational operations in an expression have been completed.

EXAMPLES:

IF A=100 AND B=100 THEN 10 (if both A and B have a value of 100 then the result is true)

A=96 AND 32: PRINT A (A = 32)

IF A=100 OR B=100 THEN 20	(if A or B is 100 then the result is true)
A=64 OR 32: PRINT A	(A = 96)
IF NOT X<Y THEN 30	(if X>=Y the result is true)
X= NOT 96	(result is -97 (two's complement))

## ***Hierarchy of operations***

All expressions perform the different types of operations according to a fixed hierarchy. In other words, certain operations are performed before other operations. The normal order of operations can be modified by enclosing two or more operands within parentheses ( ), creating a "subexpression." The parts of an expression enclosed in parentheses will be reduced to a single value before working on parts outside the parentheses.

When you use parentheses in expressions, they must be paired so that you always have an equal number of left and right parentheses. Otherwise the BASIC error message ?SYNTAX ERROR will appear.

Expressions which have operands inside parentheses may themselves be enclosed in parentheses, forming complex expressions of multiple levels. This is called nesting.

Parentheses can be nested in expressions to a maximum depth of ten levels-ten matching sets of parentheses.

The inner-most expression has its operations performed first. Some examples of expressions are:

```
A+B
C^(D+E)/2
((X-C^(D+E)/2)*10)+1
GG$>HH$
JJ$+"MORE"
K%=1 AND M<>X
K%=2 OR (A=B AND M<X)
NOT (D=E)
```

The BASIC Interpreter will normally perform operations on expressions by performing arithmetic operations first, then relational operations, and logical operations last. Both arithmetic and logical operators have an order of precedence (or hierarchy of operations) within themselves. On the other hand, relational operators do not have an order of precedence and will be performed as the expression is evaluated from left to right. If all remaining operators in an expression have the same level of precedence, then operations happen from left to right. When

performing operations on expressions within parentheses, the normal order of precedence is maintained. The hierarchy of arithmetic and logical operations is shown in Table 1-3 from first to last in order of precedence.

Table 1-3. Hierarchy of Operations Performed on Expressions

OPERATOR	DESCRIPTION	EXAMPLE
^	Exponentiation	BASE ^ EXP
-	Negation (Unary Minus)	-A
* /	Multiplication Division	AB * CD    EF / GH
+ -	Addition Subtraction	CNT + 2    JK - PQ
> = <	Relational Operations	A <= B
NOT	Logical NOT (Integer Two's Complement)	NOT K%
AND	Logical AND	JK AND 128
OR	Logical OR	PQ OR 15

## String operations

Strings are compared using the same relational operators (=, <>, <=, >=, <, >) that are used for comparing numbers. String comparisons are made by taking one character at a time (left-to-right) from each string and evaluating each character code position from the standard ASCII character set. If the character codes are the same, the characters are equal. If the character codes differ, the character with the lower code number is lower in the character set. The comparison stops when the end of either string is reached. All other things being equal, the shorter string is considered less than the longer string. Leading or trailing blanks ARE significant. Regardless of the data types, at the end of all comparisons you get an integer result. This is true even if both operands are strings. Because of this a comparison of two string operands can be used as an operand in performing calculations. The result will be - 1 or 0 (true or false) and can be used as anything but a divisor since division by zero is illegal.

## String expressions

Expressions are treated as if an implied "<>0" follows them. This means that if an expression is true then the next BASIC statements on the same program line are executed. If the expression is false the rest of the line is ignored and the next line in the program is executed. Just as with numbers, you can also perform operations on string variables. The only string arithmetic operator recognized by MC-600 BASIC is the plus sign (+) which is used to perform

concatenation of strings. When strings are concatenated, the string on the right of the plus sign is appended to the string on the left, forming a third string as a result.

The result can be printed immediately, used in a comparison, or assigned to a variable name. If a string data item is compared with (or set equal to) a numeric item, or vice-versa, the BASIC error message ?TYPE MISMATCH will occur. Some examples of string expressions and concatenation are:

```
10 A$="FILE": B$="NAME"
20 NAM$=A$+B$           (gives the string: FILENAME)
30 RES$="NEW "+A$+B$    (gives the string: NEW
                        FILENAME)
```

## ***Programming techniques***

### **DATA CONVERSIONS**

When necessary, the MC-600 BASIC Interpreter will convert a numeric data item from an integer to floating-point or viceversa, according to the following rules:

- All arithmetic and relational operations are performed in integer format whenever is possible, otherwise in floating point format. Integers are converted to floating-point form for evaluation of the expression, and the result is converted back to integer, if needed. Logical operations convert their operands to integers and return an integer result.
- If a numeric variable name of one type is set equal to a numeric data item of a different type, the number will be converted and stored as the data type declared in the variable name.
- When a floating-point value is converted to an integer, the fractional portion is truncated (eliminated) and the integer result is less than or equal to the floating-point value.

# The BASIC Filesystem

A FLASH/RAM Filesystem has been provided on the MC-600 board to allow the storage of user's programs and data.

Two virtual disk units are provided, named with alphabet letters, like into any DOS based system.

Disk unit A: is the RAM disk, while disk unit B: is the FLASH disk.

RAM disk A: is formatted during startup, at system power on, and provides storage for up to 512 Kbytes of information. Informations contained into RAM disk are LOST upon system shut down.

FLASH disk B: can be formatted at user's need, and provides nonvolatile storage for up to 768 Kbytes of information.

Informations are organized in files, like in DOS or any other common disk operating system.

File functions are provided in order to create, open, read, write, append, close and delete the files. A file COPY function allows copying of RAM resident files into nonvolatile FLASH storage.

The main difference between the two virtual disk units is that the FLASH disk does not offer an easy random access as the RAM disk, due to non immediate rewritability of data (in order to rewrite any byte into the flash disk, an entire 64K flash sector must be erased and rewritten), thus we decided to allow random read/write access only into RAM disk, which offers higher speed, and to provide FLASH disk only as a backup media, to copy and store the data once this has been definitely written and fixed.

## FILENAMES

BASIC filenames consist of a file name (maximum length 8 characters) followed by an extension (maximum length 3 characters). For example, BOOK.TXT, CONVERT.BAS, SECTION.12.

## WILDCARDS

In some special case (directory listing, file deletion), wildcard characters can be put into filenames to allow you to select the files that match a particular filename pattern.

These are the general rules for wildcard interpretation:

\* matches any number of characters

? matches one character

Like in DOS, filename and extension are treated separately. Wildcards can appear in either filename or extension, or both.

For example,

files \*.txt

lists all files with extension .txt, eg free.txt, big.txt, readme.txt

files chap\*.zip

lists all files that start with chap and have extension .zip, eg chapter1.zip, chapter2.zip chapsum.zip

files big\*.\*

lists all files that start with big and have any extension, eg. bignews.exe, bigstuff.txt, bignews.tst

erase yah???.bas

deletes all files whose name begins with 'yah' and is 6 chars long; and whose extension is bas.

# Description of BASIC keywords

## ABS

TYPE: Function-Numeric  
FORMAT: ABS(<expression>)

Action: Returns the absolute value of the number, which is its value without any signs. The absolute value of a negative number is that number multiplied by -1.

EXAMPLES of ABS Function:

```
10 X = ABS (Y)
10 PRINT ABS (X*J)
10 IF X = ABS (X) THEN PRINT "POSITIVE"
```

## AND

TYPE: Operator  
FORMAT: <expression> AND <expression>

Action: AND is used in Boolean operations to test bits. it is also used in operations to check the truth of both operands. In Boolean algebra, the result of an AND operation is 1 only if both numbers being ANDed are 1. The result is 0 if either or both is 0 (false).

EXAMPLES of 1-Bit AND operation:

0	1	0	1
AND 0	AND 0	AND 1	AND 1
-----	-----	-----	-----
0	0	0	1

The MC-600 performs the AND operation on numbers in the 32 bits range. Any fractional values are not used, and numbers beyond the range will cause an ?ILLEGAL QUANTITY error message. When converted to binary format, the range allowed yields 16 bits for each number. Corresponding bits are ANDed together, forming a 16-bit result in the same range.

EXAMPLES of 16-Bit AND Operation:

	17
	AND 194
	-----
	000000000010001
	AND 0000000011000010
	-----
(BINARY)	0000000000000000
	-----
(DECIMAL)	0

When evaluating a number for truth or falsehood, the computer assumes the number is true as long as its value isn't 0. When evaluating a comparison, it assigns a value of -1 if the result is true, while false has a value of 0. In binary format, -1 is all 1's and 0 is all 0's. Therefore, when ANDing true/false evaluations, the result will be true if any bits in the result are true.

EXAMPLES of Using AND with True/False Evaluations:

```
50 IF X=7 AND W=3 THEN GOTO 10: REM ONLY TRUE IF BOTH X=7 AND W=3 ARE TRUE
```

60 IF A AND Q=7 THEN GOTO 10: REM TRUE IF A IS NON-ZERO AND Q=7 IS TRUE

## ATN

TYPE: Function-Numeric  
FORMAT: ATN(<number>)

Action: This mathematical function returns the arctangent of the number. The result is the angle (in radians) whose tangent is the number given. The result is always in the range  $-\pi/2$  to  $+\pi/2$ .

EXAMPLES of ATN Function:

```
10 PRINT ATN(0)
20 X = ATN(J)*180/ {pi} : REM CONVERT TO DEGREES
```

## ASC

TYPE: Function-Numeric  
FORMAT: ASC(<string>)

Action: ASC will return a number from 0 to 255 which corresponds to the ASCII value of the first character in the string.

EXAMPLES OF ASC Function:

```
10 PRINT ASC("Z")
20 X = ASC("ZEBRA")
30 J = ASC(J$)
```

## AUTO

TYPE: Statement  
FORMAT: AUTO <starting line number> <, increment>

Action: AUTO statement allow automatic numbering of program lines. When no argument is entered, AUTO uses its default values (starting line number = 10, increment = 10)

EXAMPLES of AUTO Statement:

```
>AUTO 100, 25
```

Automatically generates line numbers, starting at line 100, increment of 25.

## CHR\$

TYPE: Function-String  
FORMAT: CHR\$ (<number>)

Action: This function converts a ASCII code to its character equivalent. The number must have a value between 0 and 255, or an ?ILLEGAL QUANTITY error message results.

EXAMPLES of CHR\$ Function:

```
10 PRINT CHR$(65) : REM 65 = UPPER CASE A
20 A$=CHR$(13) : REM 13 = RETURN KEY
50 A=ASC(A$) : A$ = CHR$(A) : REM CONV. TO ASCII CODE AND BACK
```

## FCLOSE

NOT IMPLEMENTED YET

TYPE: I/O Statement  
FORMAT: FCLOSE <filehandle%>

Action: This statement closes the file pointed by handle. The file handle is the handle returned when the file or device was OPENed (see OPEN statement)

EXAMPLES of CLOSE Statement:

10 CLOSE FHNDL1% (Closes the file pointed by handle in FHNDL1%)

## CONT

TYPE: Command  
FORMAT: CONT

Action: This command re-starts the execution of a program which was halted by a STOP or END statement or the <[ctrl]C> key being pressed. The program will re-start at the exact place from which it left off. While the program is stopped, the user can inspect or change any variables or look at the program. When debugging or examining a program, STOP statements can be placed at strategic locations to allow examination of variables and to check the flow of the program.

EXAMPLE of CONT Command:

```
10 PI=0:C=1
20 PI=PI+4/C-4/(C+2)
30 PRINT PI
40 C=C+4:GOTO 20
```

This program calculates the value of PI. RUN this program, and after a short while hit the <[ctrl]C> key. You will see the display:

```
>BREAK IN 20      NOTE: Might be different number.
>
```

Type the command PRINT C to see how far the MC-600 has gotten. Then use CONT to resume from where the MC-600 left off.

## COS

TYPE: Function  
FORMAT: COS (<number>)

Action: This mathematical function calculates the cosine of the number, where the number is an angle in radians.

EXAMPLES of COS Function:

```
10 PRINT COS(0)
20 X = COS(Y* {pi} /180) : REM CONVERT DEGREES TO RADIANS
```

## DATA

TYPE: Statement  
FORMAT: DATA <list of constants>

Action: DATA statements store information within a program. The program uses the information by means of the READ statement, which pulls successive constants from the DATA statements. The DATA statements don't have to be executed by the program, they only have to be present. Therefore, they are usually placed at the end of the program. All data statements in a program are treated as a continuous list. Data is READ from left to right, from the lowest numbered line to the highest. If the READ statement encounters data that doesn't fit the type requested (if it needs a number and finds a string) an error message occurs. Any characters can be included as data, but if certain ones are used the data item must be enclosed by quote marks (" "). These include punctuation like comma (,), colon (:), blank spaces, and shifted letters, graphics, and cursor control characters.

EXAMPLES of DATA Statement:

```
10 DATA 1,10,5,8
```

```
20 DATA JOHN,PAUL,GEORGE,RINGO
30 DATA "DEAR MARY, HOW ARE YOU, LOVE, BILL"
40 DATA -1.7E-9, 3.33
```

## DEL

TYPE: Statement

FORMAT: DEL <startlinenumber><-<endlinenumber>>  
[ <, startlinenumber><-<endlinenumber>>...]

Action: This statement deletes program lines whose line number is comprised into the interval given in the argument. More than one interval can be specified, each of them separated by commas.

EXAMPLES of DEL Statement:

```
>DEL 100
Deletes program line 100
>DEL 100-
Deletes program, starting at line 100, up to end
>DEL 100-200
Deletes program lines from 100 to 200
>DEL 100-200, 300
Deletes program lines from 100 to 200 and line 300
```

## DIM

TYPE: Statement

FORMAT: DIM <variable> ( <subscripts> ) [ <variable> ( <subscripts> )...]

Action: This statement defines an array or matrix of variables. This allows you to use the variable name with a subscript. The subscript points to the element being used. The lowest element number in an array is zero, and the highest is the number given in the DIM statement, which has a maximum of 32767. The DIM statement must be executed once and only once for each array. A REDIM'D ARRAY error occurs if this line is re-executed. Therefore, most programs perform all DIM operations at the very beginning. There may be 4 dimensions and 4 subscripts in an array, limited only by the amount of RAM memory which is available to hold the variables. The array may be made up of normal numeric variables, as shown above, or of strings or integer numbers. If the variables are other than normal numeric, use the \$ or % signs after the variable name to indicate string or integer variables. If an array referenced in a program was never DiMensioned, it is automatically dimensioned to 11 elements in each dimension used in the first reference.

EXAMPLES of DIM Statement:

```
10 DIM A(100)
20 DIM Z (5,7), Y(3,4,5)
30 DIM Y7%(Q)
40 DIM PH$(1000)
50 F(4)=9 : REM AUTOMATICALLY PERFORMS DIM F(10)
```

## END

TYPE: Statement

FORMAT: END

Action: This finishes a program's execution and displays the READY message, returning control to the person operating the MC-600. There may be any number of END statements within a program. While it is not necessary to include any END statements at all, it is recommended that a program does conclude with one, rather than just running out of lines. The END statement is similar to the STOP statement. The only difference is that STOP causes the computer to display the message BREAK IN LINE XX and END just displays READY. Both statements allow the computer to resume execution by typing the CONT command.

EXAMPLES of END Statement:

```
10 PRINT"DO YOU REALLY WANT TO RUN THIS PROGRAM"  
20 INPUT A$  
30 IF A$ = "NO" THEN END  
40 REM REST OF PROGRAM . . .  
999 END
```

## ERASE

TYPE: File Function

FORMAT: ERASE <filename mask>

Action: This function ERASEs the file or files specified by filename. Filename must be a valid file name, wildcards can be included; files to be erased must be closed.

WARNING: an ERASEd file is definitely lost and CANNOT BE RECOVERED

EXAMPLES of ERASE Function:

```
>ERASE "A:FILE1.BAS"  
Erase the file named FILE1.BAS from the RAM disk  
>ERASE "A:FIL*.BAS"  
Erase any .BAS file beginning with FIL from the RAM disk  
>ERASE "B:PROG1.BAS"  
Erase the file named PROG1.BAS from the FLASH disk  
>ERASE "B:PRO*.BAS"  
Erase any .BAS file beginning with PRO from the FLASH disk
```

## EXP

TYPE: Function-Numeric

FORMAT: EXP ( <number> )

Action: This mathematical function calculates the constant e (2.71828183) raised to the power of the number given. A value greater than 88.0296919 causes an ?OVERFLOW error to occur.

EXAMPLES of EXP Function:

```
10 PRINT EXP (1)  
20 X = Y*EXP (Z*Q)
```

## FILES

TYPE: File Function

FORMAT: FILES ["<filename mask>"]

Action: This function FILES lists the file directory of the specified disk unit; if no disk unit is specified, FILES lists the directory of the default disk.

EXAMPLES of FILES Function:

```
>FILES  
Prints a listing of all the files present on the default disk  
>FILES "A:"  
Prints a listing of all the files present on RAM disk A:  
>FILES "B:FIL*.BAS"  
Prints a listing of all the files present on RAM disk A:
```

## FOR ... TO ... [STEP ...

TYPE: Statement

FORMAT: FOR <variable> = <start> TO <limit> [ STEP <increment> ]

Action: This is a special BASIC statement that lets you easily use a variable as a counter. You must specify certain parameters: the floating-point variable name, its starting value, the limit of the count, and how much to

add during each cycle.

Here is a simple BASIC program that counts from 1 to 10, PRINTing each number and ENDing when complete, and using no FOR statements:

```
100 L = 1
110 PRINT L
120 L = 1 + 1
130 IF L <= 10 THEN 110
140 END
```

Using the FOR statement, here is the same program:

```
100 FOR L = 1 TO 10
110 PRINT L
120 NEXT L
130 END
```

As you can see, the program is shorter and easier to understand using the FOR statement. When the FOR statement is executed, several operations take place. The <start> value is placed in the <variable> being used in the counter. In the example above, a 1 is placed in L. When the NEXT statement is reached, the <increment> value is added to the <variable>. If a STEP was not included, the <increment> is set to + 1. The first time the program above hits line 120, 1 is added to L, so the new value of L is 2. Now the value in the <variable> is compared to the <limit>. If the <limit> has not been reached yet, the program GOes TO the line after the original FOR statement. In this case, the value of 2 in L is less than the limit of 10, so it GOes TO line 110. Eventually, the value of <limit> is exceeded by the <variable>. At that time, the loop is concluded and the program continues with the line following the NEXT statement. In our example, the value of L reaches 11, which exceeds the limit of 10, and the program goes on with line 130. When the value of <increment> is positive, the <variable> must exceed the <limit>, and when it is negative it must become less than the <limit>.

NOTE: A loop always executes at least once.

EXAMPLES of FOR...TO...STEP...Statement:

```
100 FOR L = 100 TO 0 STEP -1
100 FOR L = PI TO 6* {pi} STEP .01
100 FOR AA = 3 TO 3
```

## GET

NOT IMPLEMENTED YET

TYPE: Statement  
FORMAT: GET <variable list>

Action: This statement reads each key typed by the user. As the user is typing, the characters are stored in the MC-600's keyboard buffer. Up to 10 characters are stored here, and any keys struck after the 10th are lost. Reading one of the characters with the GET statement makes room for another character. If the GET statement specifies numeric data, and the user types a key other than a number, the message ?SYNTAX ERROR appears. To be safe, read the keys as strings and convert them to numbers later. The GET statement can be used to avoid some of the limitations of the INPUT statement. For more on this, see the section on Using the GET Statement in the Programming Techniques section.

EXAMPLES of GET Statement:

```
10 GET A$: IF A$ = "" THEN 10: REM LOOPS IN 10 UNTIL ANY KEY HIT
20 GET A$, B$, C$, D$, E$: REM READS 5 KEYS
```

**GOSUB**

TYPE: Statement  
 FORMAT: GOSUB <line number>

Action: This is a specialized form of the GOTO statement, with one important difference: GOSUB remembers where it came from. When the RETURN statement (different from the <RETURN> key on the keyboard) is reached in the program, the program jumps back to the statement immediately following the original GOSUB statement. The major use of a subroutine (GOSUB really means GO to a SUBroutine) is when a small section of program is used by different sections of the program. By using subroutines rather than repeating the same lines over and over at different places in the program, you can save lots of program space. Each time the program executes a GOSUB, the line number and position in the program line are saved in a special area called the "stack", which takes up some bytes of your memory. This limits the amount of data that can be stored in the stack. Therefore, the number of subroutine return addresses that can be stored is limited, and care should be taken to make sure every GOSUB hits the corresponding RETURN, or else you'll run out of memory even though you have plenty of bytes free.

**GOTO**

TYPE: Statement  
 FORMAT :GOTO <line number>

Action: This statement allows the BASIC program to execute lines out of numerical order. The word GOTO followed by a number will make the program jump to the line with that number. GOTO NOT followed by a number equals GOTO 0. It must have the line number after the word GOTO. It is possible to create loops with GOTO that will never end. The simplest example of this is a line that GOes TO itself, like 10 GOTO 10. These loops can be stopped by typing <CTRL-C> key on the keyboard.

EXAMPLES of GOTO Statement:

```
GOTO 100
10 GO TO 50
20 GOTO 999
```

**GOTOXY**

**NOT IMPLEMENTED YET**

TYPE: Function  
 FORMAT :GOTOXY <screen Column position, screen Row position>

Action: This is the function used by BASIC to position the cursor at a specified address on the screen. An ANSI capable terminal must be connected to the serial interface, in order to correctly execute the ANSI escape sequences needed for cursor repositioning.

EXAMPLE of GOTOXY Statement:

```
100 GOTOXY 50, 10
```

Terminal cursor is positioned at Column 50, Row 10

**IF...THEN... ELSE**

TYPE: Statement  
 FORMAT: IF <expression> THEN <line number>  
       IF <expression> GOTO <line number>  
       IF <expression> THEN <statements>

Action: This is the statement that gives BASIC most of its "intelligence," the ability to evaluate conditions and take different actions depending on the outcome. The word IF is followed by an expression, which can include

variables, strings, numbers, comparisons, and logical operators. The word THEN appears on the same line and is followed by either a line number or one or more BASIC statements. When the expression is false, everything after the word THEN on that line is ignored, and execution continues with the next line number in the program. A true result makes the program either branch to the line number after the word THEN or execute whatever other BASIC statements are found on that line.

EXAMPLE of IF...GOTO...Statement:

```
100 INPUT "TYPE A NUMBER"; N
110 IF N <= 0 GOTO 200
120 PRINT "SQUARE ROOT=" SQR(N)
130 GOTO 100
200 PRINT "NUMBER MUST BE >0"
210 GOTO 100
```

This program prints out the square root of any positive number. The IF statement here is used to validate the result of the INPUT. When the result of  $N \leq 0$  is true, the program skips to line 200, and when the result is false the next line to be executed is 120. Note that THEN GOTO is not needed with IF...THEN, as in line 110 where GOTO 200 actually means THEN GOTO 200.

EXAMPLE OF IF...THEN...Statement:

```
100 FOR L = 1 TO 100
110 IF RND(1) < .5 THEN X=X+1: GOTO 130
120 Y=Y+1
130 NEXT L
140 PRINT "HEADS=" X
150 PRINT "TAILS= " Y
```

The IF in line 110 tests a random number to see if it is less than .5. When the result is true, the whole series of statements following the word THEN are executed: first X is incremented by 1, then the program skips to line 130. When the result is false, the program drops to the next statement, line 120.

## INPUT

TYPE: Statement

FORMAT: INPUT [ "<prompt>" ; ] <variable list>

Action: This is a statement that lets the person RUNNING the program "feed" information into the MC-600. When executed, this statement PRINTs a question mark (?) on the screen, and positions the cursor 1 space to the right of the question mark. Now the computer waits, cursor blinking, for the operator to type in the answer and press the <RETURN> key. The word INPUT may be followed by any text contained in quote marks (""). This text is PRINTed on the screen, followed by the question mark. After the text comes a semicolon (;) and the name of one or more variables separated by commas. This variable is where the computer stores the information that the operator types. The variable can be any legal variable name, and you can have several different variable names, each for a different input.

EXAMPLES of INPUT Statement:

```
100 INPUT A
110 INPUT B, C, D
120 INPUT "PROMPT"; E
```

When this program RUNs, the question mark appears to prompt the operator

that the MC-600 is expecting an input for line 100. Any number typed in goes into A, for later use in the program. If the answer typed was not a number, the ?REDO FROM START message appears, which means that a string was received when a number was expected. If the operator just hits <RETURN> without typing anything, the variable's value doesn't change. Now the next question mark, for line 110, appears. If we type only one number and hit the <RETURN>, MC-600 will now display 2 question marks (??), which means that more input is required. You can just type as many inputs as you need separated by commas, which prevents the double question mark from appearing. If you type more data than the INPUT statement requested, the ?EXTRA IGNORED message appears, which means that the extra items you typed were not put into any variables. Line 120 displays the word PROMPT before the question mark appears. The semicolon is required between the prompt and any list of variables. The INPUT statement can never be used outside a program. The MC-600 needs space for a buffer for the INPUT variables, the same space that is used for commands.

## **INSTA**

TYPE: Integer Function  
FORMAT: INTSTA ()

Action: Returns the input value of the MC-600 input port.

EXAMPLES of INTSTA Function:

```
120 PRINT INTSTA()
```

```
129
```

Indicates that MC-600 board's INPUT7 and INPUT0 are active, while all others are not.

## **INT**

TYPE: Integer Function  
FORMAT: INT (<numeric>)

Action: Returns the integer value of the expression. If the expression is positive, the fractional part is left off. If the expression is negative, any fraction causes the next lower integer to be returned.

EXAMPLES of INT Function:

```
120 PRINT INT(99.4343), INT(-12.34)
```

```
99      -13
```

## **LEFT\$**

TYPE: String Function  
FORMAT: LEFT\$ (<string>, <integer>)

Action: Returns a string comprised of the leftmost <integer> characters of the <string>. The integer argument value must be in the range 0 to 255. If the integer is greater than the length of the string, the entire string will be returned. If an <integer> value of zero is used, then a null string (of zero length) is returned.

EXAMPLES of LEFT\$ Function:

```
10 A$ = "MC-600 motion platform"
```

```
20 B$ = LEFT$(A$,6): PRINT B$
```

```
RUN
```

```
MC-600
```

## LEN

TYPE: Integer Function  
Format: LEN (<string>)

Action: Returns the number of characters in the string expression. Non-printed characters and blanks are counted.

EXAMPLE of LEN Function:

```
CC$ = "MC-600 motion platform": PRINT LEN(CC$)
```

22

## LET

TYPE: Statement  
FORMAT: [LET] <variable> = <expression>

Action: The LET statement can be used to assign a value to a variable. But the word LET is optional and therefore most advanced programmers leave LET out because it's always understood and wastes valuable memory. The equal sign (=) alone is sufficient when assigning the value of an expression to a variable name.

EXAMPLES of LET Statement:

```
10 LET D= 12           (This is the same as D = 12)
```

```
20 LET E$ = "ABC"
```

```
30 F$ = "WORDS"
```

```
40 SUM$= E$ + F$      (SUM$ would equal ABCWORDS)
```

## LIST

TYPE: Command  
FORMAT: LIST [[<first-line>]-[<last-line>]]

Action: The LIST command allows you to look at lines of the BASIC program currently in the memory of the MC-600. The LIST system command displays all or part of the program that is currently in memory on screen. If no line-numbers are given the entire program is listed. If only the first-line number is specified, and followed by a hyphen (-), that line and all higher-numbered lines are listed. If only the last line-number is specified, and it is preceded by a hyphen, then all lines from the beginning of the program through that line are listed. If both numbers are specified, the entire range, including the line-numbers LISTed, is displayed.

EXAMPLES of LIST Command:

```
LIST      (Lists the program currently in memory.)
```

```
LIST 500  (Lists line 500 only.)
```

```
LIST 150- (Lists all lines from 150 to the end.)
```

```
LIST -1000 (Lists all lines from the lowest through 1000.)
```

```
LIST 150-1000 (Lists lines 150 through 1000, inclusive.)
```

## LOAD

TYPE: Command  
FORMAT: LOAD "filename"

Action: The LOAD statement reads the contents of a program file from the specified filename; filename must be unique and does not allow the use of wildcards; if no disk letter is specified into filename, the default disk unit is

assumed. The LOAD command executes an implicit NEW command before loading the new program; any program previously present in memory is erased.

EXAMPLES of LOAD Command:

LOAD "TEST.BAS"

Loads the program file TEST.BAS saved on the default disk

LOAD "B:TEST2.BAS"

Loads the program file TEST2.BAS saved on the FLASH disk

## LOG

TYPE: Floating-Point Function

FORMAT: LOG(<numeric>)

Action: Returns the natural logarithm (log to the base of e) of the argument. If the value of the argument is zero or negative the BASIC error message ?ILLEGAL QUANTITY will occur.

EXAMPLES of LOG Function:

```
25 PRINT LOG(45/7)
      1.86075234
```

```
10 NUM=LOG(ARG)/LOG(10) (Calculates the LOG of ARG to the base 10)
```

## MERGE

TYPE: Command

FORMAT: MERGE "filename"

Action: The MERGE statement reads the contents of a program file from the specified filename; filename must be unique and does not allow the use of wildcards; if no disk letter is specified into filename, the default disk unit is assumed. The MERGE command loads the program into the existing workspace; any program previously present in memory is maintained; lines with same number are replaced by lines present in the new loaded file.

EXAMPLES of MERGE Command:

MERGE "TEST.BAS"

Loads the program file TEST.BAS saved on the default disk

MERGE "B:TEST2.BAS"

Loads the program file TEST2.BAS saved on the FLASH disk

## MID\$

TYPE: String Function

FORMAT: MID\$(<string>,<numeric-1>[,<numeric-2>])

Action: The MID\$ function returns a sub-string which is taken from within a larger <string> argument. The starting position of the sub-string is defined by the <numeric-1> argument and the length of the sub-string by the <numeric-2> argument. Both of the numeric arguments can have values ranging from 0 to 255. If the <numeric-1> value is greater than the length of the <string>, or if the <numeric-2> value is zero, then MID\$ gives a null string value. If the <numeric-2> argument is left out, then the computer will assume that a length of the rest of the string is to be used. And if the source string has fewer characters than <numeric-2>, from the starting position to the end of the string argument, then the whole rest of the string is used.

EXAMPLE of MID\$ Function:

```
10 A$="GOOD"
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$ + MID$(B$,8,8)
```

GOOD EVENING

## MOD

TYPE: Integer Function  
FORMAT: MOD(<numeric1%>, <numeric2%>)

Action: Returns the remainder of the integer division of numeric1 by numeric2. If the value of the numeric2 argument is zero, the BASIC error message DIVISION BY ZERO will occur.

EXAMPLES of MOD Function:

```
25 PRINT MOD(9, 2)
1 (9 / 2 = 4, remainder is 1)
```

## NEW

TYPE: Command  
FORMAT: NEW

Action: The NEW command is used to delete the program currently in memory and clear all variables. Before typing in a new program, NEW should be used in direct mode to clear memory. NEW can also be used in a program, but you should be aware of the fact that it will erase everything that has gone before and is still in the MC-600's memory. This can be particularly troublesome when you're trying to debug your program.

BE CAREFUL: Not clearing out an old program before typing a new one can result in a confusing mix of the two programs.

EXAMPLES of NEW Command:

```
NEW          (Clears the program and all variables)
10 NEW       (Performs a NEW operation and STOPS the program.)
```

## NEXT

TYPE: Statement  
FORMAT: NEXT[<counter>][,<counter>]...

Action: The NEXT statement is used with FOR to establish the end of a FOR...NEXT loop. The NEXT need not be physically the last statement in the loop, but it is always the last statement executed in a loop. The <counter> is the loop index's variable name used with FOR to start the loop. A single NEXT can stop several nested loops when it is followed by each FOR's <counter> variable name(s). To do this each name must appear in the order of inner-most nested loop first, to outer-most nested loop last. When using a single NEXT to increment and stop several variable names, each variable name must be separated by commas. Loops can be nested to 9 levels. If the counter variable(s) are omitted, the counter associated with the FOR of the current level (of the nested loops) is incremented. When the NEXT is reached, the counter value is incremented by 1 or by an optional STEP value. It is then tested against an end-value to see if it's time to stop the loop. A loop will be stopped when a NEXT is found which has its counter value greater than the end-value.

EXAMPLES of NEXT Statement:

```
10 FOR J=1 TO 5: FOR K=10 TO 20: FOR N=5 TO -5 STEP - 1
20 NEXT N,K,J          (Stopping Nested Loops)
```

```

10 FOR L=1 TO 100
20 FOR M=1 TO 10
30 NEXT M
400 NEXT L          (Note how the loops do NOT cross each other)

```

```

10 FOR A=1 TO 10
20 FOR B=1 TO 20
30 NEXT
40 NEXT          (Notice that no variable names are needed)

```

## NOT

TYPE: Logical Operator  
 FORMAT: NOT <expression>

Action: The NOT logical operator "complements" the value of each bit in its single operand, producing an integer "twos-complement" result. In other words, the NOT is really saying, "if it isn't. When working with a floating-point number, the operands are converted to integers and any fractions are lost. The NOT operator can also be used in a comparison to reverse the true/false value which was the result of a relationship test and therefore it will reverse the meaning of the comparison. In the first example below, if the "twos-complement" of "AA" is equal to "BB" and if "BB" is NOT equal to "CC" then the expression is true.

EXAMPLES of NOT Operator:

```
10 IF NOT AA = BB AND NOT(BB = CC) THEN...
```

```

NN% = NOT 96: PRINT NN%
-97

```

NOTE: TO find the value of NOT use the expression  $X = -(X+1)$ . (The two's complement of any integer is the bit complement plus one.)

## ON

TYPE: Statement  
 FORMAT: ON <variable> GOTO / GOSUB <line-number>[, <line-number>]...

Action: The ON statement is used to GOTO one of several given line-numbers, depending upon the value of a variable. The value of the variables can range from zero through the number of lines given. if the value is a non-integer, the fractional portion is left off. For example, if the variable value is 3, ON will GOTO the third line-number in the list. If the value of the variable is negative, the BASIC error message ?ILLEGAL QUANTITY occurs. If the number is zero, or greater than the number of items in the list, the program just "ignores" the statement and continues with the statement following the ON statement. ON is really an underused variant of the IF...THEN...statement. Instead of using a whole lot of IF statements each of which sends the program to 1 specific line, 1 ON statement can replace a list of IF statements. When you look at the first example you should notice that the 1 ON statement replaces 4 IF...THEN... statements.

EXAMPLES of ON Statement:

```

ON -(A=7)-2*(A=3)-3*(A<3)-4*(A>7)GOTO 400,900,1000,100
ON X GOTO 100,130,180,220
ON X+3 GOSUB 9000,20,9000
100 ON NUM GOTO 150,300,320,390
500 ON SUM/2 + 1 GOSUB 50,80,20

```

## OPEN

TYPE: I/O Statement

**NOT IMPLEMENTED YET**

FORMAT: HANDLE% = OPEN "<filename>"

Action: This statement opens "filename" for I/O purposes. An integer number, to be used as file handle, is returned by this function; the file handle will be used in any further reference to the opened file.

EXAMPLES of OPEN Statement:

10 FILE1% = OPEN "MATRIX.TXT"  
(Opens the file MATRIX.TXT for I/O purposes)

**OR**

TYPE: Logical Operator  
FORMAT: <operand> OR <operand>

Action: Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value which can then be used in a decision. When used in calculations, the logical OR gives you a bit result of 1 if the corresponding bit of either or both operands is 1. This will produce an integer as a result depending on the values of the operands. When used in comparisons the logical OR operator is also used to link two expressions into a single compound expression. If either of the expressions are true, the combined expression value is true (-1). In the first example below if AA is equal to BB OR if XX is 20, the expression is true. Logical operators work by converting their operands to 16-bit, signed, two's complement integers in the range of -32768 to +32767. If the operands are not in the range an error message results. Each bit of the result is determined by the corresponding bits in the two operands.

EXAMPLES of OR Operator:

100 IF (AA=BB) OR (XX=20) THEN...

230 KK%=64 OR 32: PRINT KK% (You typed this with a bit value of 1000000 for 64 and 100000 for 32)

96 (the computer responded with bit value 1100000. 1100000=96.)

**PASSWORD**

TYPE: Command  
FORMAT: PASSWORD

Action: The PASSWORD statement can be used to specify a PASSWORD to be used by the system in order to avoid unauthorized access or modification to the BASIC program; upon entering the PASSWORD command, the user is asked to type its current password and to type a new system's password twice.

**PRINT**

TYPE: Statement  
FORMAT: PRINT [<variable>][<,/><variable>]...

Action: The PRINT statement is used to write data items to the screen. The <variable(s)> in the output-list are expressions of any type. If no output-list is present, a blank line is printed. The position of each printed item is determined by the punctuation used to separate items in the output-list. The punctuation characters that you can use are blanks, commas, or semicolons. The 80-character logical screen line is divided into 8 print zones of 10 spaces each. In the list of expressions, a comma causes the next value to be

printed at the beginning of the next zone. A semicolon causes the next value to be printed immediately following the previous value. However, there are two exceptions to this rule:

- 1) Numeric items are followed by an added space.
- 2) Positive numbers have a space preceding them.

When you use blanks or no punctuation between string constants or variable names it has the same effect as a semicolon. However, blanks between a string and a numeric item or between two numeric items will stop output without printing the second item. If a comma or a semicolon is at the end of the output-list, the next PRINT statement begins printing on the same line, and spaced accordingly. If no punctuation finishes the list, a carriage-return and a line-feed are printed at the end of the data. The next PRINT statement will begin on the next line. There is no statement in BASIC with more variety than the PRINT statement. There are so many symbols, functions, and parameters associated with this statement that it might almost be considered as a language of its own within BASIC; a language specially designed for writing on the screen.

EXAMPLES of PRINT Statement:

```
1)
5 X = 5
10 PRINT -5*X,X-5,X+5,X^5

-25  0  10  3125

2)
5 X=9
10 PRINT X;"SQUARED IS";X*X;"AND";
20 PRINT X "CUBED IS" X^3

9 SQUARED IS 81 AND 9 CUBED IS 729
```

## READ

TYPE: Statement  
FORMAT: READ <variable>[,<variable>]...

Action: The READ statement is used to fill variable names from constants in DATA statements. The data actually read must agree with the variable types specified or the BASIC error message ?SYNTAX ERROR will result. (\*) Variables in the DATA input-list must be separated by commas. A single READ statement can access one or more DATA statements, which will be accessed in order (see DATA), or several READ statements can access the same DATA statement. If more READ statements are executed than the number of elements in DATA statements(s) in the program, the BASIC error message ?OUT OF DATA is printed. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements will continue reading at the next data element. (See RESTORE.)

EXAMPLES of READ Statement:

```
110 READ A,B,C$
120 DATA 1,2,HELLO

100 FOR X=1 TO 10: READ A(X):NEXT

200 DATA 3.08, 5.19, 3.12, 3.98, 4.24
210 DATA 5.08, 5.55, 4.00, 3.16, 3.37
```

(Fills array items (line 1) in order of constants shown (line 5))

```
1 READ CITY$,STATE$,ZIP
5 DATA DENVER,COLORADO, 80211
```

## REM

TYPE: Statement  
FORMAT: REM [<remark>]

Action: The REM statement makes your programs more easily understood when LISTed. It's a reminder to yourself to tell you what you had in mind when you were writing each section of the program. For instance, you might want to remember what a variable is used for, or some other useful information. The REMark can be any text, word, or character including the colon (: ) or BASIC keywords.

The REM statement and anything following it on the same line-number are ignored by BASIC, but REMarks are printed exactly as entered when the program is listed. A REM statement can be referred to by a GOTO or GOSUB statement, and the execution of the program will continue with the next higher program line having executable statements.

EXAMPLES of REM Statement:

```
10 REM CALCULATE AVERAGE VELOCITY
20 FOR X= 1 TO 20 :REM LOOP FOR TWENTY VALUES
30 SUM=SUM + VEL(X): NEXT
40 AVG=SUM/20
```

## RENUM

TYPE: Command  
FORMAT: RENUM [<start>[,<increment>]]

Action: Renumber program lines. By default, the new sequence is 10,20,30,... The first argument is a new initial line number; the second argument is the increment between line numbers.

## RESTORE

TYPE: Statement  
FORMAT: RESTORE

Action: BASIC maintains an internal pointer to the next DATA constant to be READ. This pointer can be reset to the first DATA constant in a program using the RESTORE statement. The RESTORE statement can be used anywhere in the program to begin re-READING DATA.

EXAMPLES of RESTORE Statement:

```
100 FOR X=1 TO 10: READ A(X): NEXT
200 RESTORE
300 FOR Y=1 TO 10: READ B(Y): NEXT
```

```
4000 DATA 3.08, 5.19, 3.12, 3.98, 4.24
4100 DATA 5.08, 5.55, 4.00, 3.16, 3.37
```

(Fills the two arrays with identical data)

```
10 DATA 1,2,3,4
20 DATA 5,6,7,8
30 FOR L= 1 TO 8
40 READ A: PRINT A
50 NEXT
60 RESTORE
```

```
70 FOR L= 1 TO 8
80 READ A: PRINT A
90 NEXT
```

## RETI

See the interrupt section

## RETURN

TYPE: Statement  
FORMAT: RETURN

Action: The RETURN statement is used to exit from a subroutine called for by a GOSUB statement. RETURN restarts the rest of your program at the next executable statement following the GOSUB. If you are nesting subroutines, each GOSUB must be paired with at least one RETURN statement. A subroutine can contain any number of RETURN statements, but the first one encountered will exit the subroutine.

EXAMPLE of RETURN Statement:

```
10 PRINT"THIS IS THE PROGRAM"
20 GOSUB 1000
30 PRINT"PROGRAM CONTINUES"
40 GOSUB 1000
50 PRINT"MORE PROGRAM"
60 END
1000 PRINT"THIS IS THE GOSUB":RETURN
```

## RIGHT\$

TYPE: String Function  
FORMAT: RIGHT\$ (<string>,<numeric>)

Action: The RIGHT\$ function returns a sub-string taken from the right-most end of the <string> argument. The length of the sub-string is defined by the <numeric> argument which can be any integer in the range of 0 to 255. If the value of the numeric expression is zero, then a null string ("") is returned. If the value you give in the <numeric> argument is greater than the length of the <string> then the entire string is returned.

EXAMPLE of RIGHT\$ Function:

```
10 MSG$="MC-600"
20 PRINT RIGHT$(MSG$,3)
RUN
```

600

## RUN

TYPE: Command  
FORMAT: RUN [<"file">],[<line-number>]

Action: The system command RUN is used to start the program currently in memory, if no file name is given, or to LOAD a new program (from "file") and start it; before LOADING the new program, the BASIC workspace is cleared by an internally executed NEW function. If a <line-number> is specified, your program will start on that line. Otherwise, the RUN command starts at first line of the program. The RUN command can also be used within a program. If the <line-number> you specify doesn't exist, the BASIC error message UNDEFINED LINE NUMBER occurs. If the <file> you specify doesn't exist, the BASIC error message FILE NOT FOUND occurs. A RUNNING program stops and BASIC returns to direct mode when an END or STOP statement is reached, when the last line of the program is finished, or when a BASIC error occurs during execution.

EXAMPLES of RUN Command:

```
RUN          (Starts at first line of program)
RUN 500      (Starts at line-number 500)
RUN X        (Starts at line X, or UNDEFINED LINE NUMBER
              if there is no line X)
RUN "FOO.BAS" (Loads FOO.BAS and starts it at first line)
RUN "FOO.BAS",500 (Loads FOO.BAS and starts it at line 500)
```

## SAVE

TYPE: Command  
FORMAT: SAVE ["<filename>"]

Action: The SAVE command is used to store the program that is currently in memory into a file ["<filename>"] on the Flash or RAM disk; if no filename extension is given, the default BAS extension will be appended.

EXAMPLES of SAVE Command.

```
SAVE "TEST"      (Write to the default disk unit the present program as
                  file TEST.BAS)
```

## SGN

TYPE: Integer Function  
FORMAT: SGN (<numeric>)

Action: SGN gives you an integer value depending upon the sign of the <numeric> argument. If the argument is positive the result is 1, if zero the result is also 0, if negative the result is -1.

EXAMPLE of SGN Function:

```
90 ON SGN(DV)+2 GOTO 100, 200, 300
(jump to 100 if DV=negative, 200 if DV=0, 300 if DV=positive)
```

## SIN

TYPE: Floating-Point Function  
FORMAT: SIN (<numeric>)

Action: SIN gives you the sine of the <numeric> argument, in radians. The value of COS(X) is equal to SIN(x+3.14159265/2).

EXAMPLE of SIN Function:

```
235 AA=SIN(1.5):PRINT AA
.997494987
```

## SPC

TYPE: String Function  
FORMAT: SPC (<numeric>)

Action: The SPC function is used to control the formatting of data, as either an output to the screen or into a logical file. The number of SPaCes given by the <numeric> argument are printed, starting at the first available position. For screen or tape files the value of the argument is in the range of 0 to 255 and for disk files up to 254.

EXAMPLE of SPC Function:

```
10 PRINT"RIGHT "; "HERE &";
20 PRINT SPC(5)"OVER" SPC(14)"THERE"
RUN
```

```
RIGHT HERE &    OVER          THERE
```

## SQR

TYPE: Floating-Point Function  
FORMAT: SQR (<numeric>)

Action: SQR gives you the value of the SQuaRe of the <numeric> argument.  
EXAMPLE of SQR Function:

```
FOR J = 2 TO 5: PRINT J*5, SQR(J*5): NEXT
```

```
10 100
15 225
20 400
25 626
```

## SQRT

TYPE: Floating-Point Function  
FORMAT: SQRT (<numeric>)

Action: SQRT gives you the value of the SQuaRe RooT of the <numeric> argument. The value of the argument must not be negative, or the BASIC error message ?ILLEGAL QUANTITY will happen.

EXAMPLE of SQRT Function:

```
FOR J = 2 TO 5: PRINT J*5, SQRT(J*5): NEXT
```

```
10 3.16227766
15 3.87298335
20 4.47213595
25 5
```

READY

## STEP

TYPE: Statement  
FORMAT: [STEP <expression>]

Action: The optional STEP keyword follows the <end-value> expression in a FOR statement. It defines an increment value for the loop counter variable. Any value can be used as the STEP increment. Of course, a STEP value of zero will loop forever. If the STEP keyword is left out, the increment value will be + 1. When the NEXT statement in a FOR loop is reached, the STEP increment happens. Then the counter is tested against the end-value to see if the loop is finished. (See FOR statement for more information.)

NOTE: The STEP value can NOT be changed once it's in the loop.

EXAMPLES of STEP Statement:

```
25 FOR XX=2 TO 20 STEP 2           (Loop repeats 10 times)
35 FOR ZZ=0 TO -20 STEP -2       (Loop repeats 11 times)
```

## STOP

TYPE: Statement  
FORMAT: STOP

Action: The STOP statement is used to halt execution of the current program and return to direct mode. Typing the <CTRL-C> key on the keyboard has the same effect as a STOP statement. The BASIC error message ?BREAK IN LINE nnnnn is displayed on the screen, followed by READY. The "nnnnn" is the line-number where the STOP occurs. Any open files remain open and all variables are preserved and can be examined. The program can be restarted

by using CONT or GOTO statements.

EXAMPLES of STOP Statement:

```
10 INPUT#1,AA,BB,CC
20 IF AA=BB AND BB=CC THEN STOP
30 STOP
```

(If the variable AA is -1 and BB is equal to CC then:)

```
BREAK IN LINE 20
```

```
BREAK IN LINE 30 (For any other data values)
```

## STR\$

TYPE: String Function

FORMAT: STR\$ (<numeric>)

Action: STR\$ gives you the STRing representation of the numeric value of the argument. When the STR\$ value is converted to each variable represented in the <numeric> argument, any number shown is followed by a space and, if it's positive, it is also preceded by a space.

EXAMPLE of STR\$ Function:

```
100 FLT = 1.5E4: ALPHA$ = STR$(FLT)
110 PRINT FLT, ALPHA$
```

```
15000 15000
```

## TAB

TYPE: String Function

FORMAT: TAB (<numeric>)

Action: The TAB function moves the cursor to a relative SPC move position on the screen given by the <numeric> argument, starting with the left-most position of the current line. The value of the argument can range from 0 to 255. The TAB function should only be used with the PRINT statement, since it has no effect if used with PRINT# to a logical file.

EXAMPLE of TAB Function:

```
100 PRINT"NAME" TAB(25) "AMOUNT": PRINT
110 INPUT#1, NAM$, AMT$
120 PRINT NAM$ TAB(25) AMT$
```

```
NAME                AMOUNT
```

```
G.T. JONES          25.
```

## TAN

TYPE: Floating-Point Function

FORMAT: TAN (<numeric>)

Action: Returns the tangent of the value of the <numeric> expression in radians. If the TAN function overflows, the BASIC error message ?DIVISION BY ZERO is displayed.

EXAMPLE of TAN Function:

```
10 XX=.785398163: YY=TAN(XX):PRINT YY
1
```

## VAL

TYPE: Numeric Function

FORMAT: VAL (<string>)

Action: Returns a numeric VALue representing the data in the <string> argument. If the first non-blank character of the string is not a plus sign (+), minus sign (-), or a digit the VALue returned is zero. String conversion is finished when the end of the string or any non-digit character is found (except decimal point or exponential e).

EXAMPLE of VAL Function:

```
10 INPUT NAM$, ZIP$
20 IF VAL(ZIP$) < 19400 OR VAL(ZIP$) > 96699
   THEN PRINT NAM$ TAB(25) "GREATER PHILADELPHIA"
```

## WHILE...WHEND

TYPE: Statement  
FORMAT: WHILE <expression>  
          <statements>  
          WEND <expression>

Action: The WHILE command is used to Execute <statements> repeatedly until the WHILE condition (if given) becomes false, or until the WEND condition becomes true. This structure can emulate Pascal's WHILE-DO and REPEAT-UNTIL, or even both at once. If no conditions are given, the loop will never terminate unless the GOTO is used.

EXAMPLE of WHILE <expression>...WHEND Statement:

```
100 M = 0
110 INPUT "TYPE A NUMBER"; N
110 WHILE (M < N)
120 PRINT "SQUARE OF M IS=" SQR(M)
130 M=M+1
140 WHEND
150 PRINT "END OF PROGRAM"
```

This program prints out the square of all numbers comprised between 0 and N. The WHILE statement here is used to validate the INPUT: if INPUT is <= 0, no action is taken and lines between 120 and 140 are not executed.

EXAMPLE of WHILE...WHEND <expression> Statement:

```
100 M = 0
110 INPUT "TYPE A NUMBER"; N
110 WHILE
120 PRINT "SQUARE OF M IS=" SQR(M)
130 M=M+1
140 WHEND (M < N)
150 PRINT "END OF PROGRAM"
```

In this case, the program prints out the square of all numbers comprised between 0 and N, but the WHEND statement is used here to validate the exit condition: program lines 110..140 are executed at least once, before loop exit condition is evaluated.

## XOR

TYPE: Logical Operator  
FORMAT: <operand> XOR <operand>

Action: When used in calculations, the XOR gives you a bit result of 1 if the corresponding bit of the operands are different, while results in a 0 if the corresponding bits are equal.