DMBASIC 2.7 / CAN 2.4

Software of the SI2-CBB

1. Global description

1.1 Introduction

The SI2-CBB software is based on Olimex Duinomite hardware and DMBasic, including the basic CAN extension by Frank Voorburg. SI-Kwadraat extended the system and made a specific CAN firmware. It uses the open DMBasic version 2.7, which is open for every user. To make it fit, some commands are removed and some others modified. To work with the CAN firmware a license has to be installed. If no license is installed only the basic commands will work completely, the other CAN commands will only work in a demo version. The license is determined per unit. If the license is correct all functions are available, otherwise only available for demonstration.

1.2 DMBasic 2.7

DMBasic version 2.7 is the Duinomite version of MMBasic 2.7 by Geoff Graham. This was (and still is) an open source version of MMBasic. In this manual we copied the used statements of this Basic from the Duinomite manual:
https://www.olimex.com/Products/Duino/Duinomite/_resources/DuinoMite-UM-1-03.pdf
Some of the statements have been changed, extended or deleted by SI-Kwadraat. The copied original text of the statements in the manual are written in normal font, the SI-Kwadraat added issues in bold, the edited statements in italic. All statements beginning with CAN are developed by SI-Kwadraat. Different from the original manual is the sorting of the statements. They are now split into commands and functions and positioned in alphabetic order. Every line in DMBasic has to start with a command. Also after a : a new command has to be given. A line may start (also after a :) with a variable name; in this case the LET command is presumed to be in front of the variable name. Functions are in between commands, parameters and values.

1.3 Licensing of the SI2-CBB software

At initialisation a license file is opened and the reserved FIFO space and optional protocol are determined. The license filename cannot be changed by the user; the buffersize (no. of FIFO's) can be changed. The original number of FIFO's is 64. If this is the maximum which is needed, no extra memory is reserved for the FIFO's.
However if the value is greater (65 - 1024) a Basic array is created, CANMESSAGEFIFOAREA with a dimension of 4 * No. of FIFO's, eg at 1024: DIM CANMESSAGEFIFOAREA(4096). This array is created at the first CANOPEN or CANFIFO statement in a program or in the command shell. The use of the variable CANMESSAGEFIFOAREA is forbidden in Basic in this case.
When also the second CANport is used, the variable CANMESSAGEFIFOAREA2 is created and cannot be used. This array always has the same dimension as CANMESSAGEFIFOAREA.

1.4 Some of the original DMBasic statements have been deleted in the CAN version:

- All Gameduino (GD) functions
- Help
- FSE (Full screen editor)
- One-wire functions
- The size of the "A" drive has been limited to 64 kbytes (was 128 kbytes)

1.5 Overview of the DMBasic commands in alphabetic order:

Syntax: COMMAND parameter(s); parameters between [ ] are optional

Commands in standard letter type are original DMBasic commands; **bold ones** added in CAN version and *italic ones* changed in CAN version.

1. AUTO [line number][,[increment]]
2. **CANBRIDGE [[#]fileno]**
3. **CANCLOSE**
4. **CANFIFO [fifono][,rx/tx][,length]**
5. **CANFILTER [filterno][,std id][,ext id]**
6. **CANIDSCAN**
7. **CANINT [interruptcode[,lineno]]**
8. **CANIOLINK [pinno[,fifono[,timer[,startbit[,length]]]]]**
9. **CANLINK [filter no[,enable[,mask no[,fifo no]]]]**
10. **CANLOG [#fileno,][format[,fifono[,period[,lineno]]]]**
11. **CANMASK [maskno[,std mask[,ext mask]]]**
12. **CANOBJECT [#fileno,][fifono[,timer]]**
13. **CANOPEN [speed[,special[,ts resolution]]]**
14. **CANPHYS [parameter[,no]]**
15. **CANRCV [id][,type][,id_ext][,len][,data(),ok]**
16. **CANREG [regno]**
17. **CANREPLAY [#fileno,][fifono[,format]]**
18. **CANRESET [canport]**
19. **CANRETURN**
20. **CANSCRIPT filename[, #fileno] or CANSCRIPT scriptpointer[,#fileno]**
21. **CANSEND [id][,type][,id ext][,len][,data(),ok]**
22. **CANSTATUS**
23. **CANSUB [timer][,lineno]**
24. **CANVIEW [format[,fifono[,period[,lineno[,interval]]]]]**
25. CHDIR pathname$
26. CIRCLE(xcenter, ycenter), radius[,[color][,[F]]]
27. CLEAR
28. CLOSE #filenumber[,[#]filenumber]...] or CLOSE CONSOLE
29. CLS
30. **COMLOG [#fileno,][format[,period]]**
31. CONTINUE
32. *COPYRIGHT*
33. DATA constants
34. DATE$=v$
35. DELETE [line number1][-line number2] or DELETE line number1-
36. DIM variable(subscripts)[,variable(subscripts)]...
37. DO [loop statements] LOOP or DO WHILE expression [loop statements] LOOP or DO [loop statements] LOOP UNTIL expression
38. DRIVE "A:" or DRIVE "B:"
39. EDIT [line-number]
40. ELSE in multiline IF … THEN … [ELSEIF …] ELSE … ENDIF
41. ELSEIF in multiline IF … THEN … ELSEIF … ELSE … ENDIF
42. END
43. ENDIF in multiline IF … THEN … [ELSEIF …] [ELSE …] ENDIF
44. ERASE list of array variables
45. ERROR error$
46. **EVAL statement$**
47. EXIT or EXIT FOR

48. FILES [search-pattern$]
49. FONT [#fontnumber, [scale, [reverse] or FONT LOAD filename$ AS #fontnumber or FONT UNLOAD #fontnumber
50. FOR variable=x TO y [STEP z] [statements] NEXT [variable][,variable...]
51. *GOSUB line number or GOSUB variable … RETURN*
52. GOTO line number
53. I2CDIS
54. I2CEN speed, timeout [,interrupt-line]
55. I2CRCV address, bus-hold, rcv-len, rcv-buf [, snd-len, snd-data]
56. I2CSDIS
57. I2CSEN address, mask, option, send-int-line, rcv-int-line
58. I2CSEND address, option, snd-len, snd-data [,snd-data]
59. I2CSRCV rcv-len, rcv-buf, rcv-d
60. I2CSSEND snd-len, snd-data [,snd-data]
61. IF … THEN … or GOTO … [ELSEIF …] [ELSE …] [ENDIF]
62. INPUT [prompt string;] list of variables or INPUT [prompt string,] list of variables or INPUT #filenumber, list of variables
63. IRETURN
64. KILL filename$
65. LET variable = expression
66. LINE [(x1,y1)]-(x2,y2) [,[color][,B[F]]
67. LINE INPUT [prompt$] [;] [,] string-variable or LINE INPUT #filenumber, string-variable
68. *LIST [linenumber][-linenumber] or LIST [linenumber-] or LIST #filenumber, linenumber[-linenumber]*
69. *LOAD filename$ or LOAD # filename$*
70. LOCATE X,Y
71. LOOP in DO [loop statements] LOOP or DO WHILE expression [loop statements] LOOP or DO [loop statements] LOOP UNTIL expression
72. *MEMORY [debug parameter]*
73. MERGE filename$
74. MKDIR pathname$
75. *MM.BLANK seconds*
76. *MSDOFF*
77. *MSDON*
78. NAME oldfilename$ AS newfilename$
79. NEW
80. NEXT in FOR … TO … [STEP …] NEXT
81. NUM2BYTE number, array(x) or NUM2BYTE number, variable1, variable2, variable3, variable4
82. **OLED [options[,horizontal offset[,vertical offset]]] or OLED options_ext**
83. ON expression GOTO linenumbers or ON expression GOSUB linenumbers
84. *OPEN filename$ FOR mode AS [#]filenumber or OPEN "COM[n]:[speed[,buf[,int[,lvl]]]][,mode][,FC] [,OC]" AS [#]filenumber or OPEN "COM[n]:[speed[,buf[,int[,lvl]]]][,mode][,FC] [,OC]" AS CONSOLE*
85. *OPTION BASE n or OPTION ERROR CONTINUE or OPTION ERROR ABORT or OPTION PROMPT prompt$ or OPTION Fnn string$ or OPTION VIDEO ON/OFF or OPTION USB ON/OFF/DISCONNECT*
86. PAUSE number
87. PIN(n) = value
88. PIXEL (x,y) = value
89. POKE hiword,loword,byte
90. PRESET (x,y)
91. PRINT [#filenumber,][list of expressions][;,] or ? [#filenumber,][list of expressions][;,]
92. PSET (x,y)
93. **PWM [<pin>[,<period 1>[,<period 0>]]]**

94. RANDOMIZE [expression] or RANDOMIZE TIMER
95. READ list of variables
96. REM[comment] or '[comment]
97. RENUMBER [first],[increment][,start]]
98. RESTORE
99. RETURN
100. RMDIR pathname$
101. RUN [linenumber] or RUN filename$
102. SAVE filename$ or SAVE # filename$
103. SAVEBMP filename$
104. SDDISABLE
105. SDENABLE
106. SDFORMAT
107. SETPIN pin-number, config or SETPIN pin-number, config, line-number
108. SETTICK period, line-number
109. SETUP
110. SLEEP [time]
111. SOFTRESET
112. SOUND freq, duration, duty
113. TIME$ string exp
114. TIMER = value
115. TROFF
116. TRON
117. WATCHDOG [timer]
118. WEND
119. WHILE expression [loop statements] WEND
120. WRITE [#filenum,] list-of-expressions
121. XMODEM SEND file$ or XMODEM RECEIVE file$

1.6 Overview of the DMBasic functions in alphabetic order

Syntax: FUNCTION; [ ] parameter is optional

Functions in standard letter type are original DMBasic functions; **bold ones** added in CAN version and *italic ones* changed in CAN version.

1. -
2. *
3. /
4. \
5. ^
6. +
7. <
8. <=
9. <>
10. =
11. =<
12. =>
13. >
14. >=
15. ABS(n)
16. AND
17. AS
18. ASC(x$)
19. ATN(x)
20. BYTE2NUM(array(x)) or BYTE2NUM(byte1,byte2,byte3,byte4)
21. **CANCLOCK**
22. **CANOBJECT**
23. CHR$(x)
24. CINT(x)
25. COS(x)
26. CWD$
27. DATE$
28. DOW
29. ELSE
30. EOF([#]file-number)
31. EXP(x)
32. FIX(x)
33. FOR
34. *FORMAT$(number, format$)*
35. **GETDIM(varname$)**
36. **GETPIN(x)**
37. *GOSUB line numbers*
38. GOTO line number(s)
39. HEX$(x)
40. INKEY$
41. INPUT$(number, [#]file-number)
42. INSTR([n],x$,y$)
43. INT(n)
44. LCASE$(x$)
45. LEFT$(x$,n)
46. LEN(x$)
47. LOAD "fontfile" AS [#]n
48. LOADB "fontfile" AS [#]n
49. LOC([#]file-number)
50. LOF([#]file-number)
51. LOG(x)
52. MID$
53. MM.BLANK
54. MM.BOOTUP

55. MM.DRIVE
56. MM.DRIVE$
57. MM.ERRNO
58. MM.FNAME$
59. MM.HRES
60. MM.I2C
61. MM.SETUP
62. MM.SLEEP
63. MM.VER
64. MM.VRES
65. MOD
66. NOT
67. OCT$(x)
68. OR
69. PEEK(hiword,loword)
70. PIN(n)
71. PIXEL(x,y)
72. POS
73. RIGHT$(x$,n)
74. RND
75. SGN(x)
76. SIN(x)
77. SPACE$(x)
78. SPC(n)
79. SPI(rx, tx, clock[, data][,speed])
80. SQR(x)
81. STEP
82. STR$(n)
83. STRING$
84. TAB(n)
85. TAN(x)
86. THEN
87. TIME$
88. TIMER
89. TO
90. UCASE$(x$)
91. UNTIL
92. VAL
93. WHILE
94. XOR

1.7 Overview of the BASIC variables used in CAN commands and in SLEEP and OPEN.

BASIC VARIABLE	DESCRIPTION	USED IN COMMANDNO.
CANARRAY()	Array for interaction basic-script	10,20
CANBLINK	No of messages for blink green LED	10,
CANCONTINUE	Way to continue after interval	10,
CANENDOFLINE	End of line character(s)	10,
CANERRCNT	Maximum value of Rx error counter	10,24
CANEVENTINT	Timer for activating the script without message	20
CANFLOAT1	Basic variable used in script	10,20
CANFLOAT2	Basic variable used in script	10,20
CANFLOAT3	Basic variable used in script	10,20
CANI2CADDRESS	I2C address for logging	10
CANID()	CAN IDs	6
CANIDCHANGE0	ID change of bus 0 to bus 1	2
CANIDCHANGE1	ID change of bus 1 to bus 0	2
CANIDE()	CAN ID extensions	6
CANIDNO	No of found IDs	6
CANINTREG	Interrupt register value	7
CANIOADDRESS	Start address of CANIOARRAY	8
CANIOARRAY()	Parameters of CANIOLINKs	8
CANIOCOMMAND$()	Strings with commands to be executed	8
CANIOGAIN()	Gain parameters	8
CANIONO	No. of active CANIOLINKs	8
CANIOOFFSET()	Offset parameters	8
CANLINENO	No of lines (messages) to be evaluated	10,24
CANLOAD	Overall busload	10,24
CANLOADACT	Busload over last measurement period	10,24
CANLOADMAX	Maximum busload during 1 period	10,24
CANLOADMAXTIME	Time when maximum busload occured	10,24
CANLOADPERIOD()	Busload per period	10,24

CANLOGARRAY()	Array in which CAN messages are logged	10
CANLOGFIFO	No. of FIFO's active for logging	10
CANMESSAGEFIFOAREA()	Reserved memory for CAN FIFO's	18
CANMESSAGEFIFOAREA2()	Same for the second CAN port, if available	18
CANMESSNO	Total no. of messages	10,24
CANOBJCOUNTER	Total no. of sent and received objects	12
CANOBJECTPERIOD	Display update time for object view logging	10
CANOBJnnCTRL	Control byte of object nn	12
CANOBJnnDATA()	Array with databytes of object nn	12
CANOBJnnID	ID of object nn	12
CANOBJnnOK	Result of sending or receiving of object nn	12
CANOVFCNT	Total no. of CAN overflows	10,24
CANPERIOD	Duration of the active statement	10,24
CANPHYSADDRESS	The start address of CANPHYSARRAY	14
CANPHYSARRAY()	The array with the results of CANPHYS	14
CANPHYSOK	The status of physical measurement	14
CANPOSTTRIGGER	No. of messages after the trigger	10
CANPRETRIGGER	No. of messages before the trigger	10
CANRXERRCNT	Actual value of Rx Errorcounter	10,24
CANSCRIPTPTR	Pointer to CANSCRIPT	20
CANTEST()	Array with data of several commands	4,5,9,11,12,16,22,24
CANTIME	Measurement time of command	10,24
CANTRIGGERERRCNT	Value of Rx Errorcounter for trigger	10
CANTRIGGEREXTID	ID extension for trigger	10
CANTRIGGERID	ID for trigger	10
CANTRIGGERPIN	PIN for trigger	10
CANTRIGGERVALUE	Value of PIN for trigger	10
CANTSCORRECTION	Mode of timestamp correction	10
CANTXERRCNT	Actual value of Tx Errorcounter	10,24

CIV	Variable used in COMMAND$	8
RXDATA()	Default variable data array	15
RXDLC	Default variable DLC	15
RXID	Default variable ID	15
RXIDE	Default variable ID extension	15
RXOK	Default variable status	15
RXTYPE	Default variable type	15
TXDATA	Default variable data array	21
TXOK	Default variable status	21
WAKEUP_COM	Activate wakeup from COM port	110
WAKEUP_KB	Activate wakeup from keyboard	110
WAKEUP_PIN0	Activate wakeup from PIN 0	110
WAKEUP_PIN5	Activate wakeup from PIN 5	110
WAKEUP_PIN6	Activate wakeup from PIN 6	110
WAKEUP_PIN7	Activate wakeup from PIN 7	110
COM_PARITY_REPLACE	Replace character if parity error	84
COM_PARITY_ERRORS	Counter of parity errors	84
COM_BIT9_ADDRESS	The active address in 9-bit communication	84
COM_BIT9_VALUE	The value of the ninth bit	84

2. The DMBasic commands

2.1 AUTO

Purpose:
To generate and increment line numbers automatically each time you press the ENTER key.

Syntax:
`AUTO [line number][,[increment]]`

Comments:

`AUTO` is useful for program entry because it makes typing line numbers unnecessary.

`AUTO` begins numbering at `line number` and increments each subsequent line number by `increment`. The default for both values is 10.

If `line number` is not followed by a comma, and `increment` is not specified, the last increment specified in an `AUTO` command is assumed.

If `AUTO` generates a `line number` that is already being used, an asterisk appears after the number to warn that any input will replace the existing line. However, pressing ENTER immediately after the asterisk saves the line and generates the next line number.

`AUTO` is terminated by entering CTRL-BREAK or CTRL-C.

Note
The line in which CTRL-BREAK or CTRL-C is entered is not saved. To be sure that you save all desired text, use CTRL-BREAK and CTRL-C only on lines by themselves.

Examples:

```
AUTO 100, 50
```
Generates line numbers 100, 150, 200, and so on.

```
AUTO
```
Generates line numbers 10, 20, 30, 40, and so on.

2.2 CANBRIDGE

Purpose:
Copy messages of one CANbus to the other

Syntax:
CANBRIDGE [[#]fileno]

Comments:

CANBRIDGE activated after following sequence of commands:

- **CANRESET, CANRESET 0 or CANRESET 1 (setting the first CAN port)**
- **Optional: CANFIFO (FIFO 0 should be configured for Tx, FIFO 1 for Rx)**
- **Optional: CANFILTER/CANMASK/CANLINK (enable filters on the 1st CAN port)**
- **CANOPEN bitrate (start first CAN port)**
- **CANRESET 2 (switch to the other CAN port)**
- **Optional: CANFIFO (FIFO 0 should be configured for Tx, FIFO 1 for Rx)**
- **Optional: CANFILTER/CANMASK/CANLINK (enable filters on the 2nd CAN port)**
- **CANOPEN bitrate (start second CAN port)**

CANBRIDGE can now be started and all messages on CAN port 0 will be copied to CAN port 1 and vice versa. Of course when filters are active, these filters decide which messages are actually copied.

If the optional <fileno> is included in the command, logging is done in the UNICANNER format (format 2 in CANLOG). 0 or #0 will do a screen logging 1 (#1) up to 9 (#9) file logging to the file which is opened for input by this number.

The Basic variables CANPERIOD, CANLINENO, CANCONTINUE, CANENDOFLINE and CANBLINK can be used as in CANLOG. Also <ESC> or <CTRL-C> can be used to stop the CANBRIDGE command.

The Basic variables CANIDCHANGE0 and CANIDCHANGE1 can be used to increment or decrement the ID. First one changes the ID from bus 0 to bus 1, the second one from 1 to 0. Only small changes from -127 to +127 are possible. In the std ID these are the 7 LSBs; in the ext. ID bits 24 - 18.

Example:

```
10 CANRESET:CANOPEN 500:CANMASK 0,&H7FF
20 CANFILTER 1,500:CANLINK 1,1,0,1:CANOPEN 500
30 CANRESET 2:CANOPEN 250:CANMASK 0,&H7FF
40 CANFILTER 1,400:CANLINK 1,1,0,1:CANOPEN 250
50 CANBRIDGE
```

The messages with ID 500 on the first CANbus with bitrate 500 kb/sec are copied to the second bus with bitrate 250 kb/sec. Also messages with ID 400 on the second bus are copied to the first one.

2.3 CANCLOSE

Purpose:
Close the active CAN port

Syntax:
CANCLOSE

Comments:

CANLOSE has no additional features. It sets the active CAN port to configuration mode.

If active objects of CANOBJECT are running in the background, they will be stopped by CANCLOSE.

Example:

```
100 CANOPEN 500
110 DIM CANOBJ00DATA(10):CANOBJ00ID=100:CANOBJ00CTRL=0
120 CANOBJECT 0,1000
130 PAUSE 10000
140 CANCLOSE
```

CANBUS is opened at 500 kb/sec. A message with ID 100 (no data) is sent every second and automatically stopped after 10 seconds.

2.4 CANFIFO

Purpose:
Change the FIFO configuration of the active CANport..

Syntax:
CANFIFO [fifono[,mode[,depth]]]

Comments:

CANFIFO without further parameters will give the status of all 32 FIFO's.

If DIM CANTEST(31) has been executed before CANFIFO every CANTEST(x) has the status of CANFIFO x. The status value consists of:

> **depth + (64 * type) + (256 *act) + (16384 * stat) + (32768 * ovl), where**
> **depth = no. of locations; type = 0 for rx; 1 for tx; act(ive) = 1 for every tx and for linked rx; 0 for unlinked rx; stat = status bit; ovl = overload bit.**

Default the FIFO 0 is configured for Tx (length 2) and FIFO 1 for Rx (32) by CANOPEN. All other FIFO's are default for Rx with depth 1.

CANFIFO gives the opportunity to change any FIFO to a mode (0 for rx; 1 for tx) with any depth (1-32). If CANFIFO is entered with only the fifono, the FIFO is set to rx and depth 1. If it is entered without the depth parameter, depth will be set to 1.

Please keep in mind that the available space will be limited by the buffers in the license file. Changing CANFIFO will reset all linking of the filters.

Examples:

```
CANFIFO:'an overview of all 32 FIFO's is displayed

CANFIFO 0,1,4:'FIFO 0 is set for Tx, 4 messages deep

CANFIFO 1,0,32:'FIFO 1 is set for Rx, 32 messages deep
```

2.5 CANFILTER

Purpose:
Configure the filters, which are used to filter the received messages.

Syntax:
CANFILTER [filterno[,std id[,ext id]]]

Comments:

CANFILTER with no parameters will show the status of all filters.

If DIM CANTEST(63) has been executed before CANFILTER the values of CANTEST are as following:

CANTEST(2*x) = standard value of CANFILTER x
CANTEST((2*x)+1) = extended value of CANFILTER x

However:

If standard value<2048 and no extended value:
CANTEST((2*x)+1) = 999999

If standard value>2047 (filtering on databytes)
CANTEST((2*x)+1) = extended value + 100000

CANFILTER with only filterno sets both std and ext id to 0.

CANFILTER with filterno and std id sets the standard part to the id.

CANFILTER with filterno , std id and std id sets the complete id and optionally some databytes. If used as extended ID filtering std id will contain the 11 MSBs and ext id the 18 LSBs. Filtering on databytes is only possible on standard IDs:

CANFILTER [filterno(0-31)][,std id(0-2047)][,extd id(0-262143)]: std or ext ID only

CANFILTER [filterno(0-31)][,std id(2048-4095)][,DB1(0-255)]: std ID & 2047+DB1

CANFILTER [filterno(0-31)][,std id(4096-6143)][,DB2(0-255)]: std ID & 2047+DB2

CANFILTER [filterno(0-31)][,std id(6144-8191)][,DB1-2(0-65535]: std ID & 2047+DB1-2

CANFILTER [filterno(0-31)][,std id(8192-10239)][,DB3(0-255)]: std ID & 2047+DB3

CANFILTER [filterno(0-31)][,std id(10240-12287)][,DB2-3(0-65535)]: std ID & 2047+DB2-3

CANFILTER [filterno(0-31)][,std id(12288-14335)][,DB4(0-255)]: std ID & 2047+DB4

CANFILTER [filterno(0-31)][,std id(14336-16383)][,DB3-4(0-65535]: std ID & 2047+DB3-4

Examples:

```
CANFILTER:'an overview of all 32 filters

CANFILTER 0,100:'Filter 0 set to 100 standard ID

CANFILTER 0,&HFFFFFF:'Filter 0 set to 0xFFFFFF

CANFILTER 0,&H3F,&H3FFFF;'The same split in std and ext

CANFILTER 0,2148,100:'Filter 0 set to ID 100; DB1 100

CANFILTER 0,6244,25700:'The same + DB2 100
```

2.6 CANIDSCAN

Purpose:
Scan all the IDs on a network

Syntax:
CANIDSCAN

Comments:

To use this command the array CANID(n) has to be defined. The dimension has to be minimal as large as the number of ID's, otherwise the additional ID's will be lost. If only CANID has been defined, both standard as well as extended ID's are written in this array. If the extended ID is larger than 1.000.000 errors will occur due to the fact that the float in Basic is rounded. Therefore also an array CANIDE(n) can be dimensioned, which will contain the 18 LSB's of the extended ID. CANID(n) will contain the 11 MSB's in this case. All cells in the arrays which are not used will have the value of -1, so also CANIDE(x) at a standard ID.

As soon as CANIDSCAN is started it will look for unique IDs on the network. The number of IDs is displayed on the screen. If this is not wanted, one can dimension the CANTEST variable as an array. Only CANTEST(0) is used and will contain the number of IDs. If the variable CANIDNO is dimensioned the number of IDs is available in this variable.

The measurement is ended by <ESC> or after the number of ms defined in the Basic variable CANPERIOD if defined.

By default the command will display the number of ID's on the screen with a standard interval (1000 ms by default), which can be changed by CANSUB.

Example:

```
10 DIM CANID(10),CANIDE(10)
20 CANPERIOD=10000
30 CANIDSCAN
40 FOR x=0 TO 5:PRINT CANID(x);CANIDE(x);:NEXT
RUN
Found IDs:      3
 100-1 200-1 100 1-1-1-1-1-1-1
```

During 10 seconds 3 different ID's are detected on the bus: the standard IDs 100 and 200 and the extended ID 262244

2.7 CANINT

Purpose:
Jump to a Basic subroutine if an interrupt occurs

Syntax:
`CANINT [interruptcode[,lineno]]`

Comments:

As Basic is an interpreter (or scripting) language, we normally don't use direct processor interrupts. The CAN controllers have an interrupt register. Normally we handle the interrupt in the specific CAN statements. With `CANINT` it is possible to handle it at a more low level.

A CAN interrupt can occur on 9 different events in the controller. Each event can be enabled or disabled. Normally they are all disabled. The controller has a 16 bit word to control them (7 bits not used). The parameter `interruptcode` is a direct entry for the enable/disable word. It is ordered:
<IWBSONNNNNNNMCRT> where:
I = Invalid Message Received
W = Wakeup
B = Bus error
S = System error
O = Overflow error
N = Not used
M = Mode changed
C = CANclock overrun
R = Receive Message
T = Transmit Message

Setting the bit to <1> means enable, <0> disable. For more info concerning the interrupts we advise to read the section 34 (CAN section) of the Microchip PIC32 manual.
The Transmit, Receive and Overflow Interrupts find their sources in the specific FIFO interrupts and are quite difficult to be used in Basic. Therefore they are not used up to now.

The remaining 6 interrupts can be coded in the interrupt code and if also a `lineno` is filled in, the program jumps to the specified `lineno` if an interrupt occurs. To return to the main program one has to use the `CANRETURN` statement at the end of the subroutine.

Example:

```
100 CANOPEN 500
110 CANINT &H2000,200:'Interrupt on bus error
120 GOTO 120
200 PRINT "Bus error detected after";TIMER;" msec."
210 CANRETURN
```

2.8 CANIOLINK

Purpose:
Link I/O pins directly to incoming or outgoing CAN messages

Syntax:
CANIOLINK [pinno[,fifono[,timer[,startbit[,length]]]]]

Comments:

CANIOLINK gives the user the possibility to measure and control the I/O pins and enter the data directly into the CANOBJECT data. In fact it is possible now to configure the CANOBJECTs and the CANIOLINKs in a program or from the prompt and after the configuration the CAN messages are updated with the inputs and the outputs are controlled by the CAN messages.

Like the CANOBJECTs the CANIOLINKs are executed in the 1 ms timer interrupt routine. Up to 128 CANIOLINKs can be defined. To prevent timing problems it is decided that during every interrupt routine only one CANIOLINK is controlled. In this way the CANIOLINK is checked every ms if only one is declared and every CANIOLINK is checked every 0.1 sec if 100 are declared. Multiple I/O's can be linked to a CANOBJECT and also an I/O can be linked to multiple CANOBJECTs.

The variables are defined as follows:
pinno is the number of the PIN (or I/O line). 0 the BUTTON (PIN 0 as output, the green LED is disabled), 1- 6 the analogue inputs, 7 - 10 the special I/O lines, 11 - 18 standard digital I/O, 19 - 21 specials.

fifono is the used FIFO, which is identical to the CANOBJECT no. The CANOBJECT has to be made active before the CANIOLINK can be made to it.

timer is the minimal time interval for checking the specific CANIOLINK. The actual interval can be longer if more than one CANIOLINKs are defined. If timer is 0, a special routine is called. Now either the value of the PIN if it is a Tx CANOBJECT or the value of the relevant Rx CANOBJECT data is compared with the previous value. If it is changed the Tx CANOBJECT is sent immediately or the output is updated immediately after the Rx CANOBJECT. It has become a so-called Change Of State issue instead of a timed one.

startbit is the bit in the CANOBJECT data where the value of CANIOLINK starts. It can be any number from 0 to 63 (8 databytes). We do it per bit, so that we can place more than one PIN value into a CAN databyte. In case of analogue inputs or PWM outputs it is the place of the LSB. If the length (next parameter) is 1, the startbit can have every value. In all other cases it will be corrected to a byte, word or long word value.

length is the length of the data. It can have the values 0, 1, 8, 16 or 32. All other values between 0 and 63 will be corrected to one of those values. The values 8, 16 and 32 mean 1, 2 or 4 databytes are involved in the action. 1 means only the specified startbit is influenced. 0 has a special meaning: the value is interpreted as a float value. This means 4 bytes are involved.

To use the `CANIOLINK` the array `CANIOARRAY` should be dimensioned. Every link needs 4 entries, so for 1 `CANIOLINK` the minimum is `DIM CANIOARRAY(4)`, for 2 `DIM CANIOARRAY(8)`, etc. `CANIOARRAY(0) - CANIOARRAY(3)` are used for `CANIOLINK 0`, `CANIOARRAY(4) - CANIOARRAY(7)` for `CANIOLINK 1`, etc. The first 3 parameters are floating numbers and can be read (or even written) also from the Basic program. The 4th one is a combination of the parameters of `pinno`, OBJECTno (`fifono`), `startbit` and `length`. The value of the timer is in the 3rd parameter, so `PRINT CANIOARRAY(2)` will print the value of the timer of `CANIOLINK 0`. `PRINT CANIOARRAY(0)` will print the actual value of the chosen PIN. `PRINT CANIOARRAY(1)` depends on the timer; if `timer` is 0 it will print the previous value of the PIN (this is the way the Change Of State is detected); if the `timer` > 0 it will print the actual value of the ms counter, which will be reset to 0 if the value reached the timer value.

To make it possible to read back also the values of the 4th parameter (`pinno, fifono,start,length`) a Basic variable `CANIOADDRESS` can be declared. If this is done the variable gets the two low bytes of the start value of the `CANIOARRAY` in memory. By using `PEEK` it can be read, e.g.:

```
pinno_0=PEEK(&HA001,CANIOADDRESS+12):objno_0=PEEK(&HA001,CANIOADDR
ESS+13):start_0=PEEK(&HA001,CANIOADDRESS+14):length_0=PEEK(&HA001,
CANIOADDRESS+15)
```

Every next `CANIOLINK` is situated 16 bytes further.
If the Basic variable `CANIONO` is declared, it contains the number of active `CANIOLINK`s. If one or more `CANIOLINK`s are defined they are listed in `CANSTATUS` with all their parameters. They get a LINK-number in the order they have been entered. Normally the configuration of the links is done only once, but it is possible to change a configuration dynamically. This is done by deleting `CANIOLINK`s and entering new ones. For deleting the commands `CANIOLINK 100` up to `CANIOLINK 227` can be used. The number has to be subtracted by 100 to get the LINK-number which has to be deleted. In this way `CANIOLINK`s can be deleted one by one. The list is always cleaned up after every delete action.

Two additional Basic arrays can be dimensioned: `CANIOOFFSET` and `CANIOGAIN`. If they are used they can have a dimension which is lower than the number of `CANIOLINK`s. If they are defined the actual value for the `CANOBJECT data = (PIN(x) - CANIOOFFSET(x)) * CANIOGAIN(x)` or the actual value of `PIN(x) = (CANOBJECTdata(x) - CANIOOFFSET(x)) * CANIOGAIN(x)`. The default value of `CANIOOFFSET(x)` is 0 and of `CANIOGAIN(x)` is 1. In general these parameters will only be useful for analogue signals. The dimensions of these arrays may be lower than the number of `CANIOLINK`s. One has to take care to use the first `CANIOLINK`s with the offset, cq gain in this case.

Also one Basic string array can be dimensioned: `CANIOCOMMAND$`. If this is done and the actual string is not empty, the optional `CANIOOFFSET` and `CANIOGAIN` parameters are ignored for that particular `CANIOLINK`. One has to be very careful using this array. In the first place every string will occupy 256 bytes of the array memory. In the second place it is not easy to fill these strings. It should be filled with a translated Basic command. How to translate a standard Basic command into a so-called tokenized is described in the `EVAL` command. If the string is built as: `CHR$(129)+"CIV"+CHR$(186)+.....(LET CIV=....)` then the value of `CIV` is used in the calculation of the result in either the `CANOBJECT` data or the IO value. Only one command can be used in a string, so no multiple commands using a ":". Commands in the `CANIOCOMMAND$()` will not be executed if a Basic program is running. Also executing direct Basic statements from the prompt when `CANIOLINKs` with `CANIOCOMMAND$()`s can give conflicts. If multiple commands are defined and would overlap in timing, the one first started will be executed and the new one ignored. So one should be careful in placing the `CANIOLINKs` in the right timing.

If the Basic parameters are dimensioned for the `CANOBJECT` (e.g. `DIM CANOBJxxDATA(10)`) this parameter will be updated during the `CANIOLINK` operation. This will always work correctly independent of the size of the FIFOs. This is not the case if this is not done. In that case the pointer to the specific FIFO will always be to the actual location, however if a new `CANOBJECT` is sent or received the location is incremented. If the size of the FIFO > 1 the data will be really actualised after a number of messages equal to the size of the FIFO. That is why it is advised to use only FIFO's of 1 message deep, if the Basic object data is not used. The advantages of not using the Basic object data is the speed (data is directly copied into the FIFO's) and more memory is left available in RAM. So for a relatively low number of `CANIOLINK`s it is better to use the Basic object data and for a high number the system performance will be better without them.

Example:

```
10 CANOPEN 500
20 DIM CANOBJ00DATA(10):CANOBJ00ID=100:CANOBJ00CTRL=1
30 DIM CANIOARRAY(4):'enugh for one CANIOLINK
40 CANOBJECT 0,1000
50 CANIOLINK 0,0,500,0,1
RUN
>
```

The program will execute the lines 10-50 and stop after that. However the started `CANOBJECT` and `CANIOLINK` will continue. They will generate a CAN message every second, with ID 100 and 1 databyte with the value of the USER button. It will stop after CTRL-C.

2.9 CANLINK

Purpose:
Link one of the CAN filters to a mask and a FIFO.

Syntax:
```
CANLINK [filter no[,enable[,maskno[,fifono]]]]
```

Comments:

`CANLINK` with no parameters will show all links.

If `DIM CANTEST(31)` has been executed before `CANLINK` the status will be in this array and can be used in a Basic program. The value is:

```
CANTEST(x) = fifono + (32 * maskno) + (256 * enable)
```

`CANLINK` with only the `filter no` disables the filter, no change in mask and FIFO

`CANLINK` with `filter no` and `enable` will enable (1) or disable(0) the link.

`CANLINK` with `filter no, enable` and `maskno` will also set the mask

`CANLINK` with `filter, enable, maskno` and `fifono` sets the complete link, including FIFO.

A filter can only be linked to one FIFO; a FIFO however can be linked to more than one filter. Before a new link can be made the former link has to be disabled.

Examples:

```
CANLINK:'Show all 32 links

CANLINK 1,1,0,1:'Link filter 1 to FIFO 1, using MASK 0

CANLINK 1,0,0,1:'Unlink (disable) it again
```

2.10 CANLOG

Purpose:
Analysis of a bus system. It can show the data on a screen/USB terminal as well as write it to a file or local RAM memory or direct to I2C.

Syntax:
CANLOG [#fileno][,format[,fifono[,period[,lineno]]]]

Comments:

All parameters are optional. If a parameter is to be used, all previous parameters should be filled in too.

- **#fileno: Fileno has to be used as in PRINT #.**
- **format: Data format, with selection of trace/object view and optional protocols**
- **fifono: 0 -31; default value: FIFO 1**
- **period: The measurement time in ms.**
- **lineno: The number of CAN messages to be logged.**

CANLOG can be stopped in five ways:

- **CTRL-C: When used on the command line this will come back to the basic prompt.**
- **ESC does the same as CTRL-C, however this can be used within a BASIC program.**
- **Keys <P>ause, <Q>uit and <R>esume, described below at CANCONTINUE.**
- **period: When period is given (>0), it will stop after this period time in ms.**
- **lineno: When lineno is given, it will stop after the no of messages.**

A combination of period and lineno is possible. It will stop on the event which comes first. If only lineno is used, period should be set to 0. See also the variables CANPERIOD and CANLINENO. If period or lineno are used <ESC> is disabled.

The fileno operates in the following way:
#0: (default) screen/USB
#1 or #2: file or serial port opened on this port
#4: same as #0
#5: #0+#1
#6: #0+#2
#7: #0+#1+#2
#8: log to RAM area defined by the basic variable CANLOGARRAY (Array should be dimensioned before, e.g. DIM CANLOGARRAY(4096)
#9: log to I2C, configured as master, with slave destination 0x60 (this address can changed by basic variable CANI2CADDRESS)
 # higher than 9 forbidden

The formats from 0 to 31 are the trace formats:

0 -7:	**The standard trace formats**
8 - 15:	**As 0 - 7 however only registration during pressing of USER button**
16 - 23:	**As 0 - 7 however triggered by SPACE key or USER button**
24 - 31:	**As 0 - 7 however, triggered by a CANbus error or a pre-defined ID**

If the triggered formats are used, the BASIC variables `CANPRETRIGGER` and `CANPOSTTRIGGER` can be used to determine the number of messages before and after the trigger moment, which are included in the log. If `CANPRETRIGGER` is not defined it is supposed to be 0 and if `CANPOSTTRIGGER` is not defined the log will continue until it is stopped by one of the other methods. The maximum value of pre-triggered messages is the depth of the buffer which is used for logging. We have two possible buffers. If `CANLOGARRAY` has not been declared, it will be FIFO which is used. The maximum number of pre-trigger messages will be 31 in this case (FIFO depth=32). If `CANLOGARRAY` has been declared this array will be used for buffering. The maximum number of pre-trigger messages will be the dimension divided by 4. So `DIM CANLOGARRAY(4096)` means up to 1024 pre-trigger messages. The dimension must be a fourfold value (4, 8, 12, 16, etc.), otherwise the messages will be read in a wrong way. Of course we cannot use #8 as `fileno` in this case. The pre-triggered messages are displayed or logged after the post-triggered messages.

If the formats 24-31 are used also an optional variable `CANTRIGGERERRCNT` is available, which has the value or the Rx error counter when the trigger occurs (1 by default). `CANTRIGGERID` and `CANTRIGGEREXTID` (for extended IDs) can be used to determine the ID to trigger on rx ID.

By default PIN(0) (The USER button on the basic hardware) is used as an external trigger switch. This can be modified by the variables `CANTRIGGERPIN` and `CANTRIGGERVALUE`. `CANTRIGGERPIN` can be any of the 6 analog inputs (PIN 1-6) or any of the 8 digital I/O's (PIN 11-18). If `CANTRIGGERPIN` has got another value it is reset to 0. The variable `CANTRIGGERVALUE` indicates the trigger value. For digital I/O's this is by default 0; if any other input is set, it will be 1. For analog inputs `CANTRIGGERVALUE` means the value of the analog input in Volts (0 - 3.3 Volt). The `CANTRIGGERPIN` is only active if the I/O line is set to digital input (`SETPIN x,2`) or analog input (`SETPIN x,1`).

The existing trace formats are:

- **0. The Basic Microchip format (16 byte per message HEX)**
- **1. Blank; format if only busload and/or error messaging are of interest**
- **2. The UNICANNER format including error information.**
- **3. The UNICANNER error only format**
- **4. The HEX string format**
- **5. The DEC string format**
- **6. The ASCII string format**
- **7. The Custom protocol format**

If format is greater than 31 the object format is chosen. The format can now be divided as follows:
<ddddddddd><r><e><f><c><ppp><s>
In the object view the upper free FIFO space is used for storage of the objects. In the small object size one FIFO location (16 bytes) is used per object. In the large object size two FIFO locations (32 bytes). In the small object view only ID, CTRL byte, databytes and no of messages are displayed. In the large object view we see in addition the last timestamp, the first timestamp, the last interval, the smallest interval and the largest interval..

<s>: **The size of the object; 0 means the default large; 1 the small size**
<ppp>: **The protocol for data:**
000: Hex ID only 001: Blank
010: Unicanner 011: Dec ID only
100: Hex string 101: Dec string
110: ASCII string 111: Custom protocol
<c>: **If 1 and custom protocol is available an extra protocol line is included**
<f>: **The switch between trace and object format; must be 1 for object.**
<ddddddddd><r><e>:
Used for optional object specification; by id, but also by data, RTR and IDE. The first d is the first databyte D1; the last is D8. r and e: RTR/IDE. Default they are all 0 , which means only ID object definition.

`CANLOG` has a timestamp on every received CAN message. The timestamp from the CAN module of the PIC32 controller is used. The default resolution is 100 uS, which can be changed by `CANOPEN`. As this is a 16 bit number, after about 6.5 sec it will start over again. In the software this is compensated: if timestamp < prev timestamp then add 65536. This works only if messages are seen in every period of 6.5 sec. Gaps in time are corrected by the CANclock. The Basic variable `CANTSCORRECTION` can be used to control this. If not defined, the default value (3) will be used, meaning both corrections will be included. If set to 0 no corrections will be made; 1 only the correction in "if timestamp<prev timestamp" is made; 2 only the correction in `CANCLOCK` is available, which is checked every 1 ms (timestamp can be wrong if more than one message come within 1 ms).

To show activity on the bus during `CANLOG`, the application LED can be used for blinking. If the variable `CANBLINK` has a value >0 the USER LED will toggle between ON and OFF at the number of messages which is given to `CANBLINK`.

If at least one of the basic variables `CANLOAD` or `CANLOADMAX` is defined, a busload measurement is done during `CANLOG`. `CANLOAD` contains the overall busload in % and `CANLOADMAX` the maximum busload during 1 period. If also the variable `CANLOADMAXTIME` is defined, this variable contains the second when the maximum busload occured. Also the variable `CANLOADACT` is available. This specifies the actual busload over the last measurement period.
The array variable `CANLOADPERIOD(n)` can be used for the actual busload in a period (default 1 sec, but can be changed by `CANSUB`). The number of values is dependent on n in `DIM CANLOADPERIOD(n)`.

CANCONTINUE can be used as following:
CANCONTINUE=0 or CANCONTINUE>3: Every time CANLOG is called the message number and the timestamp are reset. Also the complete command line is interpreted again, so you can change all parameters. If CANCONTINUE is not defined you can use <ESC> or <Q>.
CANCONTINUE=1: If CANLOG is called again after the first time, the command is not interpreted again and message number and timestamp are not reset, but incremented from the values of the previous run. Only CTRL-C will stop CANLOG in this situation.
CANCONTINUE=2: The command is not interpreted again and continued with the parameters from the previous run. If CANCONTINUE not defined <P> will do the same. In fact by using CANSUB also this situation will be entered, only now you don't have to call CANLOG again.
CANCONTINUE=3: If CANLOG is called again after the first time, message number and timestamp are not reset, but incremented from the values of the previous run. However the command is interpreted again and can be called with different parameters. If CANCONTINUE not defined <R> will do the same.

CANPERIOD and CANLINENO can be used instead of the same parameters in the command. CANTIME and CANMESSNO can be used to read out the actual measuring period in ms and the actual number of lines at the end of the CANLOG. If the <period> and/or <lineno> parameters in the command are used, they will overrule the CANPERIOD and CANLINENO variables.

If CANERRCNT and CANOVFCNT are defined they contain resp. the maximum value of the CAN RX error counter and the value of the number of buffer overflows which occured. Every detection of a buffer overflow is indicated by a missing line number. This is very useful when interrupted logs are made.

If CANENDOFLINE is not defined or zero, every line in the CANLOG trace is ended by CRLF. If CANENDOFLINE=1 the line is only ended by CR, which gives an overlay for every line on the screen. If CANENDOFLINE=2 the line is ended by 2 spaces. If CANENDOFLINE=3 no characters at all are placed at the end of the line. This parameter can be very useful if the custom protocol is used.

The variable CANEVENTINT can be used to set an event bit, which can be used in the custom protocol.

By default the script included in the license file is used for the custom protocol translation. We have the availability to use multiple scripts, which are described in the SCRIPT statement. If the scripts are small (< 4kb) and the other user programs are stored on the B-drive maximal 16 scripts fit on the A-drive. These scripts can also be custom protocol files. To change the custom protocol two actions have to be done:

1. **DIM CANSCRIPTPTR : REM this variable will get a number 0 - 15**
2. **CANSCRIPT "PROTFILE.MSC":REM this will run this file once, set CANSCRIPTPTR and set the protocol to the correct entry.**

Now CANLOG 7 will show the CANdata as defined in the file PROTFILE.MSC. In this way we can simply change the format in one program to CANopen, J1939 or any other protocol.

If a `CANLOG` has to be done continuously, but in between also other tasks, either using `CANSUB/CANRETURN` or by doing bursts of `CANLOG`'s, it is possible that the maximum of 32 messages in one FIFO is not enough for storage. The PIC32 has the possibility to store messages into a maximum of 32 FIFO's. However in the `CANLOG` statement the FIFO to be used has to be specified. This has to be linked to a filter and a mask. By default the Filter 1 is used, linked to the Mask 0 and FIFO 1. By linking the next filter, having the same value as the previous one, to the same mask and the next FIFO (`CANLINK 2,1,0,2`) the FIFO area is automatically extended with this FIFO. The same is true for every next Filter/FIFO combination. In this way it is in principle possible to use up to 32*32 = 1024 messages deep FIFO. The extended FIFO's are detected at the first execution of `CANLOG` if the linking matches the description above and all the filters have the same value. After this execution the Basic variable `CANLOGFIFO` has the number of FIFO's, which are active for this `CANLOG`. This extended FIFO option cannot be used for pre-trigger messages. Use `CANLOGARRAY` for this feature as described before.

The Timestamp correction in the multiple FIFO configuration is done in a slightly different way as described before. For a single FIFO application it is not easy to detect which messages belong to the first `CANLOG` and which to the second. That is why all messages are corrected with the timestamp (calculated by `CANCLOCK`) of the second run. In principle this is wrong, because probably there were still messages of the first run in the FIFO. If we have lost messages, this is indicated with a missing message no. at the start and not the place where the messages are lost. If we use multiple FIFO's we skip the correction with the `CANCLOCK` (so no correction if a gap of 6.5 sec occurs). This means if we start the second `CANLOG` we still use the corrected timestamp of the first run. Now all the old messages available in the FIFO's are read with corrected timestamp from the first run and when we come at the end of all FIFO's we compensate with the `CANCLOCK` and eventually skip a messageno at the right place.

Examples:

```
CANLOG 4:'Log CAN messages on screen in the HEX format

CANLOG #1,7:'Log in file #1 in the custom format

CANPRETRIGGER=10:CANPOSTTRIGER=20:CANLOG 20
'Screen log in HEX format; triggered by USER switch or SPACE;
10 messages before trigger and 20 messages after trigger

CANLOG 28:'Log in object format; For every ID you get the
number of occurrences, first and last timestamp and intervals

See also example in CANSUB.
```

2.11 CANMASK

Purpose:
Configure the masks which are used to filter the received messages.

Syntax:
```
CANMASK [<maskno[,std mask[,ext mask]]]
```

Comments:

CANMASK without parameters will show the configuration of the masks.

If DIM CANTEST(7) has been executed before CANMASK the values of CANTEST are as following:

CANTEST(2*x) = standard value of CANMASK x
CANTEST((2*x)+1) = extended value of CANMASK x

If only standard value of CANMASK is entered:
CANTEST((2*x)+1) = 999999

CANMASK with only the maskno will set the mask to all 0, meaning all bits are not relevant. maskno can be 0 - 3.

CANMASK with maskno and std mask will mask the bits in the standard part which are set to 1. It will not mask the IDE bit itself.

CANMASK with maskno, std mask and ext mask masks the 1 bits in both standard and extended part. Also the IDE bit is masked.

Please keep in mind that after changing the mask, all Rx FIFO's are reset.

Examples:

```
CANMASK:'Show all 4 masks with parameters

CANMASK 0,&H7FF:'Set Mask 0 to all bits relevant

CANMASK 1,0:'Set Mask 1 to all bits not relevant

CANMASK 2,&H7FF,&H3FFFF:'Set Mask 2 relevant for all IDs,
standard and extended
```

2.12 CANOBJECT

Purpose:
Perform a Tx or Rx message on the CANbus in the background

Syntax:
CANOBJECT [#<fileno>][,<fifono>[,<timer>]]

Comments:

Once an object has started it will continue running until it is stopped again with the same CANOBJECT command. CTRL-C will stop all CANOBJECTs. The CANOBJECT is activated by the 1 ms interrupt and running completely independent of a Basic program. One can configure the CANOBJECTs in a Basic program, together e.g. with CANIOLINKs and after the END of the program these CANOBJECTs will continue. They have to be stopped by CTRL-C before the program or another program is started. Also don't execute LOAD or SAVE when CANOBJECTs are still running.

A total of 32 CANOBJECTs (fifono = 0 to 31) can run in the background. Every CANOBJECT is linked to an unique FIFO. So by default only 2 objects are active (Object 0 for Tx and object 1 for Rx) after CANOPEN. If more objects are needed they should be configured first by the CANFIFO and CANLINK (to be active for Rx) commands. Please keep in mind that the objects are scanned sequential starting at 0 and ending as soon as an inactive FIFO has been detected. An object is inactive when the timer is set to 0.

As the objects are linked to the FIFO's it is obvious that we have Tx and Rx objects. The values of the object can be linked to Basic variables. The following variables can be declared (DIM) for this purpose: CANOBJnnID, CANOBJnnCTRL, CANOBJnnDATA(8), CANOBJnnTS. nn should be replaced by the FIFOno. In fact it is enough to declare CANOBJnnDATA(10). This will automatically use the upper 3 value as reference to the ID, CTRL and TS variables. Never write or read to the elements 8, 9 and 10 of the array because otherwise the references to the other parameters are lost.

The DATA array contains the databytes. The CTRL parameter by default the DLC, added with 64 when the RTR is activ and added with 128 when EXT is activ. The ID contains the identifier and TS the timestamp of the last object action. As the floating point numbers in Basic are only accurate up to 1000000 and the extended identifier can go up to 536870911, it has been decided that the millions of the ID are multiplied by 1000 and added to the CTRL parameter. So a message with ID 536870911 and 8 bytes of data will have: ID=870911 and CTRL=536136 (136=128+8; 536 has to be multiplied by 1000000 and added to ID).

When all Basic parameters are defined for a Tx object, the object will be sent with these parameters. Otherwise the Tx FIFO can be configured by the CANSEND command (fill only). In this way also multiple messages can be sent by one object, if the FIFO depth is larger than 1. When the Basic parameters are defined for an Rx object, they will be refreshed with every new message for this object. Please keep in mind that all parameters have to be available, otherwise the object will be handled without the Basic parameters. In fact DIM CANOBJnnDATA(10) will do the job.

The `timer` is linked to the millisecond timer as used in Basic (`TIMER`). So the minimum repeat time for an object is 1 ms. By default the timer for Tx objects is set to 1000 (Tx every sec) and for Rx objects to 1 (check for messages every 1 ms). Any other value can be given in the optional parameter for every object. The command can be used as toggle between default and off. As the default object (no FIFO specified) is object 0, one can define all objects and after that just toggle them with the command `CANOBJECT`, because timer0=0 means inactive and also the other objects in this case. If we want only object 0 inactive and all the others still running, we should give the object a high timer value, e.g. 1000000000 or simply define it without BASIC parameters and empty FIFO (no `CANSEND`).

Also single shot Tx `CANOBJECT`s are possible. The command `CANOBJECT 1xx,timer` fires the object xx (00-31) once. The timer will have a delay time of 1 to 1.000.000.000 ms. before the message is actually transmitted. After the single shot object is fired, the object remains in the list, however with an inactive status. The object can be deleted again by setting timer to 0 or recalled by `CANOBJECT 1xx,timer`.

If the `CANOBJECT` is a Rx object, the timer in `CANOBJECT 1xx,timer` will contain the Basic line no. where to jump to (an interrupt like `SETTICK`). `IRETURN` will end this subroutine. Also for Rx objects the command has to be repeated in the subroutine just before `IRETURN` for continuous receive.

The command `CANOBJECT 1xxyy,timer` will fire the object xx, when the object yy has been detected. For Tx the timer parameter contains the no. of object yy, which has to be received; for Rx objects it is the delay time. Entering `CANOBJECT 1xxyy,0` will delete the object.

It is possible to have `CANOBJECT`s on both CANports. To do that you have to activate the first CANport and after that switch to the other port and activate that port too. Now you can define the CANOBJECTs as before. We now have the following syntaxes:
`CANOBJECT 2xx,timer`: starts repetitive objects on the alternate port
`CANOBJECT 3xx,timer`: starts a single shot object on the alternate port
`CANOBJECT 3xxyy,timer`: starts a dependant object on the alternate port

As the `CANOBJECT`s work in the background it is possible to combine them with other CAN commands. However this can lead to strange situations, e.g. when working with rx-objects and the `CANLOG` command in parallel. Some messages can be achieved by the `CANLOG` command and others by the `CANOBJECT`.

If two CAN ports are available it is possible to send objects on one port and log them on the other port. Therefore it is possible to start the objects on one port and switch to the other port with `CANRESET 2`. Now we can use all the other CAN commands on the other port. At the first `CANOBJECT` command the to be used CAN port for all `CANOBJECT`s is defined. This can only be reset by going back to this port by `CANRESET 0` or `CANRESET 1`. All objects are reset in this way.

By specifying `# fileno`, the object is logged in the UNICANNER format. #0 means logged on the screen, a number #1 or #2 in a file when it is opened for OUTPUT before. Don't forget to close the file.

A counter is used to register all objects which occur, both a successful Tx as Rx. This counter can be read by the reserved command `CANOBJECT 32` (fileno and timer are irrelevant). The value of the counter is returned. If the basic variable `CANOBJCOUNTER` is defined, the value will be stored in this variable. This variable will only be refreshed after a new `CANOBJECT 32` command.

The Basic parameter `CANBLINK` can be used to toggle the green LED.

For testing the data bytes in a tx `CANOBJECT` can have values greater than 255. In this case the value has to be interpreted as following in bits: <rtdMMMmmmvvvDDDDDDDD>. DDDDDDDD is the 8 bits data to start with; vvv is the increment/decrement value (0, 1, 2, 4, 8, 16, 32, 64); mmm is the minimum value (0, 1, 2, 4, 8, 16, 32, 64); MMM is the maximum value (255, 127, 63, 31, 15, 7, 3, 1); d is the direction (0 inc., 1 dec.); t is the type of change (0 sawtooth, 1 triangle); r is random (if this bit is 1, a random value between 0 and the inc/dec value will be used for inc/dec).

Warning: As the `CANOBJECT` is linked to the millisecond timer it can influence the behaviour of the real-time clock. If the handling of the objects exceeds 1 ms, the next ms interrupt will be delayed. It is known that if logging is active that the handling exceeds 1 ms, so use logging only for testing and not for an application running in real-time.

`CANOBJECT` can also be used as a function to get the status of the `CANOBJECT`s. Description is found in the item in the functions (3.22).

Next pages show all CANOBJECT possibilities.

Example:

```
CANFIFO 0,1,1
DIM CANOBJ00DATA(10):CANOB00ID=100:CANOBJ00CTRL=1
CANOBJ00DATA(0)=100
CANOBJECT 0,100
```

This will send a message, with ID 100 and a databyte 100 every 100 ms.

Overview of CANOBJECT possibilities

The line OBJECTS in CANSTATUS shows the status of all possible 32 objects. Every object is represented by 2 hexadecimal digits. The MSB is used for active/non-active. Active objects have a status of 80 up to FF. In this table you can see how the status has been reached.

The columns are described as following:

optional log specification: none(no logging), #0 (screen), #1 or #2 (file)

objectno.: xx (standard), 1xx (single-shot), 1xxyy (dependent of yy)
idem on alternate port: 2xx (standard), 3xx (single-shot), 3xxyy (dependent of yy)

Rx/Tx: (parameter set by CANFIFO)

optional BASIC parameters: yes or no

Single shot active: yes or no (parameter automatically changed after single shot occurred)

Timer function: timer (repeat time in ms), counter (no of messages),
delay (delay time in ms) or lineno (jump to BASIC subroutine)

CANport: standard or alternate (std or alt)

status	opt. logspec.	objectno.	Rx/Tx	opt. BASIC	Single shot	Timer function	CANport
80	none	xx	Tx	no	no	timer	std
81	none	1xxyy	Tx	no	no	counter	std
82	none	1xx	Tx	no	no	delay	std
83	none	1xx	Tx	no	yes	delay	std
84	none	2xx	Tx	no	no	timer	alt
85	none	3xxyy	Tx	no	no	counter	alt
86	none	3xx	Tx	no	no	delay	alt
87	none	3xx	Tx	no	yes	delay	alt
88	#0	xx	Tx	no	no	timer	std
89	#0	1xxyy	Tx	no	no	counter	std
8A	#0	1xx	Tx	no	no	delay	std
8B	#0	1xx	Tx	no	yes	delay	std
8C	#0	2xx	Tx	no	no	timer	alt
8D	#0	3xxyy	Tx	no	no	counter	alt
8E	#0	3xx	Tx	no	no	delay	alt
8F	#0	3xx	Tx	no	yes	delay	alt
90	#1	xx	Tx	no	no	timer	std
91	#1	1xxyy	Tx	no	no	counter	std
92	#1	1xx	Tx	no	no	delay	std
93	#1	1xx	Tx	no	yes	delay	std
94	#1	2xx	Tx	no	no	timer	alt
95	#1	3xxyy	Tx	no	no	counter	alt
96	#1	3xx	Tx	no	no	delay	alt
97	#1	3xx	Tx	no	yes	delay	alt
98	#2	xx	Tx	no	no	timer	std
99	#2	1xxyy	Tx	no	no	counter	std
9A	#2	1xx	Tx	no	no	delay	std
9B	#2	1xx	Tx	no	yes	delay	std
9C	#2	2xx	Tx	no	no	timer	alt
9D	#2	3xxyy	Tx	no	no	counter	alt
9E	#2	3xx	Tx	no	no	delay	alt
9F	#2	3xx	Tx	no	yes	delay	alt

status	opt. logspec.	objectno.	Rx/Tx	opt. BASIC	Single shot	Timer function	CANport
A0	none	xx	Tx	yes	no	timer	std
A1	none	1xxyy	Tx	yes	no	counter	std
A2	none	1xx	Tx	yes	no	delay	std
A3	none	1xx	Tx	yes	yes	delay	std
A4	none	2xx	Tx	yes	no	timer	alt
A5	none	3xxyy	Tx	yes	no	counter	alt
A6	none	3xx	Tx	yes	no	delay	alt
A7	none	3xx	Tx	yes	yes	delay	alt
A8	#0	xx	Tx	yes	no	timer	std
A9	#0	1xxyy	Tx	yes	no	counter	std
AA	#0	1xx	Tx	yes	no	delay	std
AB	#0	1xx	Tx	yes	yes	delay	std
AC	#0	2xx	Tx	yes	no	timer	alt
AD	#0	3xxyy	Tx	yes	no	counter	alt
AE	#0	3xx	Tx	yes	no	delay	alt
AF	#0	3xx	Tx	yes	yes	delay	alt
B0	#1	xx	Tx	yes	no	timer	std
B1	#1	1xxyy	Tx	yes	no	counter	std
B2	#1	1xx	Tx	yes	no	delay	std
B3	#1	1xx	Tx	yes	yes	delay	std
B4	#1	2xx	Tx	yes	no	timer	alt
B5	#1	3xxyy	Tx	yes	no	counter	alt
B6	#1	3xx	Tx	yes	no	delay	alt
B7	#1	3xx	Tx	yes	yes	delay	alt
B8	#2	xx	Tx	yes	no	timer	std
B9	#2	1xxyy	Tx	yes	no	counter	std
BA	#2	1xx	Tx	yes	no	delay	std
BB	#2	1xx	Tx	yes	yes	delay	std
BC	#2	2xx	Tx	yes	no	timer	alt
BD	#2	3xxyy	Tx	yes	no	counter	alt
BE	#2	3xx	Tx	yes	no	delay	alt
BF	#2	3xx	Tx	yes	yes	delay	alt

status	opt. logspec.	objectno.	Rx/Tx	opt. BASIC	Single shot	Timer function	CANport
C0	none	xx	Rx	no	no	timer	std
C1	none	1xxyy	Rx	no	no	counter	std
C2	none	1xx	Rx	no	no	delay	std
C3	none	1xx	Rx	no	yes	delay	std
C4	none	2xx	Rx	no	no	timer	alt
C5	none	3xxyy	Rx	no	no	counter	alt
C6	none	3xx	Rx	no	no	delay	alt
C7	none	3xx	Rx	no	yes	delay	alt
C8	#0	xx	Rx	no	no	timer	std
C9	#0	1xxyy	Rx	no	no	counter	std
CA	#0	1xx	Rx	no	no	delay	std
CB	#0	1xx	Rx	no	yes	delay	std
CC	#0	2xx	Rx	no	no	timer	alt
CD	#0	3xxyy	Rx	no	no	counter	alt
CE	#0	3xx	Rx	no	no	delay	alt
CF	#0	3xx	Rx	no	yes	delay	alt
D0	#1	xx	Rx	no	no	timer	std
D1	#1	1xxyy	Rx	no	no	counter	std
D2	#1	1xx	Rx	no	no	delay	std
D3	#1	1xx	Rx	no	yes	delay	std
D4	#1	2xx	Rx	no	no	timer	alt
D5	#1	3xxyy	Rx	no	no	counter	alt
D6	#1	3xx	Rx	no	no	delay	alt
D7	#1	3xx	Rx	no	yes	delay	alt
D8	#2	xx	Rx	no	no	timer	std
D9	#2	1xxyy	Rx	no	no	counter	std
DA	#2	1xx	Rx	no	no	delay	std
DB	#2	1xx	Rx	no	yes	delay	std
DC	#2	2xx	Rx	no	no	timer	alt
DD	#2	3xxyy	Rx	no	no	counter	alt
DE	#2	3xx	Rx	no	no	delay	alt
DF	#2	3xx	Rx	no	yes	delay	alt

status	opt. logspec.	objectno.	Rx/Tx	opt. BASIC	Single shot	Timer function	CANport
E0	none	xx	Rx	yes	no	timer	std
E1	none	1xxyy	Rx	yes	no	counter	std
E2	none	1xx	Rx	yes	no	delay	std
E3	none	1xx	Rx	yes	yes	delay	std
E4	none	2xx	Rx	yes	no	timer	alt
E5	none	3xxyy	Rx	yes	no	counter	alt
E6	none	3xx	Rx	yes	no	delay	alt
E7	none	3xx	Rx	yes	yes	delay	alt
E8	#0	xx	Rx	yes	no	timer	std
E9	#0	1xxyy	Rx	yes	no	counter	std
EA	#0	1xx	Rx	yes	no	delay	std
EB	#0	1xx	Rx	yes	yes	delay	std
EC	#0	2xx	Rx	yes	no	timer	alt
ED	#0	3xxyy	Rx	yes	no	counter	alt
EE	#0	3xx	Rx	yes	no	delay	alt
EF	#0	3xx	Rx	yes	yes	delay	alt
F0	#1	xx	Rx	yes	no	timer	std
F1	#1	1xxyy	Rx	yes	no	counter	std
F2	#1	1xx	Rx	yes	no	delay	std
F3	#1	1xx	Rx	yes	yes	delay	std
F4	#1	2xx	Rx	yes	no	timer	alt
F5	#1	3xxyy	Rx	yes	no	counter	alt
F6	#1	3xx	Rx	yes	no	delay	alt
F7	#1	3xx	Rx	yes	yes	delay	alt
F8	#2	xx	Rx	yes	no	timer	std
F9	#2	1xxyy	Rx	yes	no	counter	std
FA	#2	1xx	Rx	yes	no	delay	std
FB	#2	1xx	Rx	yes	yes	delay	std
FC	#2	2xx	Rx	yes	no	timer	alt
FD	#2	3xxyy	Rx	yes	no	counter	alt
FE	#2	3xx	Rx	yes	no	delay	alt
FF	#2	3xx	Rx	yes	yes	delay	alt

2.13 CANOPEN

Purpose:
Set a CANport to operational.

Syntax:
CANOPEN [speed[,special[,ts resolution]]]

Comments:

When CANOPEN runs for the first time and CANFIFO has not been executed yet, it will set two FIFO's. The first one for sending messages, which is 2 messages deep, the second one for receiving, 32 messages deep. Standard the receiving filter is set to receiving all messages. Standard also the timestamp is enabled and set to a resolution of 100 uS.

speed is by default the parameter which specifies the CAN bitrate in bits/sec. The range is 25000 up to 1000000. It is also possible to specify the bitrate in kbits/sec (25 - 1000).
The default value of speed is 0. This means the value is not changed. In this way one can change the special or ts resolution parameter without having to know the actual speed.
The values 1-7 are used for autobauding and the values 8-24 to change configuration and control registers of the CAN controller in a very specific way (experienced users only). See table on the next page for all possible speed values.

special is used to open the CAN-port in a non-standard way. Speed must be specified in this case. The value of special can be 0 to 255 (8 bits) where all bits have their specific meaning: <oowtmsjj>.
The 2 LSB's (jj) are the two bits to configure the SJW (synchronisation jump width):
00 : 1 clock cycle (default) 01 : 2 clock cycles
10 : 3 clock cycles 11 : 4 clock cycles
The next one (s) determines the number of samples:
0 : 1 sample (default)1 : 3 samples
The next one (m) determines the position in the period for sampling:
0 : 75% (default) 1 : 80%
The next one (t) determines the availability of timestamps:
0 : timestamps available (default)
1 : timestamps off
The next one (w) enables or disables the wakeup facility
0 : disabled (default) 1 : enabled
The two most significant bits (oo) determine the operation mode:
00 : normal operation (default)
01 : loopback mode (also own tx messages can be received)
10 : listen-only mode (no errorframes; no ACK)
11 : all message mode (all messages, also fault ones, are received)

The third optional parameter ts resolution (special must also be specified in this case) sets the timestamp resolution. This value ranges from 1 uSec minimal to 800 uSec maximal. Default value is 100 uSec.

Values of the <speed> parameter in CANOPEN

VALUE	DESCRIPTION	REGISTERS
0	Value is not changed	
1-6	Bitrate test 1-2 during 1 sec/bitrate; 3-4 during 0.1 sec; 5-6 during 10 sec	CANCFG for BTR
1, 3, 5	Testing predefined bitrates with display of the process *	CANCON for:
2, 4, 6	As above without display; to be used within a Basic program *	- Listen Only
7	Testing all possible values **	- SJW = 4
8	Increment PRSEG	CANCFG
9	Decrement PRSEG	CANCFG
10	Increment TSEG1PH	CANCFG
11	Decrement TSEG1PH	CANCFG
12	Increment TSEG2PH	CANCFG
13	Decrement TSEG2PH	CANCFG
14	Increment BRP	CANCFG
15	Decrement BRP	CANCFG
16	Increment measuring point (inc PRSEG or TSEG1PH / dec TSEG2PH)	CANCFG
17	Decrement measuring point (dec PRSEG or TSEG1PH / inc TSEG2PH)	CANCFG
18	Toggle TSEG2PHTS on/off	CANCFG
19	Set ABAT (reset all pending Tx)	CANCON
20	Increment DeviceNet Filter	CANCON
21	Decrement Devicenet Filter	CANCON
22	Toggle SIDLE bit	CANCON
23	Reserved for future use	
24	Toggle CAN module ON/OFF	CANCON
25 -1000	Set bitrate in kbits/sec	CANCFG+CANCON
1001 - 24999	Illegal	
25000 - 1000000	Set bitrate in bits/sec	CANCFG+CANCON
> 1000000	Illegal	

*** Bitrates are defined by 4 ranges starting high and taking 50% of it for the next measurement: 1000/500/250/125/62.5, 800/400/200/100/50, 666.7/333.3/166.7/83.3/41.7, 600/300/150/75/37.5. If a bitrate is found the process is stopped, the found bitrate is set and the mode is set to NORMAL.**

**** Bitrate is defined by 1000, decreased with 1% for every next measurement (time 0.1 sec). As soon as bitrate(s) are found the process is stopped. The bitrate is not set yet. This has to be done with a new `CANOPEN` command.**

Examples:

```
CANOPEN 500:'Open CANbus on 500 kbit/sec

CANOPEN 500000:'The same

CANOPEN 1:'Autobauding on most used bitrates

CANOPEN 7:'Autobauding on all bitrates

CANOPEN 500,128:'Open 500 kbit/sec listen only mode

CANOPEN 500,128,10:'The same; timestamp resolution 10us
```

2.14 CANPHYS

Purpose:
Enable physical (analogue) measurements on a CANbus

Syntax:
`CANPHYS [parameter[,no]]`

Comments:

`CANPHYS` is a statement not only using the CANports, but also most of the Arduino I/O pins are involved. That is why it can only be used with an additional Arduino hardware extension. If this hardware is included please remember that you cannot use the I/O pins for other purposes.

The statement uses the second CANport for measurement. The first one can be used to generate messages, which can be evaluated on the second one. The measurement port has the connection to the additional I/O ports. Both CAN High and CAN Low can be measured on their physical analog values. Timing is measured at the CAN Rx pin of the CAN transceiver chip. Both the edges to dominant and to recessive are determined in real time. The CAN timer is used to have an accurate measurement of the periods between the edges. In this way we get an exact bit-pattern in real time during the CAN message.

`CANPHYS parameter` consists of 16 bits, each representing an on/off parameter, used at the start of the command, during execution or at the end of the command. We used all these parameters to make the command as flexible as possible. The Basic program has to be written in such a way that the command is used in an optimal way. The bits are represented by: <lhLH CDSF EeNR 21PX>, where:

l = Do the analog measurement on CAN Lo during recessive situation
h = Do the analog measurement on CAN Hi during recessive situation
L = Do the analog measurement on CAN Lo during dominant situation
H = Do the analog measurement on CAN Hi during dominant situation

C = Calibrate before the actual measurement is started
D = Delay the analog measurement depending on the bitrate
S = Start analog measurements after SOF; normally started after the arbitration period
F = Convert relevant values to floats after the measurement, which can be used in Basic

E = Eliminate analog values which are out of range, before statistics are calculated
e = Stop measurement as soon as an error frame is detected on the bus
N= Don't check for new SOF; can be used when only one message (ID) is used
R = Recessive edges skipped; to be used at very high bitrates

2 = Higher bit to set minimum no of identical bits
1 = Lower bit to set minimum no of identical bits
P = Do the measurement on the message for pattern only
X = An extended identifier is expected; used during measurement and calculation

The analog measurements are triggered by an edge on the Rx pin of the CAN transceiver chip. Both edges (0 -> 1 and 1 -> 0) are determined by default. For high speeds it is possible to skip the recessive edge (0 -> 1) by setting the <R> option. Measurements are done one by one. E.g. if options <H>, <L>, <h>, <l> and <P> are set. The first message is evaluated on CAN Hi dominant, second CAN Lo dominant, third CAN Hi recessive, fourth CAN Lo recessive and the fifth pattern only.

At the start of the `CANPHYS` command the CAN Hi and CAN Lo voltages are also measured and stored. These measurements can be used as a reference. If the bus is idle (recessive) during this measurement, also a calibration can be done by setting the <C> parameter. As the CAN Hi measurement is done through a voltage divider (⅕), the calculation factor (1.2 by default) is calculated in this calibration. If all other parameters are 0, it is possible to do a calibration measurement only.

The analog measurement is started with a short delay after the edge. At lower bitrates it is better to have a longer delay, because then we have a more stable signal. This can be done by setting the <D> parameter.

Normally we start the analog measurements after the arbitration phase in the message. Therefore we have to set the <X> parameter if we want to check a message with an extended identifier. It is possible to start the analog measurements directly after the Start Of Frame (SOF), by setting the <S> parameter.

At the end of the measurement, statistics are started to calculate maximum, minimum and mean values. Also the standard deviation is calculated. Due to glitches sometimes an error can occur during the analog measurement. We can skip the "out of range" measurements by setting the <E> parameter. It is also possible to skip the measurements, in which the bus remains only for 1, 2 or 3 bits in a certain state. If <1> and <2> are both 0, all measurements are taken into account. If <2><1> is 01, all single bits are skipped. If <2><1> is 10 also the only 2 identical bits are skipped and if 11 also the 3 identical bits are skipped.

When we want to measure the signals of one specific message we will set a filter to that message. However normally also other messages (other identifiers) will be transmitted over the bus. If the message does not fulfill the filter parameters, the CAN controller will not recognise the message. In the measurement we have to skip such a message. Therefore we are always looking for a new message starting (new SOF). If this occurs and the previous message was not recognised by the CAN controller, normally all measurements are skipped and a new measurement is started. If we are just testing with one identifier we can disable the checking for a new SOF by setting the <N> parameter. This gives a better reliability at higher speeds.

In some circumstances also error frames can occur on the bus. By default there is no checking on errors. By setting <e> the checking is enabled and the test stopped when an error frame is detected. In the bit pattern one can see at which position in the frame the error occurred.

On the hand of the edges and the timer values, the bit pattern of the message is evaluated, including the stuff bits and CRC. The bit pattern can be read at the positions which will be shown later. If the evaluation is correct one of the bits in the array will be set to 1, otherwise it will be 0. If the <P> is set during this last measurement also the red LED will be set, if the message is not correct.

Normally one can read all measurements from memory byte by byte by the Basic PEEK statement. If <F> is set, the statistical values of all measurements are converted to floats. If this is done the original actual measurements are overwritten by these float values.

If <P> is set, an extra measurement is performed for the bit pattern.

`CANPHYS` has an optional second parameter, called `no`. If this parameter has a value of more than 1, the measurement is repeated `no` of times. After every measurement the statistics are calculated again. At the end the minimum, maximum, mean and standard deviation values are calculated over the whole range of measurements.

The Basic variable `CANPHYSARRAY` is created automatically by the `CANPHYS` command. This is an array of 128 values. So in fact it does a `DIM CANPHYSARRAY(128)`. You can set this Basic statement also before you activate the `CANPHYS` command, but never set the dimension lower than 128, otherwise the system will crash. This array will be used for the results of the `CANPHYS` measurement. So we have in fact 128 floats (if calculated to floats) or 512 bytes. The tables at the end of this command show both the byte values as well as the float values.

Two other optional Basic variables can be defined, `CANPHYSOK` and `CANPHYSADDRESS`. `CANPHYSADDRESS` is needed if we want to read the values as bytes with the `PEEK` statement. After `DIM CANPHYSADDRESS`, any byte value can be read by:
`x=PEEK(&HA001,CANPHYSADDRESS+n)`.

`CANPHYSOK` has the global result of the `CANPHYS` measurement after the measurement:
0 = Voltages at start out of range; probably not a correct connection to a CANbus
1 = Measurement OK
2 = Dominant level detected during calibration (only possible when <C> is set.
3 = Calibration only done. This means <C> was the only parameter which was set.
4 = Stop by STOP switch. This is the hardware stop switch.
5 = Stop by Error frame
6 = Message corrupt. This can only if <P> is set
7 = Edge count overrun. This means that a new SOF is not detected.
8 = Analog values not correctly registered.

The `CANPHYS` measurement has three phases during execution:

Phase 1: The pre-measurement
The voltages of CAN Hi and CAN Lo are measured. An optional calibration is performed. All interrupts of timers, VGA and USB are disabled. The USB connection on the host has to be switched off also. Now we will wait for the first CAN message received by the CANcontroller. This message is used for synchronisation.

Phase 2: The actual measurement
The message is received. The measurements are done depending on the parameters. This is done until the second message is received by the CANcontroller. Values are overwritten if they come from other messages than the selected one.

Phase 3: The post-measurement
The message is received for the second time. Values are frozen now. Statistics for maximum, minimum, mean and standard deviation are calculated. Optionally the message is calculated and compared with the values of the CANcontroller. Optional values are calculated to floats. The interrupts for timer, VGA and USB are enabled again. All Basic values are available now.

Remark: Because we switch off the USB interrupts, the connection with the host is also lost. That is why we should disconnect a terminal program on the host when the `CANPHYS` command is executed and set up a new connection after the command has ended.

The `CANPHYS` command is a quite complex issue, with a lot of possibilities. It should only be used directly by CAN experts. SI-Kwadraat delivers a set of Basic example programs to be used with it. These programs can be edited or extended by the user to satisfy his or her needs.

Table of byte values of the results of `CANPHYS` command

	y=0;z=8	y=1;z=9	y=2;z=a	y=3;z=b	y=4;z=c	y=5;z=d	y=6;z=e	y=7;z=f
0x00y	CAN0 timer LSB	CAN0 timer MSB	First analog- LSB	value MSB	Second LSB	value MSB	*Etc.*	*Etc.*
0x00z	*Analog*	*values*	*continue*	*up to*	*0x051*	*inclusive*		
0x01y	*If 1*	*voltage is*	*measured:*	*Up to 40*	*values are*	*registered.*		
0x01z	*If 2*	*voltages are*	*measured:*	*Up to 20*	*values per*	*measurement*	*are*	*registered*
0x02y	*If 3*	*voltages are*	*measured:*	*Up to 13*	*values per*	*measurement*	*are*	*registered*
0x02z	*If 4*	*voltages are*	*measured:*	*Up to 10*	*values per*	*measurement*	*are*	*registered*
0x03y								
0x03z	*The order of*	*measurement*	*is:*	*CAN-Lo recessive*	*CAN-Hi recessive*	*CAN-Lo dominant*	*CAN-Hi dominant*	
0x04y	*Value*	*CAN-Lo:*	*0 = 0 volt;*	*1023 =*	*3.3 volt*			
0x04z	*Value*	*CAN-Hi*	*0 = 0 volt*	*1023 =*	*~ 4.0 volt*			
0x05y	Last analog- LSB	value MSB	Analog - LSB	value ACK MSB	Max. value - LSB	CAN-Hi 0 MSB	Min. value - LSB	CAN-Hi 0 MSB
0x05z	Mean value- LSB	CAN-Hi 0 MSB	Std. Dev.- LSB	CAN-Hi 0 MSB	Max. value - LSB	CAN-Lo 0 MSB	Min. value - LSB	CAN-Lo 0 MSB
0x06y	Mean value- LSB	CAN-Lo 0 MSB	Std. Dev.- LSB	CAN-Lo 0 MSB	Max. value - LSB	CAN-Hi 1 MSB	Min. value - LSB	CAN-Hi 1 MSB
0x06z	Mean value- LSB	CAN-Hi 1 MSB	Std. Dev.- LSB	CAN-Hi 1 MSB	Max. value - LSB	CAN-Lo 1 MSB	Min. value - LSB	CAN-Lo 1 MSB
0x07y	Mean value- LSB	CAN-Lo 1 MSB	Std. Dev.- LSB	CAN-Lo 1 MSB	Mess. error If 1	Time offset In uS	CAN-Hi Cor LSB	rect * 1000 MSB
0x07z	Calibration - LSB	CAN-Hi MSB	Calibration - LSB	CAN-Lo MSB	No. of edges	No. of bits in message	No. of stuff bits	Total no. of measurement

	y=0;z=8	y=1;z=9	y=2;z=a	y=3;z=b	y=4;z=c	y=5;z=d	y=6;z=e	y=7;z=f
0x08y	Timestamp SOF = 0	Timestamp 2nd edge	Timestamp 3rd edge	*Etc.*	*Etc.*	| |	| |	| |
0x08z	| |	| |	| |	| |	| |	| |	| |	| |
0x09y	*Timestamps*	*are in uS.*	| |	| |	| |	| |	| |	| |
0x09z	| |	| |	| |	| |	| |	| |	| |	| |
0x0ay	*Maximum*	*128 edges*	*can occur*	*in one*	*message*	| |	| |	| |
0x0az	| |	| |	| |	| |	| |	| |	| |	| |
0x0by	| |	| |	| |	| |	| |	| |	| |	| |
0x0bz	| |	| |	| |	| |	| |	| |	| |	| |
0x0cy	| |	| |	| |	| |	| |	| |	| |	| |
0x0cz	| |	| |	| |	| |	| |	| |	| |	| |
0x0dy	| |	| |	| |	| |	| |	| |	| |	| |
0x0dz	| |	| |	| |	| |	| |	| |	| |	| |
0x0ey	| |	| |	| |	| |	| |	| |	| |	| |
0x0ez	| |	| |	| |	| |	| |	| |	| |	| |
0x0fy	| |	| |	| |	| |	| |	| |	| |	| |
0x0fz	| |	| |	| |	| |	| |	| |	| |	End of timestamps

	y=0;z=8	y=1;z=9	y=2;z=a	y=3;z=b	y=4;z=c	y=5;z=d	y=6;z=e	y=7;z=f
0x10y	*Start of*	*Bit pattern*	*CAN mess.,*	*including*	*Stuff bits*			
0x10z								
0x11y								
0x11z								
0x12y								
0x12z								
0x13y								
0x13z								
0x14y								
0x14z								
0x15y								
0x15z								
0x16y								
0x16z								
0x17y								
0x17z								

	y=0;z=8	y=1;z=9	y=2;z=a	y=3;z=b	y=4;z=c	y=5;z=d	y=6;z=e	y=7;z=f
0x18y	| |	| |	| |	| |	| |	| |	| |	| |
0x18z	| |	| |	| |	| |	| |	| |	| |	| |
0x19y	| |	| |	| |	| |	| |	| |	| |	| |
0x19z	| |	| |	| |	| |	| |	| |	| |	*End of Bit pattern*
0x1ay	CAN mess ID LSB	CAN mess ID byte 2	CAN mess ID byte 3	CAN mess ID MSB	CAN mess CTRL byte	CAN mess Data 1	CAN mess Data 2	CAN mess Data 3
0x1az	CAN mess Data 4	CAN mess Data 5	CAN mess Data 6	CAN mess Data 7	CAN mess Data 8	CAN mess. timestamp		
0x1by	Float value-	Maximum-	Time-	Shift	Float value-	Minimum-	Time-	Shift
0x1bz	Float value-	Total	Time-	Shifts	Float value-	Total -	Squares of-	Time Shifts
0x1cy	CAN Hi- LSB	Dominant- 2nd byte	Total- 3rd byte	No. values MSB	CAN Hi- LSB	Dominant- 2nd byte	Total- 3rd byte	Values MSB
0x1cz	CAN Hi- LSB	Recessive- 2nd byte	Total- 3rd byte	Squares MSB	CAN Hi- Maximum-	Dominant Value	CAN Hi- Minimum-	Dominant Value
0x1dy	CAN Lo- LSB	Dominant- 2nd byte	Total- 3rd byte	No. values MSB	CAN Lo- LSB	Dominant- 2nd byte	Total- 3rd byte	Values MSB
0x1dz	CAN Lo- LSB	Dominant- 2nd byte	Total- 3rd byte	Squares MSB	CAN Lo- Maximum-	Dominant Value	CAN Lo- Minimum-	Dominant Value
0x1ey	CAN Hi- LSB	Recessive- 2nd byte	Total- 3rd byte	No. values MSB	CAN Hi- LSB	Recessive- 2nd byte	Total- 3rd byte	Values MSB
0x1ez	CAN Hi- LSB	Recessive- 2nd byte	Total- 3rd byte	Squares MSB	CAN Hi- Maximum-	Recessive Value	CAN Hi- Minimum-	Recessive Value
0x1fy	CAN Lo- LSB	Recessive- 2nd byte	Total- 3rd byte	No. values MSB	CAN Lo- LSB	Recessive- 2nd byte	Total- 3rd byte	Values MSB
0x1fz	CAN Lo- LSB	Recessive- 2nd byte	Total- 3rd byte	Squares MSB	CAN Lo- Maximum-	Recessive Value	CAN Lo- Minimum-	Recessive Value

Table of float values of the results of `CANPHYS` command

	y=0;z=8	y=1;z=9	y=2;z=a	y=3;z=b	y=4;z=c	y=5;z=d	y=6;z=e	y=7;z=f
0x00y	CAN-Hi 0 maximum	CAN-Hi 0 minimum	CAN-Hi 0 average	CAN-Hi 0 Std. dev.	CAN-Lo 0 maximum	CAN-Lo 0 minimum	CAN-Lo 0 average	CAN-Lo 0 Std. dev.
0x00z	CAN-Hi 1 maximum	CAN-Hi 1 minimum	CAN-Hi 1 average	CAN-Hi 1 Std. dev.	CAN-Lo 1 maximum	CAN-Lo 1 minimum	CAN-Lo 1 average	CAN-Lo 1 Std. dev.
0x01y	Last ACK voltage	Calibration CAN-Hi	Calibration CAN-Lo	CAN-Hi correction	Timing correction	Timing error	Time difference	No. of edges
0x01z	No. of Bits	No. of Stuff bits	No. of Analog val.	Message Timestamp	Message ID	Message ID ext.	Message CTRL	Message CRC
0x02y	Message Data 1	Message Data 2	Message Data 3	Message Data 4	Message Data 5	Message Data 6	Message Data 7	Message Data 8
0x02z	Time Shift Maximum	Time Shift Minimum	Time Shift Average	Time Shift Std. Dev.	Not used	Not used	Not used	Not used
0x03y	Not used	Not used	Not used	Not used	Not used	Not used	Not used	Not used
0x03z	Not used	Not used	Not used	Not used	Not used	Not used	Not used	Not used
0x04y	Not used	Not used	Not used	Not used	Not used	Not used	Not used	Not used
0x04z	Not used	Not used	Not used	Not used	Not used	Not used	Not used	Not used
0x05y	Not used	Not used	Not used	Not used	Not used	Not used	Not used	Not used
0x05z	Not used	Not used	Not used	Not used	Not used	Not used	Not used	Not used
0x06y	Not used	Not used	Not used	Not used	Not used	Not used	Not used	Not used
0x06z	Not used	Not used	Not used	Not used	Not used	Not used	Not used	Not used
0x07y	Not used	Not used	Not used	Not used	Not used	Not used	Not used	Not used
0x07z	Not used	Not used	Not used	Not used	Not used	Not used	Not used	Not used

2.15 CANRCV

Purpose:
Read one message from an Rx FIFO

Syntax:
`CANRCV [id[,type[,id_ext[,len[,data(),ok]]]]]`

Comments:
`type` **consists of 16 bits (<g><ss><ttttt><nnnnn><o><r><e>), where**

- **<g> = global type; <g>=1 (set before `CANRCV`) to receive the message in the format specified by <ss>; <g>=0 to receive only <e> in <type>.**
- **<ss> = specific global type:**
  - **00 The filter no. which was hit for this message included in <ttttt>**
  - **01 The 5 LSB's of timestamp included in <ttttt>**
  - **10 The 5 MSB's of timestamp included in <ttttt>**
  - **11 The complete 16 bit timestamp in type (all other info lost)**
- **<ttttt> = The result of ss**
- **<nnnnn> = The Fifo no.**
- **<o> = Overflow flag <o>=1 when the overflow flag was set in the FIFO**
- **<r> = RTR bit; when <r>=0 a data frame is received; <r>=1 an RTR frame**
- **<e> = Extension bit; when <e>=1 the <id_ext> contains the 18 LSB of ID, if id_ext available.**

All parameters in `CANRCV` are opti onal. If not specified the default parameters rx… are dimensioned automatically. If specified by a variable, this variable has to be dimensioned by the user.

0 parameters: `dim rxid; dim rxtype; dim rxide; dim rxdlc; dim rxdata(8); dim rxok.`
1 parameter: id is specified; `dim rxtype; dim rxide; dim rxdlc; dim rxdata(8); dim rxok.`
2 parameters: id and type are specified; `dim rxide; dim rxdlc; dim rxdata(8); dim rxok`
3 parameters: id, type and len are specified; `dim rxdata(8); dim rxok` **(no id split)**
4 parameters: id, type, len and data are specified; `dim rxok` **(no id split)**
5 parameters: id, type, len, data and ok are specified (no id split)
6 parameters: id, type, ext id, len, data and ok are specified

The Basic variable `CANRXERRCNT` will be updated if it is used in Basic. It contains the actual status of the Rx error counter.

Example:

```
CANRCV
IF rxok=0 THEN PRINT "No messages received" ELSE PRINT
rxid;rxide;rxtype;rxdlc;rxdata(0)
```

2.16 CANREG

Purpose:
Read directly a CAN register or FIFO entry

Syntax:
CANREG [regno]

Comments:

CANREG shows the hex value in the CAN register(s) with regno if regno < 181. If DIM CANTEST(1) has been executed before CANREG CANTEST(0) will get the LSB and CANTEST(1) the MSB of the register value.

CANREG will set the CAN register (regno - 500) to the values of CANTEST(0) and CANTEST(1) if 499 < regno < 681.

CANREG shows the hex value in the FIFO register(s) with (regno - 1000) if 999 < regno < 2024. If DIM CANTEST(3) has been executed before CANREG CANTEST(0) - CANTEST(3) will get the values of the FIFO registers

CANREG will set the FIFO register (regno - 3000) to the values of CANTEST(0) - CANTEST(3) if 2999 < regno < 4024.

See Microchip datasheet for details about the CAN and FIFO registers.

To be used by experienced users only!

2.17 CANREPLAY

Purpose:
Transmit logged messages on a CANbus

Syntax:
CANREPLAY [#fileno[,fifono[,format]]]

Comments:

The CANREPLAY command can be used to send a logged file on the CANbus again. The log file must be in the UNICANNER format (format 2 in CANLOG).

The file opened for input by fileno (1 by default) will be sent on the CANbus, using fifono (0 by default).

fileno can also be 2 (file opened as #2) or 8 (data from CANLOGARRAY)

If fifono is specified it is possible to specify also the format parameter. If this is done the file is logged to the screen in the format as it has been specified in CANLOG. Only formats 0 - 7 are allowed. If format > 7 then no screen logging is performed.

The Basic parameters CANPERIOD, CANLINENO, CANTIME, CANMESSNO, CANCONTINUE, CANENDOFLINE and CANBLINK can be used as in CANLOG.

<ESC> or <CTRL-C> can be used to stop the CANREPLAY command.

Example:

```
CANREPLAY #1,1,4
```

The logfile opened for input as #1 is sent through FIFO 1 and the messages are displayed on the screen in format 4 (Hexadecimal)

2.18 CANRESET

Purpose:
Reset a CAN controller to its default values or change the active CANbus

Syntax:
CANRESET [canport]

Comments:

CANRESET will initialise the complete CAN configuration again. So if you change the buffer size or the protocol in the license file, the new parameters will be activated after this command. The command however will not reinitialize the memory. So if you started with a buffer size of 1024 and go back to 64, you also have to give the Basic command ERASE CANMESSAGEFIFOAREA or CLEAR to free the memory again.

CANRESET can be executed with the optional parameter <canport>.

CANRESET 0 will disable CAN port 1 and activate CAN port 0. This is already the default CAN port and this makes only sense if you want to switch from CAN port 1 to 0 again.

CANRESET 1 disables the standard CAN port 0 and enables the CAN port 1. All the other CAN commands are now available on CAN port 1.

CANRESET 2 can only be used if either CAN port 0 or CAN port 1 is already activated by the CANOPEN command. CANRESET 2 will automatically detect which port is already running and will now reset to the other port. In this case nothing can be changed on the first port anymore. The 2nd port can be configured now.

See also CANBRIDGE and CANOBJECT. These are the only two commands which benefit from the CANRESET 2 option.

Example:

```
CANRESET
CANRESET 1
CANOPEN 500
CANRESET 2
CANOPEN 500
```

CAN port 1 is opened in the background and CAN port 0 on the foreground. Both ports run at 500 kb/sec.

2.19 CANRETURN

Purpose:
Return to the calling command after executing the CAN subroutine

Syntax:
CANRETURN

Comments:

CANRETURN will return to the active CAN command after the subroutine which was entered by CANSUB or by CANINT is finished.

2.20 CANSCRIPT

Purpose:
Execute a script from a file on the A-drive

Syntax:
```
CANSCRIPT filename[, #fileno]
CANSCRIPT scriptpointer[,#fileno]
```

Comments:

CANSCRIPT is a way to manipulate CAN data as it is done in the custom format as used in CANLOG. Depending on the size of the scripts up to 15 different scripts can be defined. Like the license file the scripts have to be stored on drive A. In fact the CANxxxxx.LIC file can be used as a script file.

The command will normally be executed after a CANRCV or in the subroutine called by a received CANOBJECT. The values of CAN ID and the databytes are used in the script, as is the CANCLOCK.

The timestamp which can be used in CANSCRIPT is not the timestamp of the received message, but the timestamp of the CANSCRIPT command.

Also a message number can be used, but this is the actual value of the Basic variable CANMESSNO. If this variable is not used, it will always be 0.

The default syntax of CANSCRIPT is CANSCRIPT filename. So if a script is defined on the A drive, which is called "CUSTOM.MSC", one can activate it by CANSCRIPT "CUSTOM" (MSC is the default extension for scripts: MSC=Machine Script Code).

Default all WRITE actions are to the SCREEN and terminal port. By entering # plus a number after the filename, the WRITE action is executed to the fileno, if a file is opened with that number, just like in CANLOG.

The CANSCRIPT command can best be used in the subroutine of a received CANOBJECT, where it activates a response Tx message.

For performance reasons, it is advised to:

- **Do not use `CANMESSNO`; if not defined it will always be 0 and the variable is not looked for.**
- **Don't use WRITEs in your script.**
- **If you sometimes want to use WRITE and in the real use not anymore, you can first call the command by `CANSCRIPT "custom"` or `CANSCRIPT "custom",#0` and later `CANSCRIPT "custom",#20` (any number of 20 or higher will ignore the WRITE function)**
- **Determine the pointer at the first call and after that call it just by the pointer. This is done as following:**
  - `10 DIM CANSCRIPTPTR: REM variable for the pointer`
  - `20 CANSCRIPT "custom",#20`
  - `30 ptr=CANSCRIPTPTR`
  - `40 CANSCRIPT ptr,#20`
- **The second time `CANSCRIPT` is called now, it does not have to search for the file on drive A. It will directly be guided to the script code. If more than one script is used, one has to use a different pointer for every script of course.**

The description of the script code is found in the next pages.

DESCRIPTION OF THE SCRIPTING PROTOCOL

The license file can be extended with a custom protocol. Only one protocol can be active, and this protocol will be loaded automatically at bootup. Protocols are described in a readable script file, but have to be converted in a low level machine code to be read from the runtime `CANLOG` command.

An on-line utility can be used to convert the protocol script to the embedded code in the licence file.

The machine code consists of 6 relevant bytes per line: The first one is called the command byte and is specified below; the second one the variable, 3 and 4 are reserved for a constant and 5 and 6 for a jump to another line; in fact the offset of the line. Although only 6 relevant bytes are on a line, every line in principle has 10 bytes. This had to be done, because the code is written on the A-drive and if a byte has the value 255, the next byte is skipped. This is compensated in the 4 non-relevant bytes.

If a protocol is added to the license file, two basic commands are extended in fact:
`CANSTATUS` has an additional line where the name of the protocol is listed.
`CANLOG` has the custom protocol options enabled and the data both in trace as in object format is converted to the chosen format.

The script has four types of commands:
1. The SPECIAL commands. Used for begin and end of the script. The start command has the name of the protocol included. This is displayed in `CANSTATUS`. Also special commands are made for floating point variables and since version 1.4 for reading and writing I/O pins, reading the keyboard, configure and send a CAN message and reading and configuring a timer (special values 64 up to 212).
2. The calculation command. One of the 16 user variables gets the value of a calculation. Calculations can be done with only 2 parameters. The first parameter is always a variable. It can be one of the user variables or one of the CAN variables. The second variable can be either a constant value (decimal, hexadecimal or binary) or also one of the user variables.
3. The write command. This command is used to fill the actual protocol line. The parameter of the write command can be some text or a user or CAN parameter.
4. The if command. This command is used to make a jump in the script on a certain condition. The condition has two parameters. The first one is either a user or a CAN variable. The second one can also be such a variable or a constant value (decimal, hexadecimal or binary).

All commands are prefixed by a line number and are in principle ended by a ":" and a CRLF. After the ":" can be an optional line number to jump to. This is always used in the if command where it is used to jump to if the condition is true. However it can also be used in every other command. Based on this syntax every user specific protocol can be realised.

16 user variables can be used, numbered 0 to 15.

Variable 15 has a specific function.
In the SPECIAL commands the variable gets an error value:
0. No error

1. **No key pressed**
2. **Pin configuration error**
3. **Pin out of range error**
4. **Not a Tx Fifo**
5. **Tx Fifo limit exceeded**
6. **Tx parameter out of range**
7. **No messages in Tx Fifo**
8. **Error in creating/editing `CANOBJECT`**
9. **`CANARRAY` not dimensioned**
10. **Index of `CANARRAY` out of range**
11. **`CANPHYSARRAY` not dimensioned to at least 128**

If used in Calculation it is possible to enter a low level command. See at CALCULATION commands how this is handled.

Also 16 CAN (and other special) related variables can be used. They are specified:

LSB / MSB	00	01	10	11
00	CAN ID	CAN ctrl byte	DB1	DB2
01	DB3	DB4	DB5	DB6
10	DB7	DB8	Mess. no	Mess. spec *
11	Timestamp	Float 1	Float 2	Float 3

*** Mess. spec: 32 bit value (mmmmmmmmrrrrrrrroooooooooooooooo) m=max. rxc; r=act. rxc; o=overfl.**

The SPECIAL commands: bits 1 and 2 of the command byte are 0:

command	variable	constant	jump	description
0	NA	NA	NA	Not used
4	0	0	0	End of code; start of texts; after this line
8	text	text	text	Text line; can be more than 6 bytes; ended bij CRLF
12	NA	NA	NA	End of protocol file
16	user var	NA	line	Set Float 1 to user variable
20	can var	NA	line	Set Float 1 to CAN variable
24	can dbn	NA	line	Set Float 1 to 4 CAN bytes, n first byte, little endian
28	can dbn	NA	line	Set Float 1 to 4 CAN bytes, n first byte, big endian
32	user var	NA	line	Set Float 2 to user variable
36	can var	NA	line	Set Float 2 to CAN variable
40	can dbn	NA	line	Set Float 2 to 4 CAN bytes, n first byte, little endian
44	can dbn	NA	line	Set Float 2 to 4 CAN bytes, n first byte, big endian
48	user var	NA	line	Set Float 3 to user variable
52	can var	NA	line	Set Float 3 to CAN variable
56	can dbn	NA	line	Set Float 3 to 4 CAN bytes, n first byte, little endian
60	can dbn	NA	line	Set Float 3 to 4 CAN bytes, n first byte, big endian
64	user var	NA	line	Set user variable to ASCII value of key
68	user var	pin no	line	Set user variable to value of I/O pin
72	user var	pin no	line	Set value of I/O pin to user variable
76	user var	NA	line	Set script timer to user variable
80	user var	NA	line	Set user variable to script timer
84	user var	NA	line	Set user variable to event (1 for CANRx, 2 for timer, 3 for both)
88 - 212 *	can var	user var	line	Configure CAN FIFO 0 - 31 for Tx

*** Numbers in steps of 4. Only CAN ID, CT and D1 to D8 can be configured. User variables may have offsets of 0 to 31 times 256 to write in a specific buffer of the FIFO. So it is possible to fire up to 32 messages with one Send FIFO command. FIFO's and `CANSEND` must be configured in the right way.**

Special commands continued:

command	variable	constant	jump	description
216	fifo no	NA	line	Send CAN FIFO no
220	obj_no	timer	line	Add or change a `CANOBJECT` configuration
224	user var	offset	line	Set canarray(offset) to variable integer
228	user var	offset	line	Set variable integer to canarray(offset)
232	user var	offset	line	Set canarray(offset) to variable float
236	user var	offset	line	Set variable float to canarray(offset)
240	user var	NA	line	Set mSecTimer to variable
244	user var	NA	line	Set variable to mSecTimer
248 *	user var	NA	line	Set variable to CRC_totalbits_stuffbits
252	0	ptr	line	Start of code; ptr to protocol text

*** This special command uses the the routines in the `CANPHYS` command to calculate the CRC, the totalbits and the stuffbits. This can be only used if the Basic variable `CANPHYSARRAY` is dimensioned at least to 128.**

The IF commands: command bit 0 is 0 and command bit 1 is 1

IF x COMP y is true THEN jump to specific line ELSE next line
Comparison:

Command bits 543	comp.	description
000	=	equal
001	<	smaller than
010	>	greater than
011	!=	not equal
100	<=	smaller or equal
101	>=	greater or equal
110	NA	not used
111	NA	not used

Command bit 6 is 0: y = User var; command bit 6 is 1: y = CAN var; not relevant if bit 8 = 0

Command bit 7 is 0: x = User var; command bit 7 is 1: x = CAN var

Command bit 8 is 0: y = Constant (value in constant bytes 3 and 4); Command bit 8 is 1: y is variable.

Variable byte (bits 1-4): y value (all zero if command bit 8 is 0)
Variable byte (bits 5-8): x value

Constant: relevant value if command bit 8 is 0

Jump: the offset of the line no if condition is true.

The CALCULATION commands: command bit 1 is 0 and command bit 2 is 1

x = y CALC z
Calculation

Command bits 6543	calc	description
0000	+	add
0001	-	subtract
0010	*	multiply
0011	/	divide
0100	&	logic and
0101	|	logic or
0110	<<	shift left (multiply by 2^z)
0111	>>	shift right (divide by 2^z)

Since version 2.1 the following extra calculations have been introduced:

Command bits 6543	calc	description
1000	pow	y to the power of z
1001	mod	y modula z (calculation done as float; result is integer)
1010	sin	Sine in radians : act. formula: y * sin(z / y). Result always integer
1011	cos	Cosine in radians: act formula: y * cos(z / y)
1100	log	Natural Logarithmic: act. formula: y * log(z / y)
1101	l10	10th Logarithmic: act. formula: y * l10(z / y)
1110	xor	y xor z
1111	hwt	Hamming weight: act. formula: y + hwt(z)

Command bit 7 is 0: y = user variable; command bit 7 is 1: y = CAN variable

Command bit 8 is 0: z = constant; command bit 8 is 1: z = user variable (no in constant)

Variable byte bits 1-4: x value

Variable byte bits 5-8: y value

Constant bytes: constant value or user variable

Jump: If value other than 0 the offset for the next line

If the calculation is as follows: var15=varY+varZ, where Y and Z can be any variable between 0 and 14 a new command is calculated with the new command (low byte Y), variable (high byte Y) and constant bytes in varZ. Jump bytes are treated as before.
If the calculation is: var15=var15+varZ (0<Z<15) a conditional jump is performed. This acts like a Switch/Case statement or the Basic ON …. GOTO. If varZ=0 the next line is executed; if varZ the line with an extra offset of the value of varZ is jumped to. An extra global offset can be created by the Jump parameter.

The WRITE commands: command bit 1 is 1 and command bit 2 is 1

WRITE characters
Kind of characters:

Command bits 543	Description
000	text; constant has the offset value to the text
001	decimal value
010	hex value (no pre string)
011	hex value in format 0xvalue
100	binary value
101	float value
110	ASCII character
111	No of same ASCII characters; variable contains the number; constant the ASCII value

Command bit 6 and bit 8: reserved

Command bit 7 is 0: value is user variable; bit 7 is 1: value is a CAN variable

Variable byte: the user or CAN variable (except if no of ASCII characters is chosen; see above)

Constant bytes: offset to the text (if text is chosen) or ASCII value (no of chr); otherwise not relevant

Jump: If value other than 0 the offset for the next line

The basic variable `CANEVENTINT` can be used to activate the script by a timer. The value is in ms. The script can now be activated by a received message on the bus, but also based on a time interval. The command 84 is used to check which event occurred, 80 is used to write a new value into the timer and 76 to read the current value.

The following protocols are available as script:
CANopen
J1939
FMS

2.21 CANSEND

Purpose:
Transmit one message of a FIFO register

Syntax:
`CANSEND [id[,type[,id ext[,len[,data()[,<ok>]]]]]]`

Comments:

`type` **consists of 16 bits (<f><pp><lllll><nnnnn><s><r><e>), where**

- **f = Fill bit: 1 means not really a sending action, but fill only**
- **pp = Send priority bits in FIFO**
- **lllll = The location address in the FIFO; only used at specific write**
- **nnnnn = The Fifo no.**
- **s = The special bit when set; special actions can be undertaken**
- **r = RTR bit; when s=0 used as RTR bit; s=1 the RTR enable bit is set**
- **e = Extension bit; when e=1 the <id ext> contains the 18 LSB of ID**

The optional `id ext`**. The problem in Basic is that it only has single precision floating point variables. So the maximum number which can be calculated without exponential description is 999999. The extended ID however can go up to 536870911. That is why we advise to split the ID in 11 (std) and 18 (ext) parts, when the ID is calculated. If ID is used as a constant it is no problem to go up to 536870911 in ID.**

The kind of SEND action is determined by f, s en r bit

- **s r f**
- **0 0 0 : The standard Tx action: message is added to FIFO and sent**
- **0 0 1 : Message is added to FIFO, but not sent yet**
- **0 1 0 : As 0 0 0 but with RTR bit set**
- **0 1 1 : As 0 0 1 but with RTR bit set**
- **1 0 0 : Send only action; all messages in FIFO are sent**
- **1 0 1 : Fills the specific location lllll with the message**
- **1 1 0 : Sets the RTR enable bit for this FIFO**
- **1 1 1 : Reset RTR enable bit; Update priority with pp; Reset FIFO**

Since version 1.5 all parameters in `CANSEND` **has been made optional.**
0 parameters: Command is interpreted as Send only action; FIFO has to be filled before.
1 parameter: id is specified. `type=0; len=8; dim txdata(8); dim txok`
2 parameters: id and type are specified; `len=8; dim txdata(8); dim txok`
3 parameters: id, type and len are specified; `dim txdata(8); dim txok`
4 parameters: id, type, ext id and len are specified; `dim txdata(8); dim txok`
5 parameters: id, type, len and the names of data and ok are specified
6 parameters: id, type, ext id, len and the names of data and ok are specified

The Basic variable `CANTXERRCNT` will be updated if it is used in Basic. It contains the actual status of the Tx error counter.

Examples:

```
10 DIM txdata(8):txdata(0)=1
20 CANOPEN 500
30 CANSEND 100,0,1
```

The CANbus is activated at 500 kbit/sec and a message with ID 100 and databyte 1 is sent.

```
10 DIM txdata(8):txdata(0)=1
20 CANOPEN 500
30 CANSEND 100,&H8000,1
40 txdata(0)=2
50 CANSEND 100,&H8000,1
60 CANSEND
```

Now the messages are set into the FIFO (first one with data 1; second data 2) and after that sent directly after each other.

2.22 CANSTATUS

Purpose:
Displays the status of the CAN controller

Syntax:
CANSTATUS

Comments:

The control register: Serial no., Port no., Timestamps (on/off), Mode.

The config register: Bitrate, sample point, SJW and no of samples.

The FIFO's are marked T (Tx) or if Rx: _ (not active), + (>1), 1-9 (no of links), 0 (> 9 links).

FIFO status shows available data in the FIFO (for Tx always 1; for Rx 1 if data is not read)

FIFO rxlost shows the FIFO's which have had an overflow (1).

Next line shows the actual values of Rx and Tx error counters. The CAN clock has been added to this line. For resolution see the value on the top line (in uS).

The status line of the CANOBJECT is included here. 1 HEX byte status for every FIFO:
2 LSB's: kind of object: 00: timed; 01: dependant object; 10: once disabled; 11: once enabled
Bit 3: 0 for the standard port; 1 for the alternate port
Bits 4 and 5: log to (00: no log; 01: screen; 10 and 11 file #1 or #2)
Bit 6: Basic parameter on/off; Bit 7: Tx/Rx; Bit 8: ON/OFF.

After all objects the standard CANport of the objects is listed (_ if no objects initialised yet).

The name of an optional protocol in the license file is on the next line.

All optional CANIOLINKs are the last lines.

If DIM CANTEST(n) is executed before CANSTATUS the parameters are stored in the CANTEST array. The number of stored values is dependent on the value of n. See next page how to interpret the values.

Values of `CANTEST` in `CANSTATUS`:

`CANTEST(0)`:	**256 * (CAN version * 10) + MSB of serial number**
`CANTEST(1)`:	**LSBs of serial number**
`CANTEST(2)`:	**CAN port (0, 1, 2 or 3)**
`CANTEST(3)`:	**(0:normal; 2:loopback; 3: listen only; 4:configuration;7:all messages)**
`CANTEST(4)`:	**`CANCLOCK` (0 if timestamps are disabled)**
`CANTEST(5)`:	**Timestamp resolution**
`CANTEST(6)`:	**Bitrate (bits/sec)**
`CANTEST(7)`:	**Sample point (%)**
`CANTEST(8)`:	**Synchronisation Jump Width (1, 2, 3 or 4)**
`CANTEST(9)`:	**No. of samples (1 or 3)**
`CANTEST(10)`:	**LSBs of startaddress FIFOs**
`CANTEST(11)`:	**MSBs of startaddress FIFOs**
`CANTEST(12)`:	**No. of active FIFO locations**
`CANTEST(13)`:	**Total no. of FIFO locations**
`CANTEST(14)`:	**Actual value of Rx error counter**
`CANTEST(15)`:	**Actual value of Tx error counter**
`CANTEST(16)`	
.....	
`CANTEST(47)`:	**Status of `CANOBJECT` 0 … 31**
`CANTEST(48)`:	**CAN port of `CANOBJECT`s**
`CANTEST(49)`	
.....	
`CANTEST(70)`:	**Name of protocol if available (ASCII values)**
`CANTEST(71)`	
.....	
`CANTEST(198)`:	**Status of `CANIOLINK` 0 … 127** **0 if not active otherwise: 10000 +(100 * pin no) + object no**

2.23 CANSUB

Purpose:
Jump to a time-based subroutine during a continuous CAN command

Syntax:
CANSUB [timer[,lineno]]

Comments:

CANSUB is used like the SETTICK command. This means a subroutine starting at lineno is called every time in timer mSec. The interval time of the commands CANLOG, CANVIEW, CANBRIDGE and CANREPLAY is influenced by the timer specification in CANSUB. If lineno is not specified or lineno=0 the command only changes this interval time. Default timer is set to 1000 (1 sec) and the minimum value is 100 mS. The interval time determines when the busload is calculated and all the other specifications of CANVIEW are displayed. Also it grabs the input of the keyboard to see if it has to stop the command.

If lineno is also specified it will start the subroutine starting at lineno and will return to the command which was executing when the call was done and the command CANRETURN is seen. This means it stays within the command (CANCONTINUE=2) and will not step to the next command line as does SETTICK with IRETURN and GOSUB with RETURN.

Example:

```
10 CANOPEN 500
20 CANSUB 1000,100:'Every 1000 ms execute subroutine at 100
30 DIM canmessno:'Variable gets the actual message counter
40 CANLOG 4
100 PRINT
110 PRINT "No of messages last second:";canmessno-prevno
120 PRINT
130 prevno=canmessno:'Remember actual no.
140 CANRETURN
```

The program will print a line every second with the number of messages during the last second. By using the CANSUB the ESC or Q will not work anymore to stop the logging. Only CTRL-C will stop the program now.

2.24 CANVIEW

Purpose:
Display an overview of a running CAN network on one line

Syntax:
CANVIEW [format[,fifono[,period[,lineno[,interval]]]]]

Comments:

The parameters which are displayed are: the number of messages, the number of ID's, the busload and the error situation. Values are displayed periodically, both over the specific period and overall. The optional parameter interval can be used to set the measurement period in ms (default value 1000).

The optional parameters fifono, period and lineno are used in the same way as in the CANLOG command.
Also the Basic parameters CANPERIOD, CANLINENO, CANTIME, CANMESSNO, CANCONTINUE, CANLOAD, CANLOADACT, CANLOADPERIOD(n), CANLOADMAX, CANLOADMAXTIME and CANBLINK can be used in the same way as in the CANLOG command.

The optional parameter format is used for the format of the line.
The default value 0 displays all value in one line as: <SMlLPTiIcCEO> with:

S: The measurement time in seconds (CANTIME)
N: No of messages (CANMESSNO)
A: Actual busload in the last interval (CANLOADACT)
L: The busload overall (CANLOAD)
M: The maximum busload (CANLOADMAX)
T: Time of maximum busload (CANLOADMAXTIME)
i: The number of different message ID's in the last interval *
I: The total number of different ID's *
c: The maximum value of the Rx Error Counter in the last interval
C: The maximum value of RxC overall (CANERRCNT)
E: The total number of errors (in fact the number of increments of the RxC) **
O: The overall number of overflows (CANOVFCNT)

If format has the value 1 to 12 one of the parameters is shown in the order SMlLPTiIcCEO.
Format = 13: Busload graphical: Dynamic scaling with values A, L, and M
Format = 14: Error Counter value graphical: Scale 0 - 136 with values c and C
Format = 15: The values 1 to 12 sequential. Every parameter is displayed during one period.

If DIM CANTEST(15) has been executed before CANVIEW the parameters 1 to 12 are stored in CANTEST(1) - CANTEST(12). Other values are set to 0. The format is not relevant in this case. The busload values are multiplied by 10, because the CANTEST values can only be integers.

*** To store the used ID's the upper memory area of the FIFO space is used as in CANLOG. Default 30 FIFO's are free for this purpose. Each FIFO can store 4 ID's. So a maximum of 120 different ID's can be stored. If this maximum is exceeded, the value 999 is stored and the ID read is stopped.**

**** If the increment of the error counter > 8 the error will probably be caused by the test system itself and therefore this value is set to the maximum value 9999. The user should check for the correct physical and datalink settings of the SI2CBB.**

2.25 CHDIR

Purpose:
To change from one working directory to another.

Syntax:
```
CHDIR pathname$
```

Comments:

`Pathname$` is a string expression of up to 63 characters.

To make games the working directory, type the following command:
```
CHDIR "GAMES"
```

The special entry “..” represents the parent of the current directory and “.” represents the current directory.

2.26 CIRCLE

Purpose:
To draw a circle on the screen.

Syntax:
`CIRCLE(xcenter, ycenter), radius[,[color][,[F]]]`

Comments:

`xcenter` and `ycenter` are the x- and y- coordinates of the center of the circle, and `radius` is the radius (measured along the major axis) of the circle. The quantities `xcenter` and `ycenter` can be expressions.

`color` specifies the color of the circle. 0 draws with black color, anything different than 0 draw with white color.

The `F` is fill parameter and fills the circle with the color specified in color . If the circle goes outside the drawing area it's truncated and no error is generated.

Example 1:

```
10 CIRCLE(100,100), 50
```

Draws a circle of radius 50, centered at 100x and 100y coordinates.

```
20 CIRCLE(100,100), 50, 0
```

Erases the circle drawn with the previous command.

Example 2:

```
10 CLS ' This will draw 16 circles
20 FOR R=160 TO 0 STEP -10
30 CIRCLE (160,160),R,1
40 NEXT
```

2.27 CLEAR

Purpose:
To set all numeric variables to zero, all string variables to null. See `ERASE` for deleting specific array variables.

Syntax:
`CLEAR`

Comments:

The `CLEAR` command:

- Clears all user variables
- Resets the strings

Examples:

`CLEAR`
Zeroes variables and nulls all strings.

2.28 CLOSE

Purpose:
To terminate input/output to a disk file or a device.

Syntax:
```
CLOSE #filenumber[,[#]filenumber]...]
CLOSE CONSOLE
```

Comments:

`filenumber` is the number under which the file was opened. The association between a particular file or device and file number terminates upon execution of a `CLOSE` statement. The file or device may then be reopened using the same or a different file number. A `CLOSE` statement with no file number specified will lead to error.

A `CLOSE` statement sent to a file or device opened for sequential output writes the final buffer of output to that file or device.

A `CLOSE CONSOLE` will close the serial port which is opened for console.

Examples:

```
250 CLOSE #1
```
This closes file #1.

```
300 CLOSE #2, #3
```
Closes all files and devices associated with file numbers 2, and 3.

2.29 CLS

Purpose:
To clear the screen.

Syntax:
```
CLS
```

Comments:

`CLS` returns the cursor to the upper-left corner of the screen

Examples:

```
10 CLS
```
This clears the screen.

2.30 COMLOG

Purpose:
Log data of serial port(s)

Syntax:
COMLOG [#fileno,][format[,period]]

Comments:

#fileno may be 1 or 2. This restriction is due to the fact that the same structure as for CANLOG is used. If it is greater than 2, it is reset to 0, which means that the log is on screen. If logging to a file is wished, the file should be opened by the OPEN command before. If this option is chosen, the format parameter will always be 1, independent of what is on the command line.

Before the command can be executed also 1, 2, 3 or 4 COM ports should be opened by the OPEN command. Be careful: the #no should be unique for every port and should not conflict with the #fileno. Only the Rx lines of the COM ports are relevant. These are on the following ARDUINO pins: COM1: on D2; COM2: on D6; COM3: on UEXT 4, so not on the ARDUINO pins and COM4: on the RS232 connector (pin 2) and also on D0 if R2 is mounted on the Duinomite board (it is not available on the embedded PIC32T795 module). Normally only 2 ports are selected to monitor a serial connection. Of course the right baud rate must be given to the serial inputs.

format has 4 possible values (0-3). 0 means ASCII monitoring. This has a special layout: All data comes on a horizontal line; one for every COM port. If the ASCII value is printable (32-127) it is presented by the right ASCII character; if it is not printable a '.' is printed. Data scrolls from right to left. The other possible values are 1 for hexadecimal; 2 for decimal and 3 for binary. In these cases data scrolls in a vertical direction on the screen. Every time data comes from another serial port a line is inserted with the new COM port.

The command can be stopped by <ESC> or <CTRL-C>. In a program it can also be decided to run the logging for a certain period. The period parameter can be used for this.

IMPORTANT: All inputs (except COM4: on the RS232 connector) should be protected by Schottky diodes between 0 and 3.3 Volts.

Example:

```
10 OPEN "logfile.txt" for output as #1
20 OPEN "COM1:" as #2
30 OPEN "COM2:" as #3
40 COMLOG #1,1,10000
50 CLOSE #1
```

The serial inputs COM1: and COM2: are used for logging. Data is written to a logfile "logfile.txt" during 10 seconds.

2.31 CONTINUE

Purpose:
To continue program execution after a break.

Syntax:
`CONTINUE`

Comments:

Resumes program execution after CTRL-BREAK, or CTRL-C . Execution continues at the point where the break happened. If the break took place during an `INPUT` statement, execution continues after reprinting the prompt.

`CONTINUE` is useful in debugging, in that it lets you break code, modify variables using direct statements, continue program execution, or use `GOTO` to resume execution at a particular line number.

If a program line is modified, `CONTINUE` will be invalid.

2.32 COPYRIGHT

Purpose:
To print copyright message.

Syntax:
`COPYRIGHT`

Comments:

Print copyright message.

SI2-CBB version 2.4 added the changes of DMBasic 2.7. Added, Edited and Deleted Commands and Functions are mentioned.

2.33 DATA

Purpose:
To store the numeric and string constants that are accessed by the program READ statement(s).

Syntax:
DATA constants

Comments:

constants are numeric constants in any format (fixed point, floating-point, or integer), separated by commas. Numerical constants can also be expressions such as 5 * 60.

String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons, a keyword (such as THEN, WHILE, etc) or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

DATA statements are not executable and may be placed anywhere in the program. A DATA statement can contain as many constants that will fit on a line (separated by commas), and any number of DATA statements may be used in a program.

READ statements access the DATA statements in order (by line number). The data contained therein may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program. The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement, or a "Type Mismatch" error occurs.

DATA statements may be reread from the beginning by use of the RESTORE statement.

For further information and examples, see the RESTORE statement and the READ statement.

Example 1:

```
80 FOR I=1 TO 10
90 READ A(I)
100 NEXT I
110 DATA 3.08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37
```

This program segment reads the values from the DATA statements into array A. After execution, the value of A(1) is 3.08, and so on. The DATA statements (lines 110-120) may be placed anywhere in the program; they may even be placed ahead of the READ statement.

Example 2:

```
5 PRINT
10 PRINT "CITY","STATE","ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,","COLORADO",80211
40 PRINT C$,S$,Z
RUN
CITY STATE     ZIP
DENVER,   COLORADO  80211
```

This program reads string and numeric data from the DATA statement in line 30.

2.34 DATE$

Purpose:
To set the current date.

Syntax:
`DATE$=v$`

Comments:

The date is set to “01-01-2000” at power up. If MOD-RTC (Real-Time-Clock Module) is connected to UEXT the current date will be set from the read content from MOD-RTC.

`v$` is a valid string literal or variable.

`v$` can be any of the following formats when assigning the date:

dd-mm-yy
dd/mm/yy
dd-mm-yyyy
dd/mm/yyyy

If `v$` is not a valid string, a "Type Mismatch" error results. Previous values are retained.

If any of the values are out of range or missing, an "Invalid date" error is issued. Any previous date is retained.

To read the actual date, use the function `DATE$` as described in chapter 3

2.35 DELETE

Purpose:
To delete program lines or line ranges.

Syntax:
```
DELETE [line number1][-line number2]
DELETE line number1-
```

Comments:

`line number1` is the first line to be deleted.

`line number2` is the last line to be deleted.

MM-BASIC always returns to command level after a `DELETE` command is executed. Unless at least one line number is given, an "Invalid syntax" error occurs.

If the line number does not exist "Invalid line number" error occurs.

Examples:

```
DELETE 40
```
Deletes line 40.

```
DELETE 40-100
```
Deletes lines 40 through 100, inclusively.

```
DELETE -40
```
Deletes all lines up to and including line 40.

```
DELETE 40-
```
Deletes all lines from line 40 to the end of the program.

2.36 DIM

Purpose:
To specify the maximum values for array variable subscripts and allocate storage accordingly.

Syntax:
`DIM variable(subscripts)[,variable(subscripts)]...`

Comments:

If an array variable name is used without a `DIM` statement, a "Array must be dimensioned first" error occurs.

The maximum number of dimensions for an array is 8.

The minimum value for a subscript is always 0, unless otherwise specified with the `OPTION BASE` statement.

An array, once dimensioned, cannot be re-dimensioned within the program without first executing a `CLEAR` or `ERASE` statement.

The `DIM` statement sets all the elements of the specified arrays to an initial value of zero.

Examples:

```
10 DIM A(20)
20 FOR I=0 TO 20
30 READ A(I)
40 NEXT I
```

This example reads 21 DATA statements elsewhere in the program and assigns their values to A(0)through A(20), sequentially and inclusively.

2.37 DO

Purpose:
To create a loop.

Syntax:
```
DO [ loop statements ] LOOP
DO WHILE expression [loop statements] LOOP
DO [loop statements] LOOP UNTIL expression
```

Comments:

The first structure will loop forever; the `EXIT` command can be used to terminate the loop or control must be explicitly transferred outside of the loop by commands like `GOTO` or `RETURN` (if in a subroutine).

The second structure will loop until `expression` is zero. If `expression` is nonzero (true), loop statements are executed until the `LOOP` statement is encountered. MM-BASIC then returns to the `DO WHILE` statement and checks expression. If it is still true, the process is repeated.If it is not true, execution resumes with the statement following the `LOOP` statement.

`DO WHILE LOOP` loops may be nested up to 20 times. Each `LOOP` matches the most recent `DO WHILE`.

An unmatched `LOOP` statement causes a "LOOP without matching DO" error

`DO WHILE LOOP` is equivalent to the old `WHILE-WEND` statement which is also implemented in MM-BASIC for compatibility.

The third structure does the same as the second one, only the expression is now at the end of the loop. So the loop statements are executed at least once.

Examples:

```
10 DO
20 PRINT "HELLO"
30 IF INKEY$ = "q" THEN EXIT
40 LOOP
50 PRINT "DONE"
```

This example will print “HELLO” until “q” key is pressed.

```
10 DO WHILE INKEY$ <> “q”
20 PRINT “HELLO”
30 LOOP
40 PRINT “DONE.”
```

This example does the same as the one before.

```
10 DO
20 PRINT “HELLO”
30 LOOP UNTIL INKEY$ = “q”
40 PRINT “DONE.”
```

Again an example for printing “HELLO” until q pressed.

```
90 'BUBBLE SORT ARRAY A$
100 FLIPS=1
110 DO WHILE FLIPS
115 FLIPS=0
120 FOR N=1 TO J-1
130 IF A$(N)>A$(N+1) THEN SWAP A$(N), A$(N+1): FLIPS=1
140 NEXT N
150 LOOP
```

Sorting program for array

2.38 DRIVE

Purpose:
Change drive

Syntax:
```
DRIVE "A:"
DRIVE "B:"
```

Comments:

Drive “A:” is the virtual drive in the PIC32 internal flash memory

Drive “B:” is the SD card if available.

Drive “B:” is the default drive if the SD card is available.

The SI2-CBB software uses the A-drive for the license file and optional script files.

The size of the A-drive is restricted to 64 KB.

2.39 EDIT

Purpose:
Edit the line ‘line-number’.

Syntax:
`EDIT [line_number]`

Comments:

`line_number` is the number of a line existing in the program. If no line number is used this command will edit the previous entry typed at the command prompt. If a running program has just terminated with an error this will automatically edit the line that caused the error.

On entering the edit mode the line will be displayed, the cursor placed at the end of the line and the editing mode set to over-type.

The editing keys are:

LEFT/RIGHT ARROWS	Moves the cursor within the line
HOME/END	Moves the cursor to the start or end of the line
DELETE	Delete the character over the cursor
BACKSPACE	Delete the character before the cursor
INSERT	Will switch between insert and overtype mode.

Use `ENTER’ to finish editing (even in insert mode). The line is added to the program just as if it had been typed at the command prompt. If the line number had been changed a new (edited) copy of the line will be added to memory, if it is unchanged the line will replace ‘line-number’.

When editing a program line the UP-ARROW will switch to the previous line and DOWN-ARROW to the next line number. When doing this any changes will be automatically saved (the same as using ENTER) before moving to the next line.

MM-BASIC is always in edit mode when entering data at the command prompt or for the `INPUT` and `LINE INPUT` commands. In these cases the ARROW KEYS can be used to move within the line to correct errors. If the UP-ARROW key is pressed at the command prompt it will act the same as `EDIT` with no line number (edit the last error line or entered command). Subsequent UP/DOWN-ARROW presses will move through the list of recent entries.

All the editing keys work with Teraterm and Putty (in VT100 mode) so editing can also be accomplished over a USB or serial link using these terminal emulators or any other VT100 compatible terminal emulator.

The maximum line length that can be edited is 79 chars in VGA mode and 49 chars in composite mode. If more than this number of characters are entered in over-type mode MM-BASIC will automatically enter normal text entry mode without the editing functions. If in insert mode any extra characters will be rejected.

The current line is always the last line referenced by an `EDIT` statement, `LIST` command, or error message.

If line number refers to a line which does not exist in the program, an "Invalid Line Number" error occurs.

Examples:

```
EDIT 150
```
Displays program line number 150 for editing.

2.40 ELSE

Purpose:
Execute one or more statements or jump to another line if a condition is false.

Syntax:
`IF … THEN … [ELSEIF …] ELSE …`

Comments:

`ELSE` is described in the IF command

2.41 ELSEIF

Purpose:
Execute one or more statements or jump to another line if a condition is false and another condition is true.

Syntax:
`IF … THEN … ELSEIF … [ELSE …]`

Comments:

`ELSEIF` is described in the IF command

2.42 END

Purpose:To terminate program execution, close all files, and return to command level.

Syntax:
`END`

Comments:

`END` statements may be placed anywhere in the program to terminate execution.

An `END` statement at the end of a program is optional. MM-BASIC always returns to command level after an `END` is executed.

`END` closes all files.

Example:

```
520 IF K>1000 THEN END ELSE GOTO 20
```

Ends the program and returns to command level whenever the value of K exceeds 1000.

2.43 ENDIF

Purpose:
End the multi line IF statements.

Syntax:
`IF … THEN … [ELSEIF …] [ELSE …] ENDIF`

Comments:

`ENDIF` is described in the IF command.

2.44 ERASE

Purpose:
To eliminate arrays from a program.

Syntax:
`ERASE list of array variables`

Comments:

Arrays may be re-dimensioned after they are erased, or the memory space previously allocated to the array may be used for other purposes.

If an attempt is made to re-dimension an array without first erasing it, an error occurs.

Examples:

```
200 DIM B(250)
.
450 ERASE A,B
460 DIM B(3,4)
```

Arrays A and B are eliminated from the program. The B array is re-dimensioned to a 3-column by 4-row array (12 elements), all of which are set to a zero value.

2.45 ERROR

Purpose:
To simulate the occurrence of an error, or to allow the user to define it.

Syntax:
`ERROR error$`

Comments:

The program execution stops and error$ is displayed as error.

Example:

The following example simulate error 15 (the code for "String too long"):

```
10 INPUT “Number”;a
20 INPUT “div “;b
30 IF B = 0 THEN ERROR “DIVISION BY ZERO”
40 ELSE PRINT a/b
```

2.46 EVAL

Purpose:
Execute a statement, which is created in a string

Syntax:
```
EVAL statement$
```

Comments:

EVAL is an instruction coming from the scripting language JavaScript. In other languages it is also called EXEC. It converts a string into a statement. In this way it is possible to execute a statement, dependent on variables. As Basic is in fact also a scripting language, the interpreter translates the statement in run time, it is possible to include such a statement in Basic.

For our CAN version of Basic it is very useful to have such a statement. For the CANOBJECT statement we use the Basic array CANOBJxxDATA, where xx stands for the number of the object (00 - 31). If we use multiple CANOBJECTs we have to declare all the arrays and when we want to have the data of one specific CANOBJECT we have to find with IF-THEN or ON x GOTO/GOSUB statements the right action to take. This is because the xx is in the variable name and is not one of the variables itself. By using the EVAL statement like EVAL A$, we can build the A$ string dependent on the CANOBJECT number and in this way we can DIM multiple CANOBJxxDATA arrays in a FOR-NEXT loop or modify/read the data of a specific CANOBJECT, specified by an user input.

MMBasic (or DMBasic) translates commands, functions and arithmetic operators into tokens when a line is entered. In runtime only a tokenized program can be executed. If we would enter the string in Basic format we would have to tokenize the string before we can execute it. This is quite inefficient. That is why it is decided that the string we use in the EXEC command should already be tokenized. The tokens are the ASCII codes 128 - 255.

The structure of a statement is: <command token><space><variable><function token><variable><function token>.....<0>. The tokens in the string have to be entered by CHR$(x). So our string would be:
```
A$=CHR$(command token)+" "+"variable"+CHR$(function
token)+"variable"+CHR$(function token) ….+CHR$(0)
```

A simple method to get the tokenized translation of a statement is the following:
Temporary add a line 1 in your program, e.g. `1 PRINT "test"`
Now type MEMORY and you will get the used memory for program and variables. In our CAN Basic this command does not only give the size, but also the start of the different areas. We use the start of the program area, e.g. 0xA000yyyy (yyyy are also two hexadecimal bytes, which change at every version. Now enter the following line:
```
FOR x=0 TO 100:PRINT PEEK(&HA000,&Hyyyy+x);:NEXT
```
The result will be: 0 1 0 1 32 130 32 34 116 101 115 116 34 0 …….
The first characters are: a dummy 0, a 1 for start of line, a 0 for high byte of lineno, a 1 for low byte of line number (lineno is 1), a 32 for space after lineno, a 130 as token for PRINT, a 32 for space, a 34 for ", 116, 101, 115, 116 as ASCII for test, a 34 for " and a 0 for end of statement.

We can build the string as follows: `A$=CHR$(130)+" "+CHR$(34)+"test"+CHR$(34).` The first bytes should not be in the string, these are in fact in the `EVAL` statement itself. The ending 0 should be included as `CHR$(0)`. Don't forget the `CHR$(34)` for ". `EVAL A$` will also print test now.

The `EVAL` statement can also be used to make an unreadable Basic program, by building strings with only `CHR$(x)`. Only one statement is allowed in the string.

The following pages contain the tokens of the commands and functions. The categories are the header files from the sources of the DMBasic. These are not relevant in using the tokens in the `EVAL` command.

Remark: the token can include the opening "(". In this case this should not be added in the `EVAL` command in this case. However don't forget the closing ")".

COMMAND TABLE

<Commands.h>

128 REM	129 LET	130 PRINT	131 ?
132 LIST	133 RUN	134 SAVE	135 LOAD
136 MERGE	137 NEW	138 CLEAR	139 GOTO
140 IF	141 ELSE	142 ELSEIF	143 ENDIF
144 END	145 INPUT	146 TRON	147 TROFF
148 FOR	149 NEXT	150 DO	151 LOOP
152 WHILE	153 WEND	154 ERROR	155 EXIT
156 RANDOMIZE	157 GOSUB	158 RETURN	159 DATA
160 READ	161 RESTORE	162 LINE INPUT	163 DELETE
164 ON	165 DIM	166 ERASE	167 OPTION
168 CONTINUE	169 WRITE	170 MEMORY	171 RENUMBER
172 EDIT	173 POKE	174 AUTO	

<Files.h>

175 OPEN	176 CLOSE	177 FILES	178 MKDIR
179 RMDIR	180 CHDIR	181 KILL	182 NAME
183 DRIVE	184 MSDON	185 MSDOFF	186 SDFORMAT

<External.h>

187 PIN(	188 SETPIN	189 PWM

<Graphics.h>

190 CLS	191 CIRCLE	192 LINE	193 PSET
194 PRESET	195 LOCATE	196 PIXEL(	197 SAVEBMP

<Misc.h>

198 PAUSE	199 TIMER	200 DATE$	201 TIME$
202 SOUND	203 IRETURN	204 SETTICK	205 COPYRIGHT
206 FONT			

<I2C.h>

207 I2CEN	208 I2CDIS	209 I2CSEND	210 I2CRCV
211 I2CSEN	212 I2CSDIS	213 I2CSSEND	214 I2CSRCV
215 NUM2BYTE	216 OLED		

<XModem.h>

217 XMODEM

<Setup.h>

218 SETUP

<RTC.h>

219 SLEEP	220 MM.BLANK	221 SDENABLE	222 SDDISABLE
223 SOFTRESET	224 WATCHDOG		

<CAN.h>

225 CANOPEN	226 CANCLOSE	227 CANREG	228 CANSTATUS
229 CANFIFO	230 CANFILTER	231 CANMASK	232 CANLINK
233 CANSEND	234 CANRCV	235 CANLOG	236 CANRESET
237 CANVIEW	238 CANREPLAY	239 CANBRIDGE	240 CANOBJECT
241 CANSUB	242 CANRETURN	243 CANSCRIPT	244 CANINT
245 CANPHYS	246 CANIDSCAN	247 CANIOLINK	248 COMLOG
249 EVAL			

TOKEN TABLE FOR FUNCTIONS AND OPERATORS

<Functions.h>

128 ABS(	129 ASC(	130 ATN(	131 CHR$(
132 CINT(	133 COS(	134 EXP(	135 FIX(
136 HEX$(	137 INSTR(	138 INT(	139 LEFT$(
140 LEN(	141 MID$(	142 OCT$	143 RIGHT$(
144 RND(	145 RND	146 SGN(	147 SIN(
148 LOG(	149 SQR(	150 TAN(	151 VAL(
152 SPACE$(	153 SPC(	154 STR$(	155 STRING$(
156 FORMAT$(	157 UCASE$(	158 LCASE$(	159 PEEK(
160 MM.VER	161 GETDIM(		

<Commands.h>

162 THEN	163 ELSE	164 GOTO	165 GOSUB
166 TO	167 STEP	168 FOR	169 WHILE
170 UNTIL			

<Operators.h>

171 ^	172 *	173 /	174 \
175 MOD	176 +	177 -	178 NOT
179 <>	180 >=	181 =>	182 <=
183 =<	184 <	185 >	186 =
187 AND	188 OR	189 XOR	

<Files.h>

190 EOF(	191 LOC(	192 LOF(	193 CWD$
194 AS	195 MM.ERRNO	196 INPUT$(	197 MM.DRIVE$
198 MM.DRIVE	199 MM.FNAME$		

<External.c>

200 PIN(	201 GETPIN(

<Graphics.h>

202 PIXEL(	203 MM.HRES	204 MM.VRES

<Misc.h>

205 POS	206 TIMER	207 DATE$	208 TIME$
209 INKEY$	210 TAB(	211 SPI(	212 LOAD
213 LOADB	214 DOW		

<I2C.h>

215 BYTE2NUM(	216 MM.I2C

<Setup.h>
217 MM.SETUP

<RTC.h>
218 MM.SLEEP 219 MM.BLANK 220 MM.BOOTUP

<CAN.h>
221 CANOBJECT(222 CANCLOCK

2.47 EXIT

Purpose:
Exit from `DO...LOOP` or `FOR...NEXT` statement.

Syntax:
```
EXIT
EXIT FOR
```

Comments:

The program execution will continue after the `LOOP` or `NEXT` statement.

Examples:

This example will print “HELLO” until “q” key is pressed.

```
10 DO
20 PRINT “HELLO”
30 IF INKEY$ = “q” THEN EXIT
40 LOOP
50 PRINT “DONE”
```

This example will print the numbers from 1 to 5 instead to 10

```
10 FOR I = 1 TO 10
20 PRINT I; “ “;
30 IF I = 5 THEN EXIT FOR
40 NEXT
```

2.48 FILES

Purpose:
List files in the current directory on the SD card.

Syntax:
`FILES [search-pattern$]`

Comments:

The SD card (drive B:) may use an optional `search-pattern$`. Question marks (?) will match any character and an asterisk (*) as the first character of the filename or extension will match any file or any extension. If omitted, all files will be listed.

Example:

```
FILES "*.BAS"
```

Will list all files on the SD card with .BAS extension.

2.49 FONT

Purpose:
Select, load or unload a font for the video output.

Syntax:
```
FONT [#fontnumber, [scale, [reverse]
FONT LOAD filename$ AS #fontnumber
FONT UNLOAD #fontnumber
```

Comments:

`#fontnumber` is the font number in the range of 1 to 10, the # symbol is optional.

`scale` is the multiply factor in the range of 1 to 8 (e.g. a scale of 2 will double the size of a pixel in both the vertical and horizontal). Default is 1.

If `reverse` is a number other than zero the font will be displayed in reverse video. Default is no reverse.

There are three fonts built into MM-BASIC:

#1 is the standard font of 10 x 5 pixels containing the full ASCII set.

#2 is a larger font of 16 x 11 pixels also with the full ASCII set.

#3 is a jumbo font of 30 x 22 pixels consisting of the numbers zero to nine and the characters plus, minus, comma and full stop.

#4 … #10 can be used for custom fonts

Font #1 with a scale of one and no reverse is the default on power up and will be reinstated whenever control returns to the input prompt. So if you execute `FONT` statement at command prompt there will be no font change.

`filename$` is file name where the font is contained.

The font is loaded into the memory area used by arrays and strings, use the `MEMORY` command to check the usage of this area.

A font file is just a text file containing ordinary characters which are loaded line by line to build the bitmap of each character in the font. Each character can be up to 64 pixels high and 255 pixels wide.

Up to 255 characters can be defined.

The first non comment line in the file must be the specifications for the font as follows:
height,width, start, end

Where height and width are the size of each character in pixels, start is the number in the ASCII chart where the first character sits and end is the last character. Each number is separated by a comma. So, for example, 16, 11, 48, 57 means that the font is 16 pixels high and 11 wide. The first character is decimal 48 (the zero character) and the last is 57 (number nine character).

The remainder of the lines specify the bitmap for each character.

Each line represents a horizontal row of pixels. A space means the pixel is not illuminated and any other character will turn the pixel on. If the font is 11 pixels wide there must be 11 characters in the line. The first line is the top row of pixels in the character, the next is the second and so on. If the character is 16 pixels high there must be 16 lines to define the character. This repeats until each character is drawn. Using the above example of a font 16x11 with 10 characters there must be a total of 160 lines with each line 11 characters wide. This is in addition to the specification line at the top.

A comment line has an apostrophe (') as the first character and can occur anywhere. A comment line is completely ignored, all other lines are significant.

You can unload every loaded font to free the memory.

You cannot unload the built in fonts.

`FONT LOADB filename$ AS #fontnumber` is a non-documented feature. Using this command it is possible to load a binary format font. Only system level developers should use this feature.

Examples:

```
10 FONT #3,2,1 'doubles scale and reverse video of font #3

10 FONT 3,2,1 'same as above
```

The following example creates two arrow icons, up and down arrow. Each is 11x11 pixels with the first (up arrow) in the position of the zero character (0) and the down arrow in the position of number one (1). To display a up arrow your program would contain this:

```
10 FONT LOAD "ARROWS.FNT" AS #9 'load the font
20 FONT #9
30 PRINT "0"

'EXAMPLE OF FONT FILE
'ARROWS.FNT
11,11,48,49

     x
   xxxxx
 xxxxxxxxx
     x
     x
     x
     x
     x
     x
     x
     x
     x
     x
   xxxxx


   xxxxx
     x
     x
     x
     x
     x
     x
     x
     x
     x
     x
 xxxxxxxxx
   xxxxx
     X

'END OF FONT
```

2.50 FOR ... NEXT

Purpose:
To execute a series of instructions a specified number of times in a loop.

Syntax:
```
FOR variable=x TO y [STEP z]
 [statements]
NEXT [variable][,variable...]
```

Comments:

`variable` is used as a counter.

`x`, `y`, and `z` are numeric expressions.

`STEP z` specifies the counter increment for each loop.

The first numeric expression (`x`) is the initial value of the counter. The second numeric expression (`y`) is the final value of the counter.

Program lines following the `FOR` statement are executed until the `NEXT` statement is encountered. Then, the counter is incremented by the amount specified by `STEP`.

If `STEP` is not specified, the increment is assumed to be 1.

A check is performed to see if the value of the counter is now greater than the final value (`y`). If it is not greater, MM-BASIC branches back to the statement after the `FOR` statement, and the process is repeated. If it is greater, execution continues with the statement following the `NEXT` statement. This is a `FOR-NEXT` loop.

The body of the loop is skipped if the initial value of the loop times the sign of the step exceeds the final value times the sign of the step.

If `STEP` is negative, the final value of the counter is set to be less than the initial value. The counter is decremented each time through the loop, and the loop is executed until the counter is less than the final value.

Nested Loops:
`FOR-NEXT` loops may be nested; that is, a `FOR-NEXT` loop may be placed within the context of another `FOR-NEXT` loop. When loops are nested, each loop must have a unique variable name as its counter.

The `NEXT` statement for the inside loop must appear before that for the outside loop.

If nested loops have the same end point, a single `NEXT` statement may be used for all of them.

The variable(s) in the `NEXT` statement may be omitted, in which case the `NEXT` statement will match the most recent `FOR` statement.

If a `NEXT` statement is encountered before its corresponding `FOR` statement, a "Cannot find a matching FOR" error message is issued and execution is terminated.

If a `NEXT` variable statement is encountered before its corresponding `FOR` variable statement, a "FOR without matching NEXT" error message is issued and execution is terminated.

Examples:The following example prints integer values 1 to 10.

```
20 FOR I =1 TO 10 STEP 1
30 PRINT I;
40 NEXT I
RUN
1 2 3 4 5 6 7 8 9 10
```

In the following example, the loop does not execute because the initial value of the loop exceeds the final value. Nothing is printed by this example.

```
10 R=0
20 FOR S=1 TO R
30 PRINT S
40 NEXT S
```

In the next example, the loop executes 10 times. The final value for the loop variable is always set before the initial value is set.

```
10 S=5
20 FOR S=1 TO S+5
30 PRINT S;
40 NEXT S
RUN
1 2 3 4 5 6 7 8 9 10
```

In the next example, the NEXT variable is wrong and error occurs.

```
10 FOR I=1 TO 5
20 PRINT I;
30 NEXT S
RUN
1
Error line 30: Cannot find variable
>
```

2.51 GOSUB ... RETURN

Purpose:
To branch to, and return from, a subroutine.

Syntax:
`GOSUB line number` *or* *`GOSUB variable`*
`RETURN`

Comments:

`line number` is the first line number of the subroutine.

A subroutine may be called any number of times in a program, and a subroutine may be called from within another subroutine.

A `RETURN` statement in a subroutine causes MM-BASIC to return to the statement following the most recent `GOSUB` statement. A subroutine can contain more than one `RETURN` statement, should logic dictate a `RETURN` at different points in the subroutine.

Subroutines can appear anywhere in the program, but must be readily distinguishable from the main program.

To prevent inadvertent entry, precede the subroutine by an `END`, or `GOTO` statement to direct program control around the subroutine.

Added in the CAN version of DMBasic is the possibility to use a variable for the branch. This variable must be defined as an array by the `DIM` statement before. The array must be filled with executable BASIC statements, including the `RETURN` statement. The filling of the array can be done by the modified `LOAD` command.

Examples:

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
40 PRINT "SUBROUTINE";
50 PRINT " IN";
60 PRINT " PROGRESS"
70 RETURN
RUN
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
```

The `END` statement in line 30 prevents re-execution of the subroutine.

This is an example of `GOSUB` by variable name:

```
10 DIM sub_rt(100): REM 400 bytes available for subroutine
20 LOAD "sub_rt.bas": REM .bas is default extension and can
be deleted
30 GOSUB sub_rt
40 PRINT "End program"
```

The subroutine "sub_rt.bas" has to be developed separately and in principle earlier:

```
10 PRINT "This is the subroutine"
20 RETURN
```

Stand-alone this can be tested by `GOSUB 10`. `RUN` will generate an error message (RETURN without GOSUB). Program can be written with `SAVE "sub_rt.bas"`.

Execution of the main program will give the following result:

```
RUN
This is the subroutine
End program
>
```

2.52 GOTO

Purpose:
To branch unconditionally out of the normal program sequence to a specified line number.

Syntax:
`GOTO line number`

Comments:

`line number` is any valid line number within the program.

If `line number` is an executable statement, that statement and those following are executed. If it is an on-executable statement, execution proceeds at the first executable statement encountered after `line number`.

Examples:

```
10 READ R
20 PRINT "R ="; R;
30 A = 3.14*R^2
40 PRINT "AREA ="; A
50 GOTO 10
60 DATA 5, 7, 12

RUN
R = 5 AREA = 78.5
R = 7 AREA = 153.86
R = 12 AREA = 452.16
Error line 10: No more DATA to read
```

The "No more DATA to read" advisory is generated when the program attempts to read a fourth `DATA` statement (which does not exist) in line 60.

2.53 I2CDIS

Purpose:
Disable the I2C port, which has been enabled as master.

Syntax:
`I2CDIS`

Comments:

It will also send a stop if the bus is still held.

2.54 I2CEN

Purpose:
Enable the I2C port as master.

Syntax:
```
I2CEN speed, timeout [,interrupt-line]
```

Comments:

`speed` is a value between 10 and 400 (for bus speeds 10kHz to 400kHz).

If you do not need the higher speeds then operating at 100kHz is the safest choice.

`timeout` is a value in milliseconds after which the master send and receive commands will be interrupted if they have not completed. The minimum value is 100. A value of zero will disable the timeout (though this is not recommended).

`interrupt-line` is optional. It specifies the line number of an interrupt routine to be run when the send or receive command completes. If this is not supplied, the send and receive command will only return when they have completed or timed out. If it is supplied then the send and receive will complete immediately and the command will execute in the background.

The interrupt routine operates the same as a general interrupt on an external I/O pin and must be terminated with an `IRETURN` command to return control to the main program when completed.

2.55 I2CRCV

Purpose:
Receive data as master from a slave, with the option to send data first.

Syntax:
`I2CRCV address, bus-hold, rcv-len, rcv-buf [, snd-len, snd-data]`

Comments:

`address` is the slave `I2C` address (note that 10 bit addressing is not supported).

`Option` is a number between 0 and 3; 1 = keep control of the bus after the command (a stop condition will not be sent at the completion of the command); 2 = treat the address as a 10 bit address; 3 = combine 1 and 2 (hold the bus and use 10 bit addresses).

`rcv-len` is the number of bytes to receive.

`rcv-buf` is the variable to receive the data - this is a one dimensional array or if `rcv-len` is 1 then this may be a normal variable. The array subscript does not have to be zero and will be honoured, also bounds checking is performed.

Optionally you can specify data to be sent first using `snd-len` and `snd-data`. These parameters are used the same as in the `I2CSEND` command (ie, `snd-data` can be a constant, an array or a string variable).

Examples:

```
I2CRCV &H6F,1,1,BYTE

I2CRCV &H6F,1,5, ARR(0)

I2CRCV &H6F,1,4,ARR(2),3,&H12,&H34,&H56

I2CRCV &H6F,1,3,RCVARRAY(0),4,SNDARRAY(0)
```

The automatic variable `MM.I2C` (description at I2CSRCV) will hold the result of the transaction.

2.56 I2CSDIS

Purpose:
Disable the I2C port, which has been enabled as slave.

Syntax:
`I2CSDIS`

Comments:

2.57 I2CSEN

Purpose:
Enable the I2C port as slave.

Syntax:
`I2CSEN address, mask, option, send-int-line, rcv-int-line`

Comments:

Enables the `I2C` module in slave mode.

`address` is the slave `I2C` address

`mask` is the address mask (bits set as 1 will always match)

`option` is a number between 0 and 3; 1 = allows MM-BASIC to respond to the general call address. When this occurs the value of `MM.I2C` will be set to 4; 2 = treat the address as a 10 bit address; 3 =combine 1 and 2 (respond to the general call address and use 10 bit addresses).

`send-int-line` is the line number of a send interrupt routine to be invoked when the module has detected that the master is expecting data.

`rcv-int-line` is the line number of a receive interrupt routine to be invoked when the module has received data from the master.

2.58 I2CSEND

Purpose:
Send data as master to a slave

Syntax:
`I2CSEND address, option, snd-len, snd-data [,snd-data]`

Comments:

`Address` is the slave `I2C` address.

`option` is a number between 0 and 3, 1 = keep control of the bus after the command (a stop condition will not be sent at the completion of the command); 2 = treat the address as a 10 bit address; 3 =combine 1 and 2 (hold the bus and use 10 bit addresses).

`snd-len` is the number of bytes to send.

`snd-data` is the data to be sent - this can be specified in various ways (all values sent will be between 0 and 255):

The data can be supplied in the command as individual bytes.
Example:

```
I2CSEND &H6F,1,3,&H23,&H43,&H25
```

The data can be in a one dimensional array (the subscript does not have to be zero and will be honoured, also bounds checking is performed).
Example:

```
I2CSEND &H6F,1,3,ARRAY(0)
```

The data can be a string variable (not a constant).
Example:

```
I2CSEND &H6F,1,3,STRING$
```

The automatic variable `MM.I2C` (description at I2CSRCV) will hold the result of the transaction.

2.59 I2CSRCV

Purpose:
Receive data as a slave from the master

Syntax:
`I2CSRCV rcv-len, rcv-buf, rcv-d`

Comments:

This command should be used in the receive interrupt (ie in the `rcv-int-line` when the master has sent some data).

Alternatively a flag can be set in the receive interrupt routine and the command invoked from the main program loop when the flag is set.

`rcv-len` is the maximum number of bytes to receive.

`rcv-buf` is the variable to receive the data - this is a one dimensional array or if `rcv-len` is 1 then this may be a normal variable. The array subscript does not have to be zero and will be honoured, also bounds checking is performed.

`rcv-d` will contain actual number of bytes received by the command.

I2C Automatic Variable `MM.I2C` is set to indicate the result of an I2C operation.

0 = The command completed without error.
1 = Received a NACK response
2 = Command timed out
4 = Received a general call address (when in slave mode)

2.60 I2CSSEND

Purpose:
Send data as slave to the master

Syntax:
```
I2CSSEND snd-len, snd-data [,snd-data]
```

Comments:

This command should be used in the send interrupt (ie in the `snd_int-line` when the master has requested data).

Alternatively a flag can be set in the send interrupt routine and the command invoked from the main program loop when the flag is set.

`snd-len` is the number of bytes to send.

`snd-data` is the data to be sent. This can be specified in various ways, see the `I2CSEND` commands for details.

2.61 IF

Purpose:
To make a decision regarding program flow based on the result returned by an expression.

Syntax:
```
IF expression THEN statement [ELSE statement]
IF expression GOTO line number [ELSE statement]

IF expression THEN
 statement(s)
[ELSE
 statement(s)]
[ENDIF]

IF expression THEN
 statement(s)
[ELSEIF expression THEN
 statement(s)]
[ENDIF]
```

Comments:

If the result of `expression` is non zero (logical true), the `THEN` or `GOTO` line number is executed.

If the result of `expression` is zero (false), the `THEN` or `GOTO` line number is ignored and the `ELSE`, if present, is executed. Otherwise, execution continues with the next executable statement.

A comma is allowed before `THEN` and `ELSE` (not before `ELSE` in the multi line).

`THEN` and `ELSE` may be followed by a line number for branching in the single line command instead of a statement.

`THEN` and `ELSE` may also be followed by one or more statements to be executed in the multi line.

`GOTO` is always followed by a line number.

If the statement does not contain the same number of `ELSE`'s and `THEN`'s, each `ELSE` is matched with the closest unmatched `THEN`. For example:
`IF A=B THEN IF B=C THEN PRINT "A=C" ELSE PRINT "A < > C"`
will not print "`A < > C`" when A <> B.

If an `IF...THEN` statement is followed by a line number in the direct mode, an "`Invalid line number`" error results, unless a statement with the specified line number was previously entered in the indirect mode.

If the `THEN statement` is on one line also the `ELSE` clause must be on that line.

Examples:

In the following example, a test determines if N is greater than 10 and less than 20. If N is within this range, execution branches to line 200. If N is not within this range, execution continues with line 110.

```
100 IF(N<20) and (N>10) THEN 200
110 PRINT "OUT OF RANGE"
```

This multi line example will check A to be smaller, equal or greater than 5.

```
100 FOR A = 3 TO 7
110 PRINT A
120 IF A < 5 THEN
130 PRINT "A < 5"
140 ELSEIF A = 5 THEN
150 PRINT "A = 5"
160 ELSE
170 PRINT "A > 5"
180 ENDIF
190 NEXT
```

2.62 INPUT

Purpose:
To prepare the program for input from the terminal during program execution or to read data items from a sequential file and assign them to program variables.

Syntax:
```
INPUT [prompt string;] list of variables
INPUT [prompt string,] list of variables
INPUT #filenumber, list of variables
```

Comments:

`prompt string` is a request for data to be supplied during program execution.

`list of variables` contains the variable(s) that stores the data in the prompt string.

Each data item in the prompt string must be surrounded by double quotation marks, followed by a semicolon or comma and the name of the variable to which it will be assigned. If more than one variable is given, data items must be separated by commas.

The data entered is assigned to the variable list. The number of data items supplied must be the same as the number of variables in the list.The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item input must agree with the type specified by the variable name.

If more than the list of variables is entered only the first are used and the extra inputs are ignored.
```
INPUT A,B,C? 1,2,3,4,5
```
will assign A=1 B=2 C=3 and the 4 and 5 entries will be ignored.

If the variable is number but string is entered then 0 will be assigned to the variable
```
INPUT A,B,C? 1,hi
```
will assign A=1 B=0 C=0.

A comma may be used instead of a semicolon after prompt string to suppress the question mark. For example, the following line prints the prompt with no question mark:
```
INPUT "ENTER BIRTHDATE",B$
```

When an `INPUT` statement is encountered during program execution, the program halts, the prompt string is displayed, and the operator types in the requested data. When the operator presses the ENTER key, program execution continues.atement

In `INPUT #file number, list of variables` file number is the number used when the file was opened for input.

`variable list` contains the variable names to be assigned to the items in the file. The data items in the file appear just as they would if data were being typed on the keyboard in response to an INPUT statement.

With `INPUT #`, no question mark is printed, as it is with `INPUT`.

For numeric values, leading spaces and line feeds are ignored. The first character encountered (not a space or line feed) is assumed to be the start of a number. The number terminates on a space, carriage return, line feed, or comma.

If MM-BASIC is scanning the sequential data file for a string, leading spaces and line feeds are ignored.

If the first character is a double quotation mark ("), the string will consist of all characters read between the first double quotation mark and the second. A quoted string may not contain a double quotation mark as a character. The second double quotation mark always terminates the string.

If the first character of the string is not a double quotation mark, the string terminates on a comma, carriage return, line feed, or after 255 characters have been read.

If end of the file is reached when a numeric or string item is being `INPUT`, the item is terminated.

Examples:

To find the square of a number:

```
10 INPUT X
20 PRINT X "SQUARED IS" X^2
30 END
RUN
?
```

The operator types a number (5) in response to the question mark.

```
5 SQUARED IS 25
```

To find the area of a circle when the radius is known:

```
10 PI=3.14
20 INPUT "WHAT IS THE RADIUS"; R
30 A=PI*R^2
40 PRINT "THE AREA OF THE CIRCLE IS"; A
50 PRINT
60 GOTO 20
RUN
WHAT IS THE RADIUS? 7.4
THE AREA OF THE CIRCLE IS 171.9464
```

2.63 IRETURN

Purpose:
To return from an interrupt.

Syntax:
`IRETURN`

Comments:

The `IRETURN` statement causes MM-BASIC to branch back to the statement where interrupt occurs.

A subroutine may contain more than one `IRETURN` statement to return from different points in the subroutine.

Subroutines may appear anywhere in the program.

2.64 KILL

Purpose:
To delete a file from a disk.

Syntax:
```
KILL filename$
```

Comments:

`filename$` can be a program file or data file.

`KILL` is used for all types of disk files.

You must specify the filename's extension when using the KILL command. Remember that files saved in MM-BASIC are given the default extension .BAS.

If a `KILL` command is given for a file that is currently open, no error will occurs.

Examples:

The following command deletes the MM-BASIC file DATA1, and makes the space available for reallocation to another file:

```
200 KILL "DATA1.BAS"
```

The following command deletes the MM-BASIC file RAINING from the subdirectory dogs:

```
KILL "CATS\DOGS\RAINING.BAS"
```

2.65 LET

Purpose:
To assign the value of an expression to a variable.

Syntax:
`[LET] variable=expression`

Comments:

The word `LET` is optional; that is, the equal sign is sufficient when assigning an expression to a variable name.

The `LET` statement is seldom used. It is included here to ensure compatibility with previous versions of BASIC that require it.

When using `LET`, remember that the type of the variable and the type of the expression must match. If they don't, an error occurs.

```
A = "Hello"
Error: Expected a number
```

Example 1:The following example lets you have downward compatibility with an older system. If this downward compatibility is not required, use the second example, as it requires less memory.

```
110 LET D=12
120 LET E=12^2
130 LET F=12^4
140 LET SUM=D+E+F
```

Example 2:

```
110 D=12
120 E=12^2
130 F=12^4
140 SUM=D+E+F
```

2.66 LINE

Purpose:
To draw lines and boxes on the screen.

Syntax:
`LINE [(x1,y1)]-(x2,y2) [,[color][,B[F]]`

Comments:

`x1,y1` and `x2,y2` specify the end points of a line.

`color` draw with white if nonzero

`B` (box) draws a box with the points (x1,y1) and (x2,y2) at opposite corners.

`BF` (filled box) draws a box (as `,B`) and fills in the interior with points.

If `x1,y1` is not specified last `x,y` coordinates are used.

```
LINE (100,100)-(200,200),1,BF 'draw box 100,100,200,200
LINE -(300,100),1,BF 'draw box 200,200,300,100
```

Examples:

```
LINE (MM.HRES \2,0)-(MM.HRES \2,MM.VRES)
```
Draws a vertical line which divides the screen in half from top to bottom.

```
LINE (0,MM.VRES \2)-(MM.HRES,MM.VRES \2)
```
Draws a horizontal line which divides the screen in half from left to right.

```
LINE (0,0)-(MM.HRES,MM.VRES)
```
Draws a diagonal line from the top left to lower right corner of the screen.

```
LINE (10,10)-(20,20),1
```
Draws a line from 10,10 to 20,20 with white color.

```
LINE (10,10)-(20,20),0
```
Draws a line from 10,10 to 20,20 with black color (erases the line drawn above).

```
10 CLS
20 LINE -(RND(1)*MM.HRES,RND(1)*MM.VRES)
30 PAUSE RND(1)*100
40 GOTO 20
```
Draw random lines with random speed.

```
10 CLS
20 FOR I = 1 TO MM.HRES STEP 5
30 LINE (0,0)-(I,MM.VRES)
40 LINE (MM.HRES,0)-(I,MM.VRES)
50 NEXT
60 DO : LOOP UNTIL INKEY$ = " "
```

Draws pattern and waits until SPACE is pressed.

2.67 LINE INPUT

Purpose:
To input an entire line (up to 255 characters) from the keyboard or from a sequential disk file into a string variable, ignoring delimiters.

Syntax:
```
LINE INPUT [prompt$] [;] [,] string-variable
LINE INPUT #filenumber, string-variable
```

Comments:

`prompt$` is a string literal, displayed on the screen, that allows user input during program execution.

A question mark is not printed no matter if the delimiter is [;] or [,] unless it is part of `prompt$`.

`string-variable` accepts all input from the end of the prompt to the carriage return. Trailing blanks are not ignored.

`filenumber` is the number under which the file was opened.

`LINE INPUT` is almost the same as the `INPUT` statement, except that it accepts special characters (such as commas) in operator input during program execution.

If a line-feed/carriage return sequence (this order only) is encountered, both characters are input and echoed. Data input continues.

A `LINE INPUT` may be escaped by typing CTRL-BREAK. MM-BASIC returns to command level and displays '>'.

Typing CONT resumes execution after the `LINE INPUT` line.

`LINE INPUT #` is especially useful if each line of a data file has been broken into fields, or if a BASIC program saved in ASCII mode is being read as data by another program.

Examples:

```
100 LINE INPUT A$
```

Program execution pauses at line 100, and all keyboard characters typed thereafter are input to string A$ until ENTER, CTRL-M, CTRL-C, or CTRL-BREAK is entered.

```
10 OPEN "INFO.TXT" AS OUTPUT #1
20 LINE INPUT "CUSTOMER INFORMATION?"; C$
30 PRINT #1, C$
40 CLOSE #1
50 OPEN "INFO.TXT" AS INPUT #1
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE #1
RUN
CUSTOMER INFORMATION?
```

If the operator enters LINDA JONES 234, 4 MEMPHIS
then the program continues with the following:

```
LINDA JONES 234, 4 MEMPHIS
```

2.68 LIST

Purpose:
To list all or part of a program to the screen, line printer, or file.

Syntax:
```
LIST [linenumber][-linenumber]
LIST [linenumber-]
LIST #filenumber, linenumber[-linenumber]
```

Comments:

`linenumber` is a valid line number within the range of 0 to 65000.

Use the hyphen to specify a line range. If the line range is omitted, the entire program is listed.

`linenumber-` lists that line and all higher numbered lines. `-linenumber` lists lines from the beginning of the program through the specified line.

Any listing may be interrupted by pressing CTRL-BREAK. If the lines are more than size of the screen the list will display first lines then "PRESS ANY KEY ..." message will be displaying and listing will continue after key is pressed for the next screen of lines.

`LIST` has been extended in the CAN version with an optional `#` (`LIST #filenumber, ...`) E.g. `LIST #1,1-100` will save the lines 1-100 to the file opened for output as #1. If the port no. is included also a line specification has to be included, e.g. `LIST #1,1-` for the complete listing. In this way only a part of the total program can be saved.

Examples:

```
LIST
```
Lists all lines in the program.

```
LIST -20
```
Lists lines 1 through 20.

```
LIST 10-20
```
Lists lines 10 through 20.

```
LIST 20-
```
Lists lines 20 through the end of the program.

2.69 LOAD

Purpose:To load a file from file into memory.

Syntax:
`LOAD filename$`
`LOAD # filename$`

Comments:

`filename$` is the filename used when the file was saved. If the extension was omitted, .BAS will be used.

`LOAD` closes all open files and deletes all variables and program lines currently residing in memory before it loads the designated program. Second `"` could be omited.

`LOAD # filename$` has been added in the CAN version of the software. It will load a file which has been saved with the `SAVE #` option (see `SAVE` for details).

Examples:

```
LOAD "DEMO
```
Loads the file DEMO.BAS.

```
LOAD # "DEMO_2.BAS"
```
Loads the file DEMO_2.BAS which was saved by `SAVE #`.

2.70 LOCATE

Purpose:
To move the cursor to the specified X,Y pixel position on the screen. Subsequent PRINT command will place its output at this location

Syntax:
LOCATE X,Y

Comments:

X is the horizontal pixel position with value from 0 to MM.HRES.

Y is the vertical pixel position with value from 0 to MM.VRES.

If X value is over MM.HRES X will be assigned to 0.

If Y value is over MM.VRES Y will be assigned to MM.VRES.

Example:

```
10 LOCATE 100,100
20 PRINT "HI" ' print HI on position 100,100
```

2.71 LOOP

Purpose:
To branch back to the beginning of a loop started by `DO`.

Syntax:

```
DO [ loop statements ] LOOP
DO WHILE expression [loop statements] LOOP
DO [loop statements] LOOP UNTIL expression
```

Comments:

See `DO` command for further details

2.72 MEMORY

Purpose:
List the amount of memory currently in use. *Extended in CAN version with range and video memory. Also debug options included.*

Syntax:
`MEMORY [debug parameter]`

Comments:

Program memory is cleared by the `NEW` command. Variable, array and string memory spaces are cleared by many commands (eg, `NEW`, `RUN`, `LOAD`, etc) as well as the specific commands `CLEAR` and `ERASE`.

The added address range are the logical addresses which can also be accessed by the `PEEK` and `POKE` statements. Please be careful in using them.

Video memory is also included. If the video is disabled in `SETUP`, Video is replaced by Extended. This RAM memory can be used in this case by the array `_EXTMEM` or the string array `_EXTMEM$`. The maximum sizes are: `DIM _EXTMEM(6475)` or `DIM _EXTMEM$(100)`.

Debugging the memory has been made easier than by using `PEEK`. `MEMORY debug parameter` lists ranges of memory. `debug parameter` can have the following values: p (program area), v (variable area), e (extended or video area), a (array/string area), r (the complete RAM area), f (the complete flash area) and d (the drive A: area). Only reading is possible with the `MEMORY` command. Writing still has to be done using `POKE` (not the areas of the options f and d).

Examples:

```
>MEMORY
  19kB ( 64%) Program  memory [0xA00033B8 - 0xA000ABB8] used
   2kB ( 14%) Variable memory [0xA000AFE0 - 0xA000E690] used
  26kB (100%) Video    memory [0xA000F5AC - 0xA0015AEC] used
   6kB ( 17%) Arr/Str  memory [0xA0015FC0 - 0xA001E7C0] used
>

>MEMORY a
A001-5FC0  00 00 00 00 00 …
A001-5FD0  00 40 C6 45 00 …
……
A001-61B0  00 00 00 00 00 …

Enter any key for next page; <CTRL>-C to stop
```

2.73 MERGE

Purpose:
Adds program lines from `filename$` to the program in memory. Unlike `LOAD`, it does not clear the program currently in memory.

Syntax:
```
MERGE filename$
```

Comments:

`filename$` is a valid string expression containing the filename. If no extension is specified, then MM-BASIC assumes an extension of .BAS.

The SD card is searched for the named file. If found, the program lines on the SD card are merged with the lines in memory. After the `MERGE` command, the merged program resides in memory, and MM-BASIC returns to the direct mode.

If any line numbers in the file have the same number as lines in the program in memory, the lines from the file replace the corresponding lines in memory.

Examples:

```
MERGE "CODE.BAS"
```

Merges the file code.bas with the program currently in memory, provided code.bas was previously saved.

2.74 MKDIR

Purpose:
To create a subdirectory.

Syntax:
```
MKDIR pathname$
```

Comments:

`pathname$` is a string expression, identifying the subdirectory to be created.

Examples:

```
MKDIR "SALES\JOHN"
```

Creates the subdirectory JOHN within the directory of SALES.

2.75 MM.BLANK

Purpose:
To set a time for the video screen saver

Syntax:
`MM.BLANK seconds`

Comments:

`seconds` *is the time in seconds after which the display goes blank.*

`MM.BLANK` *was available in the original DMBASIC 2.7, however not documented.*

`MM.BLANK` *can only be used if the screen saver is enabled in* `SETUP` *(n option; any value 5-60 is OK)*

The value of `MM.BLANK` *is overruled by the value of* `SETUP` *if a key is pressed on the keyboard.*

`MM.BLANK` *can be used to temporary blank the display*

Example:

This example will show MESSAGE on the display during 10 seconds; after that 10 seconds blank; 10 seconds on; 10 seconds off ...

```
10 MM.BLANK 10
20 CLS
30 FONT 2,4
40 PRINT "MESSAGE"
50 PAUSE 20000
60 GOTO 10
```

2.76 MSDOFF

Purpose:
To switch off the availability of the SDCard file system to the terminal system

Syntax:
MSDOFF

Comments:

MSDOFF was available in the original DMBASIC 2.7, however not documented.

MSDOFF was meant for switching off the terminal system to the SDCard, however the command is useless, because MSDON, used to switch on the system, will always wait for a key to be pressed. Even if MSDOFF is situated in an interrupt routine, it will never be activated.

2.77 MSDON

Purpose:
To switch on the availability of the SDCard file system to the terminal system

Syntax:
`MSDON`

Comments:

`MSDON` *was available in the original DMBASIC 2.7, however not documented.*

`MSDON` *will stop all local control until a key is pressed. Meanwhile the terminal system gets control over the SDCard is if it is an external drive to the system. Read and write are possible.*

2.78 NAME

Purpose:
To change the name of a disk file.

Syntax:
```
NAME oldfilename$ AS newfilename$
```

Comments:

`oldfilename$` must exist and `newfilename$` must not exist; otherwise, an error results.

After a `NAME` command, the file exists on the same disk, in the same disk location, with the new name.

Examples:

```
NAME "ACCTS" AS "LEDGER"
```

The file formerly named ACCTS will now be named LEDGER. The file content and physical location on the disk is unchanged. If LEDGER exist “Error: Cannot open file“ results.

2.79 NEW

Purpose:
To delete the program currently in memory and clear all variables.

Syntax:
`NEW`

Comments:

`NEW` is entered at command level to clear memory before entering a new program. MM-BASIC always returns to command level after a `NEW` is executed.

Examples:

```
NEW
```

or

```
980 PRINT "Do You Wish To Quit (Y/N)
990 ANS$=INKEY$: IF ANS$=""THEN 990
1000 IF ANS$="Y" THEN NEW
1010 IF ANS$="N" THEN 980
1020 GOTO 990
```

2.80 NEXT

Purpose:
To branche back to the first statement in the `FOR … NEXT` loop.

Syntax:

```
FOR variable=x TO y [STEP z]
 [statements]
NEXT [variable][,variable...]
```

Comments:

See description at `FOR`.

2.81 NUM2BYTE

Purpose:
To convert a float number to 4 separate bytes

Syntax:
```
NUM2BYTE number, array(x)
```
or
```
NUM2BYTE number, variable1, variable2, variable3, variable4
```

Comments:

MM-BASIC numbers are stored as the C float type and are four bytes in length. The bytes can be returned as four separate variables, or as four elements of `array` starting at index `x`.

`NUM2BYTE` is very useful at the I2C send and receive commands, but can also be used in other applications.

2.82 OLED

Purpose:
To control either a MOD-OLED-128x64 display or a MOD-LCD2.8RTP display and touch screen.

Syntax:
`OLED [options[,horizontal offset[,vertical offset]]]`
or
`OLED options_ext`

Comments:

`options` can have the values of 0 to 255 and is used for the MOD-OLED-128x64. These options are commented first.

`OLED` is an addition to the I2C commands, which are standard available in DMBasic 2.7. It only makes sense if an 128x64 OLED display is connected to the I2C interface. `OLED` copies an area of the video display to the OLED display. By default this is the upper left corner. 128 pixels of the total of 480 horizontal and 64 pixels of the total of 432 vertical (PAL video).

Horizontal offset can be given in steps of 32 pixels. 0 up to 11 are the possible values, with 11 being the far right position. To use `horizontal offset`, the `options` parameter has to be filled in (0 by default). Vertical offset can be given in steps of 8 pixels. 0 up to 46 are the possible values, with 46 being the bottom position. To use `vertical offset` both `options` and `horizontal offset` have to be filled in (0 by default).

Specifications of the resolution are based on a module using VGA (e.g. Duinomite Mega). If no VGA is available (e.g. PIC32_T795) resolution is decreased to 304x216. Therefore the possible offsets are also decreased to a maximum of 5 for horizontal and a maximum of 19 for vertical. Please be aware of the automatic Carriage Return at a longer text line.

`options` is the parameter which sets the possible visualisation options of the 128x64 display. It is done by an eight bit number (0-255) where all bits represent a special feature. The number can be represented by <arbzfivv>

vv (0-3): visibility 0=normal; 1=low intensity; 2=high intensity; 3=blank
i (4): inverse; pixels are black on a white background
f (8): flip; pixels are vertical flipped
z(16): zoom; vertical pixels are doubled
b(32): blink; display is blinking with fade out
r(64): re-init; initialisation of the display is normally only done once; now re-init and display.
a(128): alternate address: the display uses standard address 3C; by hardware modification this can be changed to 3D. In this way it is possible to use 2 displays with different areas of the video display.

`options_ext` can have a value of 256-65535. In this case `horizontal offset` and `vertical offset` will result in an error message.

`OLED` is in fact a bad name for the command, because when used with the `options_ext` parameter it does not support an OLED display, but an LCD display with touch screen. However it is decided not to introduce a new command, which would have resulted in more changes in the original software. Also a change of the name `OLED` would have resulted in incompatibility with older Basic programs.

The 8 most significant bits of the OLED extended options parameter can be represented by: <rlsoipmc>. Whenever a bit is set (1) and the bits left are cleared (0) all bits to the right of this bit are used as sub parameter for the specific range of values of the options parameter.
<c>onfiguration: This has to be done before any other extended OLED command.
<m>ove: This will copy the upper left corner of the video display to the LCD display
<p>oll: This will poll the touch screen if enabled
<i>nterrupt: This will set an interrupt time to poll the touch screen in the background
<o>ffset: This will set an offset for the touch screen both in x and y direction
<s>caling: This will set a scaling for the touch screen both in x and y direction
<l>ineno: This will set a Basic lineno to jump to if a touch is detected
<r>eal-time: This will convert the LCD display into a real display

The hardware uses two communication ports on the UEXT connector. The LCD display communicates over the SPI bus. The communication lines (MISO/MOSI) are the same as used by the SDcard on the Duinomite, only the chip select is different. Therefore the SDcard is disabled by configuring the LCD display. Whenever the SDcard should be used again, the display should be disabled and after the use of the the SDcard enabled again.

The LCD display can be used together with a monitor, VGA or composite video. If a VGA monitor is connected, it will be automatically detected and the resolution will be set to 480x432. If it is not connected the resolution will be decreased to 304x216. By normal use of the LCD display the resolution will be further reduced to the size of the display. This is depending on the use as portrait or landscape and also if it is used with double sized pixels. If used together with an extra monitor one can decide to use the original monitor resolution. In that case a part of information can be off-screen on the LCD display. During configuration and move commands the monitor will be switched off for a few seconds.

The touch screen uses the I2C bus. This is the same bus as the original OLED display uses. Both the OLED display and the touch screen are slaves on the I2C bus and are configured on different addresses (0x3C or 0x3D for the OLED, 0x4D for the touch screen). So it is possible to use both interfaces parallel (using both UEXT ports on the Duinomite or using the UEXT extension module of Olimex). The original OLED command will address the OLED module, or modules (if both addresses are used) and the extended OLED command will address the touch screen. It is however not allowed to use the interrupt for polling in this situation.

A little modification has been made in the SPI command. The LCD display uses a D/C (Data/Command) line, which is placed on the MISO pin of the UEXT connector (the display itself does not produce any data for the master). The original Basic however wants the MISO pin (PIN 9) configured as input to use the SPI command. If the MOSI pin (PIN 8) and the CLK pin (PIN 10) and the Chip Select pin (PIN 19) are defined as output, it is also possible now to define the MISO pin (PIN 9) as output. In this case the MISO pin becomes the D/C pin and whenever txdata of the format 0x1yy is given it is interpreted as command yy.

`options_ext` values 256 - 511 are used for initialisation and setting the basic configuration. An `OLED` command with one of these values should be occured before any of the other `OLED` commands for the LCD display can be used. As the value of the `options_ext` parameter is 9 bits wide, the 8 LSB's can be used now for further specific settings. For normal use of the LCD display, including the touch screen, OLED 256 will do the job. If the command is given in this format in a cold startup situation, the display will be filled with pixels in all kind of colors. The local video memory will have random values, which are displayed.
The other 8 bits have the following meaning: <btnnrsid>:
<d> or +1: If this bit is set, the display turns off. This will result in a full white screen. The local video memory will not be changed, so if the configuration command is given again with the <d> bit cleared, the original display will reappear. It is also possible to change the actual video memory using other commands, during the display off situation. When turning on again the new information is displayed.
<i> or +2: If this bit is set, the display is inverted. Black becomes white and also all other colors will get the inverse value.
<s> or +4: If this bit is set, scrolling of the screen is activated. This only makes sense if the real-time modus will be used. It will actually scroll the screen if a new line is added at the end of the screen. One should be careful using this option, because it will take quite some time to scroll. All pixels above will be reloaded when a scroll occurs, resulting in a delay of about 2 seconds. It can be used during editing of a program, however not during printing of lots of data in a runtime situation.
<r> or +8: By default the overall resolution will be automatically set to the display. This can be 240x320, 320*240, 120x160 or 160x120 (see the move command, in which this is configured). If the <r> bit is set, the original resolution is used. So if no VGA monitor is connected a non-VGA resolution (304x216) is used and the display resolution is set to 304x216. If a VGA monitor is connected, the resolution will become 480x432. So it can always be possible that either on the monitor or on the LCD display part of the information will be off-screen. One can always check the actual resolution with the reserved Basic variables `MM.HRES` and `MM.VRES`.
<n> or +16 or +32: These bits are not used yet
<t> or +64: If this bit is set, the touch screen is deactivated. If the I2C port has already been enabled by either the `OLED` command or by the `I2CEN` command, it will stay enabled. Only touches of the screen are not detected anymore.
<b> or +128: This will actually disable the SPI bus for the display and it will automatically enable the SDcard again. This has always to be done whenever data should be read from or written to the SDcard. When it is disabled it is not possible to refresh the display, however the existing information will be unchanged. The touch screen commands will stay active during an inactive SPI bus, because it is communicating over the I2C bus. When ready reading or writing the SDcard, one has to enable the display again with a configuration command with the <b> bit cleared.

`options_ext` values 512 - 1023 are used for moving (in fact copying) the upper left portion of the video memory to the display local video memory. In fact the same action as `OLED options` does on the OLED display. However with the LCD display we have some extra features. As the display supports colors the move commands already use a maximum of 8 different colors, meaning RGB (Red Green Blue) either fully open or fully closed. RGB=111 means white, RGB=000 means black. Also offered are double width, double height pixels, which means characters twice the standard dimensions. Of course one can also choose a bigger `FONT`, which can also be added to this. The last extra option is writing only the foreground pixels. For the move commands we have 9 bits, which can be defined by <fldrgbRGB>.

<RGB> defines the foreground color. As we normally like a white color on a black background, it is decided to invert the foreground colors. In this way the options value 512 results in a white text on a black background. So if we would like a red foreground, we should not set RGB to xxxxxx100, but to xxxxxx011 (so 512+3). As the display is configured to use 16 bit color information (5 bits red, 6 bits green, 5 bits blue) it is decided to give the user the opportunity to use all colors. This can be done by using the Basic variable `LCD_COLORF`. If this variable is defined by given it a value like `LCD_COLORF=&HF800` (full red) the foreground color information given in the command is overruled. Please keep in mind that once given the variable a value, the RGB information in the `OLED` command will be always overruled. The only way to get rid of this variable is the `CLEAR` command.

<rgb> defines the background color. Default it is black (000). If we like a red background, we set it to xxx100xxx (so 512+32). As with `LCD_COLORF` for the foreground color we can use the Basic variable `LCD_COLORB` to set the background color.

<d> defines the double width, double height. So 1 pixel in the original video memory becomes 4 pixels on the LCD screen.

<l> turns the display to landscape instead of portrait. So the normal resolution of 240x320 becomes 320x240. If <d> is on these values are 120x160 and 160x120.

<f> means only foreground pixels are sent to the display. This speeds up the process, especially when we have only text. Of course we should start with an empty screen (background color), because otherwise we get a mess. One should consider if using the real time option (described later) is not a better solution in this situation.

Please keep in mind that starting the real time option does not give you the possibility to choose a double width/height pixel <d>, nor the landscape option <l>. If they are needed the move command should be executed first. Colors are not of interest, because they have to be entered in the real time situation again anyway, also with more depth of each color.

`options_ext` values 1024 - 32767 are all used for the touch screen. In fact the values are also organised in groups referring to the MSB:
1024 - 2047: used for polling the touch screen
2048 - 4095: used for setting an interrupt for automatic periodically polling
4096 - 8191: used for setting an offset for x and y values
8192 - 16383: used for setting a scaling for x and y values
16384 - 32767: used for setting a lineno. of a subroutine to be called in case of touch

Whenever the touch screen is enabled by the configuration, two Basic variables are automatically created: `TOUCHX` and `TOUCHY`. Optional the following Basic variables can be created by the user: `TOUCHX_OFFSET, TOUCHY_OFFSET, TOUCHX_SCALING, TOUCHY_SCALING` and `TOUCH_LINENO`. This can be done by `DIM variable` or by simply giving the variable a value. If a new value is given to the variable in Basic or with an `OLED` command (4096 - 32767), it will overrule the previous value.

The resolution of the touch screen is 12 bits, both in X and Y direction. This means that both `TOUCHX` and `TOUCHY` can have a value from 0 to 4095. In practice about 350 on all 4 sides are off-screen; so the real borders are 350 - 3750. This may vary for every display. The coordinate 350-350 is in the top left corner and 3750-3750 in the bottom right corner. If the display has been changed from portrait to landscape by the move command, the X and Y coordinates have been corrected accordingly.

The 350 - 350 coordinate in the top left corner can be corrected by the offset components. Both `TOUCHX_OFFSET=350:TOUCHY_OFFSET=350` and `OLED 4096+350:OLED 6144+350` will change the 350 - 350 coordinate to 0 - 0. So the X-offset can be corrected by `OLED 4096+x` and the Y-offset by `OLED 6144+y`. Minimum (also default) values for x and y are 0; maximum values 2047. If the value is greater than 350 an area from the left side and/or top side will result in a value of 0 for `TOUCHX` and/or `TOUCHY`.

For many applications it will be easier to have only a few coordinates on the screen instead of 4096x4096. This can be done by introducing a scaling factor. E.g. we want to separate the whole screen in 4x4 areas. We already measured that we can use 350 - 3750 for both x and y. By introducing the offset as above we changed this to 0 - 3400. Now we can further correct by `TOUCHX_SCALING=850:TOUCHY_SCALING=850` or `OLED 8192+850:OLED 12288+850`. This will result in: 0-850=>0; 850-1700=>1; 1700-2550=>2; 2550-3400=>3. So it is a division by the scaling factor to a fixed value. X-scaling can be done by `OLED 8192+x` and y-scaling by `12288+y`. Maximum values 4095 (which will always result in 0). Scaling factor 0 is automatically corrected to 1, preventing dividing by 0.

To start an interrupt subroutine if the screen is touched, one can use the Basic variable `TOUCH_LINENO=lineno` or `OLED 16384+lineno`. Using the `OLED` alternative the maximum number is 16387; the Basic variable can go up to 65000. The lineno should exist in the program otherwise an error will occur. The Basic subroutine has to be ended by the `IRETURN` statement.

Whenever the touch screen is activated the coordinates are stored in the local registers of the touch controller. As long as it does not get a request over the I2C bus from the master, these values remain in the registers and will not be updated by a new touch. As soon as the master has requested the values, they can be updated by a new touch. So the master always has to poll the registers to get the actual data. This can be done in two ways.

The manual polling is done by `OLED 1024+jitter time`. Every time this command is given the `TOUCHX` and `TOUCHY` are updated if a new touch has occurred. The `jitter time` can vary from 0 to 1023 ms. So the command can be `OLED 1024` up to `OLED 2047`. There will be no new check within this `jitter time`, if a touch has been detected.

The automatic polling is done based on the ms timer. `OLED 2048+period` will check every `period` in ms for new values of `TOUCHX` and `TOUCHY`. `period` can be 1 up to 2047 ms. If it is set to 0 (`OLED 2048`) the check for new values is stopped. As mentioned earlier it is not allowed to use this method if some other I2C device (e.g. an OLED display) is on the same I2C bus. This will crash the system.

`option_ext` values 32768 - 65535 will activate the real time modus of the display. The `option_ext` parameter has the syntax: <1rrggbbRRRGGGBBB>. So the MSB of the 16 bit value is 1; all the other bits are used to set the color, both foreground and background.

Before it is possible to set the display in real time control the configuration and move commands have to be given. E.g. the optional scroll, resolution fit and inversion are taken over from the configuration setting and orientation (portrait/landscape) and single/double pixel size from the move settings. The move color setting is overruled by the color setting in the options parameter in this real time command.

What happens if the real time command is activated? Writing to the Duinomite video memory is done pixel by pixel. Also writing to the LCD video memory is done pixel by pixel. By linking the pixel writing of the LCD video memory directly to the pixel writing of the Duinomite video memory, we fill the LCD screen simultaneously with the local Duinomite display. Of course we get extra delay in writing to the local video and the USB terminal by using the real time option. However for most applications this will not be a problem. Only a command like `CLS`, which clears directly the whole video memory takes now about 1 sec extra to clear also the LCD video memory. One can change foreground and background colors at any time in the program by simply given the real time command again with other color parameters. So every next character can have another fore- and background color. Also the graphic statements `CIRCLE, LINE` and `PIXEL` can be used in any color, including the filling of circle or box.

The different colors have been extended compared to the move command. We now use 3 bits per color (red, green, blue) for the foreground and 2 bits per color for the background. This results in 512 different foreground colors and 64 different background colors. As in the move command the basic variables `LCD_COLORF` and `LCD_COLORB` will overrule the color settings in the real time command. However keep in mind that just changing the variables `LCD_COLORF` and/or `LCD_COLORB` will not change the colors directly. A new real time command has to be given to activate the new color(s).

Example:

```
OLED &H100: 'initiate the display
OLED &H200+&H100: 'move to the LCD in landscape
OLED &H8000: 'activate rt mode; bg black; fg white
LCD_COLORF=&HF800: 'New foreground color is red
LCD_COLORB=&H001F: 'New background color is blue
OLED &H8000: 'New colors are activated.
```

2.83 ON

Purpose:
To branch to one of several specified line numbers, depending on the value returned when an expression is evaluated.

Syntax:
```
ON expression GOTO linenumbers
ON expression GOSUB linenumbers
```

Comments:

In the `ON ... GOTO` statement, the value of `expression` determines which line number in the list will be used for branching. For example, if the value is 3, the third line number in the list will be destination of the branch. If the value is a non-integer, the fractional portion is rounded.

In the `ON ... GOSUB` statement, each line number in the list must be the first line number of a subroutine.

If the value of `expression` is zero or greater than the number of items in the list (but less than or equal to 255), MM-BASIC continues with the next executable statement.

If the value of `expression` is negative, or greater than 255, an "`Number out of range`" error occurs.

Examples:

```
100 IF R<1 or R>4 then print "ERROR":END
```

If the integer value of R is less than 1, or greater than 4, program execution ends.

```
200 ON R GOTO 150,300,320,390
210 PRINT "NEXT"
```

If R=1, the program goes to line 150.
If R=2, the program branches to line 300 and continues from there.
If R=3, the branch will be to line 320.
If R=4, the branch will be to line 390.

2.84 OPEN

Purpose:
To establish input/output (I/O) to a file or allocate a buffer to support serial asynchronous communications with other computers and peripheral devices.

Syntax:
```
OPEN filename$ FOR mode AS [#]filenumber
OPEN "COM[n]:[speed[,buf[,int[,lvl]]]][,mode][,FC][,OC]" AS
[#]filenumber
OPEN "COM[n]:[speed[,buf[,int[,lvl]]]][,mode][,FC][,OC]" AS
CONSOLE
```

Comments:

`filename$` is the name of the file 8 characters max with extension .XXX max 3 characters.

`mode` is a string expression: `OUTPUT` Sequential output mode, will overwrite existing file with the same name. `INPUT` Sequential input mode. `APPEND` Sequential output mode, but from position at end of file. If there is no existing file the `APPEND` option will act the same as the `OUTPUT` mode (i.e. the file is created then opened for writing).

`filenumber` is a number between 1 and 10. The number associates an I/O buffer with a disk file or device. This association exists until a `CLOSE` or `CLOSE` file number statement is executed.

The `INPUT`, `LINE INPUT`, `PRINT`, `WRITE` and `CLOSE` commands as well as the `EOF()` and `INPUT$()` functions all use `filenumber` to identify the file being operated on. See also `OPTION ERROR` and `MM.ERRNO` for error handling.

More than one file can be opened for input at one time with different file numbers. For example, the following statements are allowed:
```
OPEN "TEMP.TXT" FOR INPUT AS #1
OPEN "TEMP.TXT" FOR INPUT AS #2
```

However, a file may be opened only once for output or appending. For example, the following statements are illegal:
```
OPEN "TEMP.TXT" FOR OUTPUT AS #1
OPEN "TEMP.TXT" FOR OUTPUT AS #2
```

Be sure to close all files before removing SD card (see `CLOSE`).

When a disk file is opened for `APPEND`, the position is initially at the end of the file. `PRINT` then extends the file.
If the file is opened as `INPUT`, attempts to write to the file result in "`Cannot find file`" errors.
If the file is opened as `OUTPUT`, attempts to read the file result in "`Cannot read from file`" errors.
Opening a file for `OUTPUT` or `APPEND` fails, if the file is already open in any mode.

COM[n] is a valid communications device: com1: com2: com3: com4:

The OPEN "COMx:" command has been edited in the CAN 2.4 version of the software. A lot of errors in the code and the documentation has been solved.

com1: RX is Arduino.D2 or GPIO.13;
TX is Arduino.D3 or GPIO.14;
RTS is Arduino.D4 or GPIO.15 (if FC is used);
CTS is Arduino.D5 or GPIO.16 (if FC is used);
com2: RX is Arduino.D6 or GPIO.17;
TX is Arduino.D7 or GPIO.18;
com3: RX is UEXT.4;
TX is UEXT.3;
com4: RX is RS232. Rx if R2 is mounted also Arduino.D0 or GPIO.11;
TX is RS232. Tx if R3 is mounted also Arduino.D1 or GPIO.12;
RTS and CTS are available on the Arduino Con1 on the Duinomite MEGA. Unfortunately the internal UART does not have external RTS and CTS pins. So if they have to be used (FC activated) it has to be done in the application software.

speed is a literal integer specifying the transmit/receive baud rate. The default is 9600 bps.

com1: and com2: are implemented with bit-bang by MM-BASIC and maximum speed is 19200 bps,com3: and com4: are real UARTs and maximum speed is 8 000 000 bps (speed over 115200 bps is not reliable in practice).

buf is the buffer size used for receive and transmit, the same value for both of them. The default value is 128 characters. The minimum value is 8. There is no maximum defined, however the buffer is taken from the Array/String memory. So a value of 10000 means 20k, or 57% of this memory. If buf has to be specified also baud must be specified.

int is the line number of the interrupt routine which is called whenever a number of characters, specified by lvl has been reached. If int has to be specified also baud and buf must be specified.

lvl is the level of the buffer (no. of characters) which has to be reached before the interrupt routine starts. The default value is 1. The maximum value of course buf. If lvl has to be specified also baud, buf and int must be specified.

OC will set the output of the TX and RTS on COM1: to Open Collector. This parameter is case sensitive and should always be the last parameter in the command string.

FC enables flow control on com1: or on com4: (see restriction above). This parameter is case sensitive and should be the last or the one before OC in the command string.

mode is new in CAN 2.4. It specifies an important lack in the original DMBasic specification. The serial communication specification has a variable length of data bits and stop bits. Also a parity can be defined. The original DMBasic always uses the most used mode: 8 databits, no parity, 1 stop bit. mode is now defined as a 3-digit number. The first digit specifies the number of data bits (5 -9), the second one the parity (0 = no parity; 1=odd; 2=even) and the last one the number of stopbits (1-2). So the default value of mode is 801. mode should always be the last parameter or the one directly before FC or OC.

If an illegal value is given for mode it will be ignored and the value will probably be interpreted as value for one of the parameters earlier in the string. An exception is the number of data bits in the COM3: and COM4: specification. These ports are linked to internal UART channels in the CPU and these UARTs only accept values 8 and 9 as the number of databits. So an error message is the result if a value of 5, 6 or 7 is given.

If parity in mode is set to odd or even, two Basic variables are created. The first one is COM_PARITY_REPLACE. The default value of this parameter is -1. This means the received character gets the wrong value in the buffer, so it is not replaced. If COM_PARITY_REPLACE has been changed in Basic (e.g. to 255), the character will get this value (0 -255). The second Basic variable is COM_PARITY_ERRORS. This is the counter of parity errors. It will be incremented by 1 at every parity error. In Basic it can be read and also be reset by COM_PARITY_ERRORS=0. It is also reset at every new OPEN command.

If the number of data bits is 9, parity is set to no. Two Basic variables are created. The first one COM_BIT9_VALUE has the value of 0 or 1 of the ninth bit. It had to be separate from the other 8 bits, because Basic cannot handle 9 bit characters. So if you want to transmit a value with the ninth bit 1, the following statements should be given: COM_BIT9_VALUE=1:PRINT CHR$(x); where x contains the ASCII value of the 8 lower bits. After the transmit the bit 9 value is automatically set to 0 again. Receiving is more difficult, because the 8 bits are placed in the buffer, however the COM_BIT9_VALUE can continuously vary between 0 and 1. If the second variable COM_BIT9_ADDRESS is on the default value of -1, it will work this way and only very slow communication will be possible, character by character. However the nine data bits mode is mostly used in RS485 communication and a message with the ninth bit set to 1 means the address of the active node is sent. If COM_BIT9_ADDRESS has a value of 0 - 255, this will be the address of this node. If it receives the message with the ninth bit set and its address, it will be set open for the further datastream. In this case the COM_BIT9_VALUE will get the value of 1 and will stay on that value as long as no other address has been given (unfortunately this only works on COM1: and COM2:). As soon as a new address message has been received with another address, it will stop and refuse the following messages and the COM_BIT9_VALUE will get the value of 0.

A communications device may be opened to only one filenumber at a time.
If you try to open com other than com1:- com4: "Invalid syntax" error will occur, if you try to open already opened filenumber "COMx is already open" error will occur.
When the port filenumber is opened the port can be written to and read from using any commands or functions that use a file number.

A serial port can be opened with "AS CONSOLE". In this case any data received will be treated the same as keystrokes received from the keyboard and any characters sent to the video output will also be transmitted via the serial port. This enables the remote control of MMBasic via a serial interface.

Examples:

In the following, COM1 is opened for communications with all defaults: speed at 9600 bps, no parity, eight databits, and one stop bit on port 1.

```
10 OPEN "COM1:" AS 1
```

In the following, COM1 is opened for communications at 2400 bps on port 2.

```
20 OPEN "COM1:2400" AS #2
```

In the following, COM1 is opened for asynchronous I/O at 1200 bits/second on port 1. 8 databits, odd parity and 2 stopbits are to be produced or checked.

```
10 OPEN "COM1:1200,812" AS #1
```

In the following all specifications are given. COM1 is opened at 9600 bits/second, the buffers have been extended to 256 bytes, an interrupt routine at line 1000 is called if the buffer is half full (128 bytes), the mode is: 8 databits, no parity, 1 stopbit, flowcontrol is enabled and the Tx and RTS outputs are set to open collector.

```
10 OPEN "COM1:9600,256,1000,128,801,FC,OC" AS #1
```

2.85 OPTION

Purpose:
To declare some behaviour of the system

Syntax:
```
OPTION BASE n
OPTION ERROR CONTINUE
OPTION ERROR ABORT
OPTION PROMPT prompt$
OPTION Fnn string$
OPTION VIDEO ON/OFF
OPTION USB ON/OFF/DISCONNECT
```

Comments:

In `OPTION BASE n` can `n` be 1 or 0. The default base is 0.

If the statement `OPTION BASE 1` is executed, the lowest value an array subscript can have is 1.

An array subscript may never have a negative value.

`OPTION BASE` gives an error only if you change the base value. This allows chained programs to have `OPTION BASE` statements as long as the value is not changed from the initial setting.

You must code the `OPTION BASE` statement before you can define or use any arrays. If an attempt is made to change the option base value after any arrays are in use, an error results.

The `OPTION ERROR CONTINUE` will cause MM-BASIC to ignore file related errors. The program must check the variable `MM.ERRNO` to determine if and what error has occurred.

The `OPTION ERROR ABORT` sets the normal behavior (ie, stop the program and print an error message). The default is `ABORT`.

`OPTION ERROR` only relates to errors reading or writing from the SD card, it does not affect the handling of syntax and other program errors.

In the following example if “123.TXT” exists on the SD card 0 will be printed. If “123.TXT” does not exist 6 will be printed (“Cannot find file”)

```
10 OPTION ERROR CONTINUE
20 OPEN "123.TXT" FOR INPUT AS #1
30 PRINT MM.ERRNO
```

`prompt$` in `OPTION PROMPT prompt$` can be an expression which will be evaluated when the prompt is printed.

Maximum length of the prompt string is 48 characters. The prompt is reset to the default ("> ") on power up but you can automatically set it by saving the following example program as "AUTORUN.BAS" on the internal flash drive A:

```
10 OPTION PROMPT "MY PROMPT: "
20 NEW
```

Added in CAN version:

OPTION VIDEO OFF: This disables the video output; Arduino D8 and D9 are available as standard I/O pins, logical PIN(19) and PIN(20). Last one is still connected to the yellow LED onboard.

OPTION VIDEO ON: This enables the video output again.

OPTION USB OFF: Disables the USB output

OPTION USB ON: Enables the USB output

OPTION USB DISCONNECT: Disables USB completely (also input); ON enables again.

Examples:
Next example changes the prompt as in APPLE]:

```
OPTION PROMPT "]"
```

This is also valid prompt:

```
OPTION PROMPT TIME$+" : "
```

or

```
OPTION PROMPT CWD$+" : "
```

`Fnn` in `OPTION Fnn string$` is the function key `F1` to `F12`. Maximum string length is 12 characters.

`string$` can also be an expression which will be evaluated at the time of running the `OPTION` command. This is most often used to append the ENTER key (chr$(13)), or double quotes (chr$(34)).

Normally these commands should be included in an "AUTORUN.BAS" file (see `OPTION PROMPT` for an example).
Examples:

```
OPTION F1 "RUN" + CHR$(13)
OPTION F2 "SAVE " +CHR$(34)
```

2.86 PAUSE

Purpose:
Will halt execution of the running program for `number` milliseconds.

Syntax:
`PAUSE number`

Comments:

The maximum value of `number` is 4294967295 (about 49 days).

Examples:
Next code blink LED which anode (+) is connected to Arduino.A0 and cathode (-) is connected to GND.

```
10 SETPIN 1,8'set A0 as Digital output
20 DO 'do loop
30 PIN(1) = 1 'switch ON the LED
40 PAUSE 500 'wait ½ second
50 PIN(1) = 0 'switch OFF the LED
60 PAUSE 500 'wait ½ second
70 LOOP 'loop forever
```

2.87 PIN

Purpose:
To set an output to a value.

Syntax:
`PIN(n) = value`

Comments:

For a `PIN()` configured as digital output this will set the output to low if `value` is zero or high if `value` is non zero. You can set an output high or low before it is configured as an output and that setting will be the default output when the `SETPIN` command takes effect.

See the function `PIN()` for reading from a pin and the command `SETPIN` for configuring it.

Extensions in the CAN version of DMBASIC:

The PIN reference is extended by a range. To use this `n` has to be 10000 up. The format is: `1xxyy`, where xx is the first PIN in the range and yy the last one. The command `PIN(1xxyy)=z` can be used now.

If `y` in `SETPIN x,y` is set to 18, 19, 28 or 29, `value` is interpreted as value for PWM. See also description of `PWM`. `PIN(1xxyy)=z` is not supported in this case.

Examples:

Next code blink LED which anode (+) is connected to Arduino.A0 and cathode (-) is connected to GND.

```
10 SETPIN 1,8 'set A0 as Digital output
20 DO 'do loop
30 PIN(1) = 1 'switch ON the LED
40 PAUSE 500 'wait ½ second
50 PIN(1) = 0 'switch OFF the LED
60 PAUSE 500 'wait ½ second
70 LOOP 'loop forever
```

2.88 PIXEL

Purpose:
To display a point at a specified place on the screen.

Syntax:
`PIXEL (x,y) = value`

Comments:

`(x,y)` represents the coordinates of the point.

`value` if zero the point is with black color otherwise with white.

See also function `PIXEL(x,y)` which returns the value of a pixel with coordinates `(x,y)`.

Coordinate values can be beyond the edge of the screen.

(0,0) is always the upper-left corner and (`MM.HRES, MM.VRES`) is the lower-right corner.

Example 1:
The following draws a diagonal line from (0,0) to (100,100).

```
10 CLS
20 FOR I=0 TO 100
30 PIXEL(I,I)=1
40 NEXT
```

Example 2:
The following clears out the line by setting each pixel to 0.

```
50 FOR I=100 TO 0 STEP -1
60 PIXEL(I,I) = 0
70 NEXT I
```

2.89 POKE

Purpose:
To write (poke) a byte of data into a memory location.

Syntax:
`POKE hiword,loword,byte`

Comments:

`hiword` is the upper 16 bit of the memory address.

`loword` is the low 16 bit of the memory address.

`Byte` must be within the range of 0 to 255.

The PIC32 maps all control registers, flash (program) memory and volatile (RAM) memory into a single address space. The PIC32MX7XX Family Data Sheet lists the details of this address space while the source code will provide the symbolic names used in the firmware and the .map file (produced after a successful compile) will list the addresses of these symbols. These addresses will change with each version of the firmware so programs should use the predefined variable `MM.VER` to determine the currently running version.

This command is for expert users only.

If you use this facility to access an invalid memory address the MIPS CPU will throw an exception which causes the processor to reset and clear all memory. To see this effect try `POKE 0,0,0`.

The complementary function to `POKE` is `PEEK`. The argument to `PEEK` is an address from which a byte is to be read.

`POKE` and `PEEK` are useful for efficient data storage, loading assembly language subroutines, and for passing arguments and results to and from assembly language subroutines.

2.90 PRESET

Purpose:
To clear a point at a specified place on the screen. Please for new code do use `PIXEL(x,0)`.

Syntax:
`PRESET(x,y)`

Comments:

`(x,y)` represents the coordinates of the point.

Coordinate values can be beyond the edge of the screen.
(0,0) is always the upper-left corner and (MM.HRES, MM.VRES) is the lower-right corner.

Example:
The following clears out the line by setting each pixel to 0.

```
50 FOR I=100 TO 0 STEP -1
60 PRESET(I,I)
70 NEXT I
```

2.91 PRINT

Purpose:
To output a text to the screen, file or com port.

Syntax:
```
PRINT [#filenumber, ][list of expressions][;,]
? [#filenumber, ][list of expressions][;,]
```

Comments:

If `list of expressions` is omitted, a blank line is displayed.

If `list of expressions` is included, the values of the expressions are displayed. Expressions in the list may be numeric and/or string expressions, separated by commas, spaces, or semicolons.

Stringconstants in the list must be enclosed in double quotation marks.

A semicolon (;) at the end of the expression list will suppress the automatic output of a carriage return/ newline at the end of a print statement.

A question mark (`?`) may be used in place of the word `PRINT`.

When numbers are printed on the screen, the numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus (−) sign. Single-precision numbers are represented with seven or fewer digits in a fixed-point or integer format.

Integers (whole numbers) are printed without a decimal point while fractions are printed with the decimal point and the significant decimal digits. Large numbers (greater than six digits) are printed in scientific format.

The function `FORMAT$()` can be used to format numbers. The function `TAB()` can be used to space to a certain column and the string functions can be used to justify or otherwise format strings.

`filenumber` is the number used when the file was opened for `OUTPUT` or `APPEND`.

In the CAN version of DMBasic the functionality is not changed. However a bug in version 2.7 has been corrected. PRINT, but also WRITE, INPUT, LINE INPUT, MEMORY and LIST can write larger amounts of data to the USB buffer. If these commands write data and the interrupt of USB comes to transfer the data to the terminal port, sometimes data is written two times. In the CAN version we synchronized the buffer input and output. This was also necessary for all CAN commands which transfer data to the USB port.

Examples:

```
10 X$= "---"
20 PRINT X$"MONTHLY REPORT" X$
RUN
---MONTHLY REPORT---
```

The ',' will space up to 10 characters.

```
10 PRINT 1,2,3,4,5,6,7,8,9,10
RUN
1    2    3    4    5    6    7    8    9    10
```

```
10 PRINT #1, A
```

0 is the result in the file, because A did not get a value yet.

```
10 A=26
20 PRINT#1, A
```

26 is the result in the file

```
10 A=26
20 PRINT#1, "A"
```

A is the result in the file, because A is interpreted as string.

If double quotation marks are required within a string, use `CHR$(34)` (the ASCII character for double quotation marks).

```
100 PRINT #1,"He said,"Hello", I think"
```

Result: He said, 0, I think, because the machine assigns the value 0 to the variable "Hello."

```
100 PRINT #1, "He said, "CHR$(34) "Hello,"CHR$(34) " I
think."
```

Result: He said, "Hello," I think

```
10 A$="12345": B$="67890"
20 PRINT#1, A$, B$
```

gives a file image of: 12345 67890

```
30 PRINT#1, A$; B$
```

gives a file image of: 1234567890

2.92 PSET

Purpose:
To display a point at a specified place on the screen. Please for new code do use `PIXEL(x,1)`.

Syntax:
`PSET(x,y)`

Comments:

`(x,y)` represents the coordinates of the point.

Coordinate values can be beyond the edge of the screen.
(0,0) is always the upper-left corner and (MM.HRES, MM.VRES) is the lower-right corner.

Example:

The following draws a diagonal line from (0,0) to (100,100).

```
10 CLS
20 FOR I=0 TO 100
30 PSET(I,I)
40 NEXT
```

2.93 PWM

Purpose:
To generate a flexible PWM output on a digital output

Syntax:
PWM [<pin>[,<period 1>[,<period 0>]]]

Comments:

PWM converts the digital output pins to PWM outputs. A maximum of 8 PWM outputs can be created this way. The clock which is used for the PWM outputs is based on the 1 ms interrupt of the Basic language. This is a relative low frequency, however if a higher frequency would be used the overall system performance would be decreased. Also the CANOBJECT messages and the CANIOLINKs are based on this interrupt, so using both PWM and CANOBJECT can influence the real-time behaviour of the system.

To actually use the PWM command on one or more pins, the used pins should be set to digital output with either SETPIN x,8 (standard) or SETPIN x,9 (open collector).

PWM without any parameters or PWM 0, y,z clears all PWM settings. PWM x,0,0 clears PWM setting of PIN x. This setting is free for any other PIN now.

PWM x,y,z is the normal use of the command. PIN x will be set to 1 during y msec and after that to 0 during z msec. The range of x is 1 - 20 (20 being the on-board yellow LED). The range of y and z is 0 - 4095. Using both y and z gives the option to choose between high speed inaccurate and low speed accurate.

PWM x,y or PWM x,y,0 are pre-defined options for PWM x,y,z. It will use y as a percentage value of full scale. E.g. PWM x,10 is the same as PWM x,10,90 and PWM x,70 as PWM x,70,30. Any value of y>100 will result in y=100

In the situation above PWM x,50 will set the output high during 50 ms, after that low during 50 ms, then again high during 50 ms, etc. In many situations we prefer a possible switching every 1 ms. We would like PWM x,50 to result in 10101010….
The signal will be following small changes more smoothly and the low pass filter at the output can be set to a higher frequency. To achieve accuracies of 1 promille we need a period of 1 second instead of 100 ms. This should also be possible using the percentage. To use these options above it is decided to give the z-parameter two special values. PWM x,y,10000 is in principle the same as PWM x,y, however by placing the 1's and 0's smoothly over the 100 ms period. PWM x,y,100000 will do the same however over the 1 second period. So both PWM x,50,10000 and PWM x,500,100000 will result in the above 10101010…. as described above.
It has been decided to use these options also directly within the SETPIN statement.
SETPIN x,18 will use PIN(x)=y as PWM x,y,10000
SETPIN x,19 does the same with OC output
SETPIN x,28 will use PIN(x)=y as PWM x,y,100000
SETPIN x,29 does the same with OC output

<CTRL-C> will stop all PWMs and set the outputs to 0.

2.94 RANDOMIZE

Purpose:
To reseed the random number generator.

Syntax:
```
RANDOMIZE [expression]
RANDOMIZE TIMER
```

Comments:

If expression is omitted, MM-BASIC will generate error "`Invalid syntax`"

If the random number generator is not reseeded, the `RND()` function returns the same sequence of random numbers each time the program is run.

To change the sequence of random numbers every time the program is run, place a `RANDOMIZE` statement at the beginning of the program, and change the argument with each run. One good way to do this is use the TIMER function.

```
RANDOMIZE TIMER
```

Example:
```
10 RANDOMIZE TIMER
20 FOR I=1 to 5
30 PRINT RND(1)*100;
40 NEXT I
```

2.95 READ

Purpose:
To read values from a DATA statement and assign them to variables.

Syntax:
READ list of variables

Comments:

A READ statement must always be used with a DATA statement.

READ statements assign variables to DATA statement values on a one-to-one basis.

READ statement variables may be numeric or string. If DATA value is string but you attempt to read number "Expected a number" error results. The opposite is acceptable i.e. DATA value to be number but to read it in string variable.

A single READ statement may access one or more DATA statements. They are accessed in order.

Several READ statements may access the same DATA statement.

If the number of variables in list of variables exceeds the number of elements in the DATA statement(s), an "No more DATA to read" error occurs.

If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA statements from the start, use the RESTORE statement.

Examples:

```
10 DATA 1, 2, 3, "A", "B", "C"
20 READ V1, V2, V3, S1$, S2$, S3$
30 PRINT V1, V2, V3
40 PRINT S1$, S2$, S3$
RUN
 1 2 3
A B C

10 DATA 11, 22, 33
20 READ A$, B$, C$ 'strings are read with numbers as DATA
30 PRINT A$; B$; C$
RUN
112233
```

2.96 REM

Purpose:
To allow explanatory remarks to be inserted in a program.

Syntax:
```
REM[comment]
'[comment]
```

Comments:

`REM` statements are not executed, but are output exactly as entered when the program is listed.

Once a `REM` or its abbreviation, an apostrophe (`'`), is encountered, the program ignores everything else until the next line number or program end is encountered.

`REM` statements may be branched into from a `GOTO` or `GOSUB` statement, and execution continues with the first executable statement after the `REM` statement. However, the program runs faster if the branch is made to the first statement.

Remarks may be added to the end of a line by preceding the remark with an apostrophe (`'`) instead of `REM`.

Examples:

```
120 REM CALCULATE AVERAGE VELOCITY
130 FOR I=1 TO 20
140 SUM=SUM+V(I)
150 NEXT I
```

or

```
120 FOR I=1 TO 20 'CALCULATED AVERAGE VELOCITY
130 SUM=SUM+V(I)
140 NEXT I
```

2.97 RENUMBER

Purpose:
To renumber program lines currently held in memory including all references to line numbers in commands such as ELSE, GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB.

Syntax:
```
RENUMBER [first],[increment][,start]]
```

Comments:

`first` is the first line number to be used in the new sequence. The default is 10.

`increment` is the increment for each line. The default is 10.

`start` is the line number in the old program where renumbering should commence from. The default is the first line of the program.

This command will first check for errors that may disrupt the renumbering process and it will only change the program in memory if no errors are found. However, it is prudent to save the program before running this command in case there are some errors that are not caught.

If a nonexistent line number appears after one of these statements ELSE, GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB, the error message "`Line number does not exist. Cancelling RENUMBER`" appears.

`RENUMBER` cannot be used to change the order of program lines or to create line numbers greater than 65000.

Examples:

```
RENUMBER
```
Renumbers the entire program. The first new line number will be 10. Lines increment by 10.

```
RENUMBER 1000,10
```
Renumbers the entire program. The first new line number will be 1000. Lines increment by 10.

```
RENUMBER 1000,10,200
```
Renumbers the lines from 200 up so they start with line number 1000 and are incremented by 10.

2.98 RESTORE

Purpose:
To allow DATA statements to be reread from the start.

Syntax:
RESTORE

Comments:

After RESTORE the next READ statement accesses the first item in the first DATA statement.

Example:

```
10 READ A, B, C
20 RESTORE
30 READ D, E, F
40 DATA 57, 68, 79
```

Assigns the value 57 to both A and D variables, 68 to B and E, and so on.

2.99 RETURN

Purpose:
To return to the calling position after execution of a subroutine

Syntax:
```
RETURN
```

Comments:

See `GOSUB` for details.

2.100 RMDIR

Purpose:
To delete a subdirectory.

Syntax:
`RMDIR pathname$`

Comments:

`pathname$` is a string expression, identifying the subdirectory to be removed from its parent.

The subdirectory to be deleted must be empty of all files except "`.`" and "`..`" or a "`Directory not empty`" error occurs.

Example:

```
RMDIR "SALES"
```

2.101 RUN

Purpose:
To execute the program currently in memory, or to load a file from the SD card into memory and run it.

Syntax:
```
RUN [linenumber]
RUN filename$
```

Comments:

`RUN` or `RUN linenumber` runs the program currently in memory.

If `linenumber` is specified, execution begins on that line. Otherwise, execution begins at the lower line number.

If there is no program in memory then `RUN` will do nothing.

`RUN filename$` closes all open files and deletes the current memory contents before loading the specified file from SD card into memory and executing it.

If an extension is not specified ".BAS" will be added to the file name.

Examples:

```
RUN "HELLO"
```

Runs HELLO.BAS .

2.102 SAVE

Purpose:
To save a program file on the SD card or the local A drive.

Syntax:
`SAVE filename$`
`SAVE # filename$`

Comments:

`filename$` is a quoted string that follows the normal MS-DOS naming conventions i.e. 8 characters name and 3 characters extension. If `filename$` already exists, the file will be written over. If the extension is omitted, .bas will be used.

The `SAVE # filename$` command has been added in the CAN version. Using this syntax all statements will be saved with tokens (single ASCII characters 128 -255) for commands and functions. This will result in smaller files (about 20%). The `LOAD # filename$` option must be used to load the program in memory in this case.

Examples:

The following commands save the file TEST.BAS in the two different structures:

```
SAVE "TEST"
SAVE # "TEST"
```

2.103 SAVEBMP

Purpose:
To save the current VGA or composite screen as a BMP file in the current working directory on the SDcard.

Syntax:
`SAVEBMP filename$`

Comments:

`filename$` is a quoted string that follows the normal MS-DOS naming conventions i.e. 8 characters name and 3 characters extension. If `filename$` already exists, the file will be written over. If the extension is omitted, .BMP will be added.

Note that Windows 7 Paint has trouble displaying the image. This appears to be a bug in Paint as all other software tested (including Windows XP Paint) can display the image without fault. To save a program file the SD card.

Examples:

The following command saves the file IMAGE.BMP :

```
SAVEBMP "IMAGE"
```

2.104 SDDISABLE

Purpose:
To switch off the SD-CARD from the SPI interface

Syntax:
`SDDISABLE`

Comments:

`SDDISABLE` should be used if the SPI interface is used for other purposes. The SPI data lines are on the same pins as the data lines of the SD interface. Therefore the Chip Select (CS) line of the SD card should be switched off (logical 1). This is done by `SDDISABLE`.

Command existed in original DMBasic 2.7, but was not documented.

2.105 SDENABLE

Purpose:
To switch on the SD-CARD again if it is switched off.

Syntax:
`SDENABLE`

Comments:

`SDENABLE` can be used to activate the SD-CARD again, after it has been switched off by `SDDISABLE.`

As long as the SPI interface is not used for other purposes, this command has no meaning. The SD-CARD is activated by default.

Command existed in original DMBasic 2.7, but was not documented.

2.106 SDFORMAT

Purpose:
To format a new or used SD-CARD

Syntax:
`SDFORMAT`

Comments:

The SD-CARD will be formatted in the existing FAT format.

Don’t format an unformatted disk, because this will format it in the smallest FAT format, which will give you about 50 MB of disk space.

All existing information on the SD-CARD is destroyed.

Disk Name will be DMBasic.

Command existed in original DMBasic 2.7, but was not documented.

2.107 SETPIN

Purpose:
To configure external IO port.

Syntax:
```
SETPIN pin-number, config
SETPIN pin-number, config, line-number
```

Comments:

`pin-number` is the number of the GPIO port. It's in range 1..20 for DuinoMite-Mega, DuinoMite-Mini and DuinoMite and PIC32T795.

The PIN reference is extended by a range. To use this `pin-number` has to be 10000 up. The format is: `1xxyy`, where xx is the first PIN in the range and yy the last one. The command `SETPIN(1xxyy),config` can be used now.

`config` defines how the port is configured for use.

0 Not configured or inactive
1 Analog input(reads input voltage 0-3.3V, accuracy better than ±1%)
2 Digital input(read 0 if input voltage is 0..0.65V, 1 if voltage is 2.5..3.3V/5.5V)(5V tolerant ports are: ....)(All digital inputs are Schmitt Trigger buffered.)
3 Frequency input(measure frequency up to 200KHz)
4 Period input(measure period pulse width 10nS or more)
5 Counting input(counts pulses up to 200KHz pulse width 10nS or more)
6 Interrupt on low to high input change(configure as digital input and interrupt is generator on level change low to high)
7 Interrupt on high to low input change(configure as digital input and interrupt is generator on level change high to low)
8 Digital output(output 3.3V when 1 and 0V when 0)(Typical current draw or sink ability on any I/O pin: 10mA)(Maximum current draw or sink on any I/O pin: 25mA)(Maximum current draw or sink for all I/O pins combined: 150mA)
9 Open collector digital output to 5V(0V when 0, Vcc when 1)(Maximum open collector voltage (I/O pins 11 to 20): 5.5V)

Added in the CAN version are the PWM output configurations:

18 PWM output 0 - 3.3 V; accuracy 1%
19 Open collector PWM output 0 - Vcc; accuracy 1%
28 PWM output 0 - 3.3 V; accuracy 0.1%
29 Open collector PWM output 0 - Vcc; accuracy 0.1%

`Line-number` could be add only if `config` is 6 or 7 and configure the GPIO port to generate interrupt on level change.

`line-number` is the start of the interrupt routine. This mode also configures the GPIO port as a digital input so the value of the port can always be retrieved using the function `PIN()`. See also `IRETURN` to return from the interrupt.

See the function `PIN()` for reading inputs and the command `PIN()=` for outputs.

Examples:

```
10 SETPIN 1,8 'make GPIO.1 / ARDUINO.A0 as OUTPUT
20 PIN(1) = 1' output 3.3V to the port
30 PIN(1) = 0' output 0V to the port

10 SETPIN 11118,2 'PINs 11 - 18 set as digital input
```

2.108 SETTICK

Purpose:
To setup periodic interrupt and forward the execution to interrupt routing.

Syntax:
`SETTICK period, line-number`

Comments:

`period` is the time in milliseconds between interrupts. The period can range from 1 to 4294967295mSec (about 49 days).

`line-number` is the line number of the interrupt routine. See also `IRETURN` to return from the interrupt. This interrupt can be disabled by setting `line-number` to zero (i.e., `SETTICK 0,0`).

Examples:
In the following example, LED connected to GPIO.1 (ARDUINO.A0) will start blinking when “1” is pressed and will stop when “2“ is pressed.

```
10 SETPIN 1,8 'ARDUINO.A0 is output
20 DO 'loop forever
30 C$=INKEY$: IF C$="" THEN 30 'wait key pressed
40 IF C$="1" THEN SETTICK 100,100 'set interrupt every 0.1
sec
50 IF C$="2" THEN SETTICK 0,0: PIN(1)=0 'clear the interrupts
60 LOOP
100 IF I=0 THEN I=1: PIN(1)=1: IRETURN 'toggle LED
110 IF I=1 THEN I=0: PIN(1)=0: IRETURN
```

2.109 SETUP

Purpose:
Change some default settings at start up or reset.

Syntax:
`SETUP`

Comments:

Command shows a menu with settings which can be changed by single key strokes.

When ready you can choose either to ignore (Q) the changes or change them in flash (X)

After the exit with changes, you have to reset or reboot to make them effective.

Command existed in original DMBasic 2.7, but was not documented.

2.110 SLEEP

Purpose:
Set the module in a low power standby mode

Syntax:
`SLEEP [time]`

Comments:

Command existed in original DMBasic 2.7, but was not documented.

`SLEEP` sets the PIC32 processor into a sleep mode, which consumes lower power. Of course the running program is frozen at the moment the command is activated.

The system can be awakened in several ways. The original way is after some `time` delay. This parameter must be entered as a text string. The following strings are allowed: SECOND, 10SECOND, MINUTE, 10MINUTE, HOUR, DAY, WEEK, MONTH and YEAR.

In the original DMBasic 2.7 already 2 other ways could be used to wake up:

- Pressing any key on the PS2 keyboard.
- Pressing the button (PIN 0).

In the CAN version are added:

- *The PINS 5,6 and 7.*
- *One of the COM ports*
- *One of the CAN ports.*

Multiple wake up events can be used in parallel. The first one to come, will wake up the system.

The configuration can be done by using a Basic parameter. This can be 0 (or simply not defined) if the wake up utility is not used or 1 (or any other positive value) when it is enabled.

The CAN wake up utility has to be done by setting the `CANINT` command.

The table on the next page shows the different wake up possibilities. `MM.SLEEP` has the value which represents the actual wake up source.

Wake up source	Basic variable	Parameter	Default value	MM.SLEEP value
Time elapsed	none	<time>	none	1
PIN 0 (button)	WAKEUP_PIN0		1	4
PIN 5	WAKEUP_PIN5		0	5
PIN 6	WAKEUP_PIN6		0	3
PIN 7	WAKEUP_PIN7		0	2
keyboard	WAKEUP_KB		1	8
COM port	WAKEUP_COM		0	16
CAN 1	none	CANINT	none	6
CAN 2	none	CANINT	none	7

2.111 SOFTRESET

Purpose:
Reset the system by command

Syntax:
SOFTRESET

Comments:

SOFTRESET performs a standard reboot of the system. In the standard DMBASIC 2.7, these kind of resets did not start the AUTORUN.BAS program if it was available either on DRIVE A or DRIVE B. A softreset was mostly caused by an illegal PEEK or POKE statement.

In the CAN version this has been changed. At every start or restart the AUTORUN.BAS is executed if available.

In the bootup message now an extra text is included: Status: 0x0yyy, in which yyy represents three extra hexadecimal numbers. The value of status can also be asked for in Basic by MM.BOOTUP, e.g. PRINT MM.BOOTUP. This can be used in the AUTORUN.BAS to execute different actions depending on the way the system is booted.

The status has a range of 0x000 to 0x3FF (0 - 1023), so 10 bits which can be represented by <CVEsNWSIBP>. Table below shows the meaning.

Configuration Mismatch	Illegal configuration set		
Voltage Regulator Standby	Voltage during sleep		
External Reset	Reset pin on chip		On at Reset switch
software Reset	Softreset		Also on at illegal
Not used			Peek or Poke
Watchdog Time out	Reset caused by Watchdog timer		If Watchdog active
Sleep Flag	Was in sleep state at reset		After Sleep
Idle Flag	Was in idle state at reset		Also set after sleep
Brown out flag	Voltage to low		Also at Power up
Power on flag	Cold start		At power up

2.112 SOUND

Purpose:
To generate single tone sound.

Syntax:`SOUND freq, duration, duty`

Comments:

`freq` is the tone frequency in Hertz (cycles per second). `freq` is a numeric expression within the range of 20 and 1,000,000, which corresponds to frequencies from 20Hz to 1Mhz for `duration` of milliseconds. The sound is played in the background and does not stop program execution.

If `duration` is zero, any active `SOUND` statement is turned off. If no `SOUND` statement is running, a `duration` of zero has no effect.

`duty` is the optional duty cycle of the waveform in percent. If it is close to zero the output will be a narrow positive pulse, if 50 a square wave will be generated and if close to 100 it will be a very wide positive pulse. If it is not specified the `duty` cycle will default to 50%.

Setting the duty cycle allows the sound output to be used as a Pulse Width Modulation (PWM) output for driving analogue circuits. The signal will be available at the sound connector and the voltage divider on this output should be removed so that the full signal level is available. The frequency of the output is locked to the PIC32 crystal and is very accurate and for frequencies below 100KHz the duty cycle will be accurate to 0.1%.

Examples:

The following example creates random sounds of short duration:

```
10 SOUND RND(1)*1000+100, 10
20 GOTO 10
```

2.113 TIME$

Purpose:
To set the current time.

Syntax:
`TIME$ = string exp`

Comments:

`string exp` is a valid string literal or variable that lets you set hours (`hh`), hours and minutes(`hh:mm`), or hours, minutes, and seconds (`hh:mm:ss`).

`hh` sets the hour (0-23). Minutes and seconds default to 00.

`hh:mm` sets the hour and minutes (0-59). Seconds default to 00.

`hh:mm:ss` sets the hour, minutes, and seconds (0-59).

If `string exp` is not a valid string "`Invalid Syntax`" error will occur.

As you enter any of the above values, you may omit the leading zero, if any. You must, however, enter at least one digit. If you wanted to set the time as a half hour after midnight, you could enter `TIME$="0:30"`, but not `TIME$= ":30"`.

If any of the values are out of range, an "`Invalid time`" error is issued. The previous time is retained.The current time is stored if `TIME$` is the target of a string assignment.

The time is set to "00:00:00" at power up.

See also the `TIME$` function for reading the time.

Examples:

The following example sets the time at 8:00 o'clock and reads it after 5 seconds :

```
TIME$ = "08:00"
PRINT TIME$
08:00:05
```

2.114 TIMER

Purpose:
To set the timer to a number of milliseconds.

Syntax:
```
TIMER = value
```

Comments:
`TIMER` sets the `TIMER` to number of milliseconds. Normally this is used to reset the timer to zero, but you can set it to any positive integer value.

`T`he timer is reset to zero on power up and reset.

2.115 TROFF

Purpose:
To switch off the trace option.

Syntax:
`TROFF`

Comments:

See `TRON`

2.116 TRON

Purpose:
To trace the execution of program statements.

Syntax:
`TRON`

Comments:

As an aid in debugging, the `TRON` (trace on) command enables a trace flag that prints each line number of the program as it is executed. The numbers appear enclosed in square brackets.

`TRON` may be executed in either the direct or indirect mode.The trace flag is disabled with the `TROFF` (trace off) command, or when a `NEW` command is executed.

Examples:

```
TRON
10 K=10
20 FOR J=1 TO 2
30 L=K + 10
40 PRINT J; K; L
50 K=K+10
60 NEXT
70 END
RUN
 [10][20][30][40] 1 10 20
 [50][60][30][40] 2 20 30
 [50][60][70]
TROFF
```

2.117 WATCHDOG

Purpose:
To activate and stop the internal PIC32 watchdog

Syntax:
`WATCHDOG [timer]`

Comments:

If active the watchdog is kicked by the video interrupt routine, which is called at every horizontal sync. By default it is set by just entering `WATCHDOG` and it is activated at every sync interrupt.

The watchdog is stopped by entering for `timer` the value 0.

Entering any other positive number will start a counter in the video routine and if the counter reaches the value of `timer` the watchdog is kicked.

For using the watchdog to reset the system if anything goes wrong, we advise to use the default value.

It is possible for time critical situations that it has an additional value if also the watchdog is set to a critical value. As this value is dependent on the used hardware, one has to experiment with this value. If the value is set too high the system is directly reset.

2.118 WEND

Purpose:
To branch back to the beginning of the `WHILE` loop

Syntax:
`WEND`

Comments:

Please for new code use `DO ... LOOP` statements.

See `WHILE` for details.

2.119 WHILE

Purpose:
To execute a series of statements in a loop as long as a given condition is true.

Syntax:
```
WHILE expression
 [loop statements]
WEND
```

Comments:

Please for new code use `DO ... LOOP` statements.

If `expression` is non-zero (true), `loop statements` are executed until the `WEND` statement is encountered. MM-BASIC then returns to the `WHILE` statement and checks expression. If it is still true, the process is repeated.

If it is not true, execution resumes with the statement following the `WEND` statement.

`WHILE` and `WEND` loops may be nested to any level. Each `WEND` matches the most recent `WHILE`.

An unmatched `WHILE` statement causes a "`WHILE without matching WEND`" error. An unmatched `WEND` statement causes a "`LOOP without matching DO`" error.

Examples:
```
10 SETPIN 1,8
20 WHILE PIN(0)=0 'blink LED on ARDUINO.A0 until USER button
is pressed
30 PIN(1) = 1: PAUSE 100: PIN(1) = 0: PAUSE 100
40 WEND
```

2.120 WRITE

Purpose:
To write data to a sequential file.

Syntax:
```
WRITE [#filenum,] list-of-expressions
```

Comments:

Please use `PRINT` for new code.

Outputs the value of each `expression` separated by commas (,).

`filenum` is the number under which the file was opened for output or append, if missing writes to screen.

`List-of-expressions` is a list of string and/or numeric expressions separated by commas or semicolons.

The `WRITE` and `PRINT` statements differ in that `WRITE` inserts commas between the items as they are written and delimits strings with quotation marks, making explicit delimiters in the list unnecessary.

Another difference is that `WRITE` does not put a blank in front of a positive number. After the last item in the list is written, a carriage return/line feed sequence is inserted.

If the `expression` is a number it is outputted without preceding or trailing spaces. If it is a string it is surrounded by double quotes (“). The list is terminated with a new line.

Examples:

```
WRITE 1, 2, 3, 4, 5, “HELLO”
1,2,3,4,5,”HELLO”
```

As you can see `WRITE` removes the spaces, add commas between the expressions and keeps the “”

Let A$ = "CAMERA" and B$ = "93604-1". The following statement:

```
WRITE #1, A$, B$
```

writes the following image to disk:

```
"CAMERA", "93604-1"
```

A subsequent `INPUT$` statement, such as the following, would input "CAMERA" to A$ and "93604-1"to B$:

```
INPUT #1, A$, B$
```

2.121 XMODEM

Purpose:
To transfer a file to or from a remote computer using the XModem protocol.

Syntax:
```
XMODEM SEND file$
XMODEM RECEIVE file$
```

Comments:

Transfers a file to or from a remote computer using the XModem protocol. The transfer is done over the USB connection or, if a serial port is opened as console, over that serial port.

`file$` is the file (on the SD card or internal flash) to be sent or received.The XModem protocol requires a cooperating software program running on the remote computer and connected to its serial port. It has been tested on Teraterm running on Windows and it is recommended that this be used. After running the XMODEM command in MM-BASIC select:

File -> Transfer -> XMODEM -> Receive/Send

from the Teraterm menu to start the transfer.

The transfer can take up to 15 seconds to start and if the XMODEM command fails to establish communications it will return to the MMBasic prompt after 30 seconds.

Download Teraterm from http://ttssh2.sourceforge.jp/
For Linux we recommend to use MINICOM.

Examples:

```
XMODEM RECEIVE "FILE.TXT"
XMODEM SEND "DATA.BAS"
```

3. The DMBasic functions

The first 14 functions are all one or two characters operators. They are described briefly with number, description and example.

No.	char	description	example
3.1	-	subtraction	7 - 2 = 5
3.2	*	multiplication	7 * 2 = 14
3.3	/	division	7 / 2 = 3.5
3.4	\	integer division	7 \ 2 = 3
3.5	^	exponentiation	7 ^ 2 = 49
3.6	+	addition	7 + 2 = 9
3.7	<	less than	(7 < 2) = 0 (false) (2 < 7) = 1 (true)
3.8	<=	less than or equal	(7 <= 7) = 1 (true)
3.9	<>	not equal	(7 <> 7) = 0 (false) (7 <> 2) = 1 (true)
3.10	=	equal	(7 = 2) = 0 (false) (7 = 7) = 1 (true)
3.11	=<	equal or less than	same as 3.8
3.12	=>	equal or greater than	(7 => 7) = 1 (true)
3.13	>	greater than	(7 > 2) = 1 (true) (2 > 7) = 0 (false)
3.14	>=	greater than or equal	same as 3.12

3.15 ABS

Purpose:
To return the absolute value of the expression `n`.

Syntax:
`ABS(n)`

Comments:

`n` must be a numeric expression.

Examples:

```
PRINT ABS(7*(-5))
 35
```

Prints 35 as the result of the action.

3.16 AND

Purpose:
Operator for logical AND function

Syntax:
`AND`

Comments:

`AND` is a bitwise operation

Example:

```
PRINT 3 AND 6
 2
```

Both 3 and 6 have the second bit 1 and no other common bits.

3.17 AS

Purpose:
Separation between parameters in some commands

Syntax:
AS

Comments:

AS is not a real function, but is used to make some commands more readable.

Used in: OPEN, NAME and FONT.

Examples:

```
OPEN "TEMP.TXT" FOR INPUT AS #1

NAME "ACCTS" AS "LEDGER"

10 FONT LOAD "ARROWS.FNT" AS #9 'load the font
20 FONT #9
30 PRINT "0"
```

3.18 ASC

Purpose:
To return a numeric value that is the ASCII code for the first character of the string x$.

Syntax:
ASC(x$)

Comments:

If x$ is null, 0 is returned.

If x$ begins with an uppercase letter, the value returned will be within the range of 65 to 90.

If x$ begins with a lowercase letter, the range is 97 to 122.

Numbers 0 to 9 return 48 to 57, sequentially.

See the CHR$ function for ASCII-to-string conversion.

See ASCII codes appendix for ASCII codes.

Examples:

```
10 X$="TEN"
20 PRINT ASC(X$)
RUN
84
```

84 is the ASCII code for the letter T.

3.19 ATN

Purpose:
To return the arctangent of x, when x is expressed in radians.

Syntax:
ATN(x)

Comments:

The result is within the range of -π/2 to π/2.

The expression x may be any numeric type.

To convert from degrees to radians, multiply by π/180.

Examples:

```
10 INPUT X
20 PRINT ATN(X)
RUN
? 3
1.24905
```

Prints the arctangent of 3 radians (1.24905).

3.20 BYTE2NUM

Purpose:
Return the number created by storing the four arguments as consecutive bytes

Syntax:
`BYTE2NUM(array(x))`
`BYTE2NUM(byte1,byte2,byte3,byte4)`

Comments:

MM-BASIC numbers are stored as the C float type and are four bytes in length.

The bytes can be supplied as four separate numbers or as four elements of `array` starting at index `x`.

See the command `NUM2BYTE` for the reverse of this function.

Example:

This code changes MOD-IO address by I2C commands:

```
5 'edit line 40 to change new address
10 CLS
20 INPUT "Press Hold But on MOD-IO then hit enter ";a
30 CurI2c = &h58
40 NewI2c = &h5a
50 I2CEN 100,100 ' Enable I2C
60 I2CSEND CurI2c,1,2, &hf0, NewI2c
70 I2CDIS
80 END
```

3.21 CANCLOCK

Purpose:
To get the value of the clock of the active CAN controller

Syntax
CANCLOCK

Comments:

The resolution is dependent on the resolution set by `CANOPEN`. Default resolution is 100 uS, but it can be as low as 1 uS.

The canclock is reset by the `CANOPEN` command. It has already started at boot up.

The CAN timestamps are compared with the 1 ms clock and are corrected if the internal CAN clock has an overflow. The total timestamp is a 32 bit value. If the resolution is set to 1uS, it means that the maximum measurement time is 1.25 hours.

Value of `CANCLOCK` is included in `CANSTATUS`.

Examples:

```
PRINT CANCLOCK
 123456

x = CANCLOCK
```

3.22 CANOBJECT

Purpose:
To get some data of existing CANOBJECTs.

Syntax:
CANOBJECT

Comments:

For the objects 0 to 31 status = CANOBJECT(objectno) will give status the value 0 if it is not active and the value of the lower 16 bits of the first address of the FIFO for this object. The higher 16 bits can be read by the CANSTATUS command.

Example:

```
IF CANOBJECT(0) > 0 THEN POKE &HA000, CANOBJECT(0)+8, PIN(0)
```

This Basic instruction will replace the databyte 0 in the FIFO by the actual value of PIN(0). In this example we use the default configuration with 64 FIFO's and FIFO 0 set as TX FIFO. If the FIFO's are in the Basic memory area, the first parameter will probably be &HA001. Using the POKE command has quite some risk, but it is much faster than searching for the Basic parameter every time.

counter = CANOBJECT(32) will give counter the value of the total object counter.

Besides the total counter, counters are included for every object separately. These counters can be read by the function: counter = CANOBJECT(1xx), where xx has the value of 00 up to 31.

status = CANOBJECT(2xx) will give the object status of xx (value as in CANSTATUS).

timer = CANOBJECT(3xx) will give timer the actual timer value of xx.

dependant = CANOBJECT(4xx) will give dependant the value of the objectno. of which xx is dependant (0 if not used).

See also the command CANOBJECT.

3.23 CHR$

Purpose:To convert an ASCII code to its equivalent character.

Syntax:
`CHR$(n)`

Comments:

`n` is a value from 0 to 255.

`CHR$` is commonly used to send a special character to the terminal or printer. For example, you could add `CHR$(13)` or `CHR$(34)` in string.

See the `ASC()` function for ASCII-to-numeric conversion.

See ASCII codes appendix for ASCII codes.

Examples:

```
PRINT CHR$(66);
B
```

This prints the ASCII character code 66, which is the uppercase letter B.

```
PRINT CHR$(13);
```

This command prints a carriage return.

3.24 CINT

Purpose:
To round numbers with fractional portions to the next whole number or integer.

Syntax:
`CINT(x)`

Comments:
`x` should integer number range. See also INT() and FIX() both of which return integers.

Examples:

```
PRINT CINT(45.67)
 46
```

45.67 is rounded up to 46.

```
PRINT CINT(-35.54)
 -36
```

-35.54 is rounded up to -36.

3.25 COS

Purpose:
To return the cosine of the argument `x` in radians.

Syntax:
`COS(x)`

Comments:

`x` must be in radians. `COS` is the trigonometric cosine function. To convert from degrees to radians,multiply by π/180.

Example 1:

```
10 X=2*COS(.4)
20 PRINT X
RUN
 1.84212
```

Example 2:

```
10 PI=3.14159
20 PRINT COS(PI)
30 DEGREES=180
40 RADIANS=DEGREES*PI/180
50 PRINT COS(RADIANS)
RUN
 -1 -1
```

3.26 CWD$

Purpose:
To return the current working directory on the SD card as a string.

Syntax:
CWD$

Comments:

Can be used in string expression.

Example:

```
PRINT "CURRENT WORKING DIRECTORY IS: ";CWD$
CURRENT WORKING DIRECTORY IS: \
```

3.27 DATE$

Purpose:
To retrieve the current date.

Syntax:
DATE$

Comments:

The date is set to “01-01-2000” at power up. If MOD-RTC (Real-Time-Clock Module) is connected to UEXT the current date will be set form the read content from MOD-RTC.

The current date (as assigned when the operating system was initialized) is fetched and assigned to the string variable if DATE$ is the expression in a LET or PRINT statement.The current date is stored if DATE$ is the target of a string assignment.

With v$=DATE$, DATE$ returns a 10-character string in the form dd-mm-yyyy. dd is the day (01 to 31), mm is the month (01 to 12) and yyyy is the year (2000 to 9999).

Example:

```
V$=DATE$
PRINT V$
01-01-2000
```

3.28 DOW

Purpose:
To retrieve the day of the week

Syntax:
`DOW`

Comments:

Returns the number of the day in the week.

0 =sunday; 1=monday; 2=tuesday; 3=wednesday; 4=thursday; 5=friday; 6=saturday.

Example:

```
PRINT DOW
 1
```

This means it is monday, presuming `DATE$` has the right date.

This function was already in the original DMBasic 2.7, however not documented.

3.29 ELSE

Purpose:
To activate the `ELSE` command in a single line `IF … THEN … ELSE` statement

Syntax:
`ELSE`

Comments:

A statement starts with a command and further only functions, variables, constants, etc. The `IF … THEN … ELSE` statement however starts a new statement after the `THEN` and `ELSE`, also if it is a single line statement. That is why the pseudo `THEN` and `ELSE` functions are created. When they are detected, the `THEN` and `ELSE` commands are executed.

See further the `IF` command.

3.30 EOF

Purpose:To return true if the end of file has been reached, or to return 0 if end of file (EOF) has not been found.

Syntax:
`EOF([#]file-number)`

Comments:

Will return true if the file previously opened for `INPUT` with the `file-number` is positioned at the end of the file.

If used on a `file-number` opened as a serial port this function will return true if there are no characters waiting in the receive buffer.

The # is optional. Also see the `OPEN`, `INPUT` and `LINE INPUT` commands and the `INPUT$` function.

Examples:

```
10 OPEN "COM1:19200" AS #5
20 IF NOT EOF(#5) THEN PRINT INPUT$(1,#5);
30 GOTO 20
```

Open the COM1 serial port and if character is received in the buffer print it on the screen.

3.31 EXP

Purpose:
To return e (the base of natural logarithms) to the power of x.

Syntax:
EXP(x)

Comments:

x must not cause overflow.

Example:

```
10 x = 5
20 PRINT EXP(x-1)
RUN
 54.5981
```

Prints the value of e to the 4th power.

3.32 FIX

Purpose:
To truncate `x` to a whole number.

Syntax:
`FIX(x)`

Comments:

`FIX` does not round off numbers, it simply eliminates the decimal point and all characters to the right of the decimal point.

`FIX(x)` is equivalent to `SGN(x)*INT(ABS(x))`. The major difference between `FIX` and `INT` is that `FIX` does not return the next lower number for negative `x`. This behavior is for Microsoft compatibility.

`FIX` is useful in modulus arithmetic.

See also `CINT()`.

Examples:

```
PRINT FIX(9.89)
 9

PRINT FIX(-2.11)
 -2
```

3.33 FOR

Purpose:
Separation between parameters in some commands

Syntax:
`FOR`

Comments:

`FOR` is not a real function, but is used to make some commands more readable.

Used in: `OPEN` and `EXIT.`

`FOR` as command has nothing to do with this function.

3.34 FORMAT$

Purpose:
To return a string representing `number` formatted according to the specifications in the string `format$`.

Syntax:
`FORMAT$(number, format$)`

Comments:

The `format$` specification starts with a % character and ends with a letter. Anything outside of this construct is copied to the output as is.

The structure of a format specification is:
`% [flags] [width] [.precision] type`

Where `flags` can be:

-	Left justify the value within a given field width
0	Use 0 for the pad character instead of space
+	Forces the + sign to be shown for positive numbers
space	Causes a positive value to display a space for the sign. Negative values still show the – sign

`width` is the minimum number of characters to output, less than this the number will be padded, more than this the width will be expanded.

`precision` specifies the number of fraction digits to generate with an `e`, or `f` type or the maximum number of significant digits to generate with a `g` type. If specified the precision must be preceded by a dot (`.`).

`type` can be one of:

`g`	Automatically format the number for the best presentation.
`f`	Format the number with the decimal point and following digits
`e`	Format the number in exponential format
h	*Format the number in hexadecimal*
d	*Format the number as integer*

If uppercase `G, F` or *`H`* is used the exponential output will use an uppercase `E` / *`A`* – *`F`*. If the format specification is not specified "`%g`" is assumed.

Formats h and d are added in the CAN version.

Examples:

```
format$(45)
```
will return `45`

```
format$(45,"%g")
```
will return `45`

```
format$(24.1,"%g")
```
will return `24.1`

```
format$(24.1,"%f")
```
will return `24.100000`

```
format$(24.1,"%e")
```
will return `2.410000e+01`

```
format$(24.1,"%09.3f")
```
will return `00024.100`

```
format$(24.1,"%+.3f")
```
will return `+24.100`

```
format$(24.1,"**%-9.3f**")
```
will return `**24.100 **`

`format$(24.1,"%d")`
Will return 24

`format$(24.1,"%h")`
Will return 18

3.35 GETDIM

Purpose:
To get the status of a variable name

Syntax:
`GETDIM(varname$)`

Comments:

`GETDIM` **can be used to check if a variable name has been used already. For an array it can be used to get the size (dimensions) of the array.**

`varname$` **must be a string.**

If the variable is a standard variable (number or string) it will return -1 if it does not exist and 0 if it exists.

If the variable is an element of an one dimensional array it will return the dimension of that array.

If the variable is an element of a multidimensional array it will return the total number of elements of that array.

Example:

```
10 DIM a(20),b(10,10):c=5
20 PRINT GETDIM("a");GETDIM("a(0)");GETDIM("b(2,3)");
GETDIM("c");GETDIM("d");GETDIM("a(25)")
RUN
 -1 20 100 0 -1
ERROR: Array index out of bounds
```

`GETDIM` **cannot be used for an array element which is not within the** `DIM` **specification. Element 0 is always available.**

3.36 GETPIN

Purpose:
To get the configuration of an I/O pin

Syntax:
GETPIN(x)

Comments:

Possible values:

0	**Not configured**
1	**Analogue input**
2	**Digital input**
3	**Frequency input**
4	**Period input**
5	**Count input**
6	**Interrupt Low to High input**
7	**Interrupt High to Low input**
8	**Digital input**
9	**Digital input Open Collector**
18	**PWM input 1%**
19	**PWM 1% Open Collector**
28	**PWM input 0.1%**
29	**PWM 0.1% Open Collector**

Example:

```
10 SETPIN(1,2)
20 PRINT GETPIN(1)
RUN
 2
```

3.37 GOSUB

Purpose:
To activate the `GOSUB` command within the `ON` command

Syntax:
`(ON x )GOSUB line numbers`

Comments:

See `ON` command for details

`ON x GOSUB variables` is an illegal statement and will generate an "Invalid line number" error.

3.38 GOTO

Purpose:
To activate the `GOTO` command within the `ON` or `IF` command.

Syntax:

```
(IF expression) GOTO line number
(ON x) GOTO line numbers
```

Comments:

See `IF` and `ON` commands for details.

3.39 HEX$

Purpose:
To return a string which represents the hexadecimal value of the numeric argument.

Syntax:
`HEX$(x)`

Comments:

`HEX$` converts decimal values within the range of ±1677100 string expression within the range of 0 to FFFFFFFF.

Hexadecimal numbers are numbers to the base 16, rather than base 10 (decimal numbers).

`x` is rounded to an integer before `HEX$(x)` is evaluated.

If `x` is negative, 2's (binary) complement form is used i.e. -1 is FFFFFFFF

Example:

```
10 CLS: INPUT "INPUT DECIMAL NUMBER";X
20 A$=HEX$(X)
30 PRINT X "DECIMAL IS "A$" HEXADECIMAL"
RUN
INPUT DECIMAL NUMBER? 32
32 DECIMAL IS 20 HEXADECIMAL
```

3.40 INKEY$

Purpose:
To return one character read from the keyboard.

Syntax:
INKEY$

Comments:

If no character is pending in the keyboard buffer, a null string (length zero) is returned.

If several characters are pending, only the first is returned.

No characters are displayed on the screen, and all characters except the following are passed to the program:

CTRL-BREAK
CTRL-NUM LOCK
CTRL-ALT-DEL
CTRL-PRTSCR
PRTSCR

Example:

```
10 CLS 'game moves rectangle 1,2,3 keys are used
20 X = 0 : D = 1
30 LINE (X*20,100)-(X*20+20,105),1,BF
40 PAUSE 100
50 LINE (X*20,100)-(X*20+20,105),0,BF
60 C$ = INKEY$
70 IF C$ = "1" THEN D = -1 'move right to left
80 IF C$ = "2" THEN D = 1 'move left to right
90 IF C$ = "3" THEN END 'quit
100 X = X + D
110 IF X > 23 THEN X = 0
120 IF X < 0 THEN X = 23
130 GOTO 30
```

3.41 INPUT$

Purpose:
To read and return characters from file.

Syntax:
`INPUT$(number, [#]file-number)`

Comments:

Will return a string composed of `number` characters read from a file previously opened for `INPUT` with the file number `file-number`. This function will read all characters including carriage return and new line without translation.

When reading from a serial communications port this will return as many characters as are waiting in the receive buffer up to `number`. If there are no characters waiting it will immediately return with an empty string.

The # is optional. Also see the `OPEN` command.

Examples:

In the following example, MOD-GPS is connected to DuinoMite-Mega `UEXT` and the GPS message in NMEA format is read and will display it on the screen.

```
10 OPEN "COM3:19200" AS #3
20 PRINT INPUT$(1,#3);
30 GOTO 20
```

3.42 INSTR

Purpose:
To search for the first occurrence of string y$ in x$, starting from position n and return the position at which the string is found.

Syntax:
INSTR([n,]x$,y$)

Comments:

Optional offset n sets the position for starting the search. The default value for n is 1. If n equals zero, the error message "Number out of bounds" is returned. n must be within the range of 1 to 255. If n is out of this range, an "Number out of bounds" error is returned.

INSTR returns 0 if:

- n > LEN(x$)
- x$ is null
- y$ cannot be found

If y$ is "", INSTR returns n.
x$ and y$ may be string variables, string expressions, or string literals.

Example:

```
10 X$="ABCDEBXYZ"
20 Y$="B"
30 PRINT INSTR(X$, Y$); INSTR(4, X$, Y$)
RUN
 2 6
```

The interpreter searches the string "ABCDFBXYZ" and finds the first occurrence of the character B at position 2 in the string. It then starts another search at position 4 (D) and finds the second match at position 6 (B). The last three characters are ignored, since all conditions set out in line 30 were satisfied.

3.43 INT

Purpose:
To truncate an expression to a whole number.

Syntax:
`INT(n)`

Comments:

Negative numbers return the next lowest number. This behaviour is for Microsoft compatibility, the `FIX()` function provides a true integer function.

The `FIX` and `CINT` functions also return integer values.

Examples:

```
PRINT INT(98.89)
 98

PRINT INT(-12.11)
 -13
```

3.44 LCASE$

Purpose:
To return `x$` converted to lowercase characters.

Syntax:
`LCASE$(x$)`

Comments:

`x$` may be empty string.

Example:

```
PRINT LCASE$("qWeRTyUIop")
qwertyuiop
```

3.45 LEFT$

Purpose:
To return a string that comprises the leftmost `n` characters of `x$`.

Syntax:
`LEFT$(x$,n)`

Comments:

`n` must be within the range of 0 to 255. If `n` is greater than `LEN(x$)`, the entire string (`x$`) will be returned. If `n` equals zero, the null string (length zero) is returned (see the `MID$()` and `RIGHT$()` substring functions).

Example:

```
10 A$="BASIC"
20 B$=LEFT$(A$, 3)
30 PRINT B$
RUN
BAS
```

The left-most three letters of the string "BASIC" are printed on the screen.

3.46 LEN

Purpose:
To return the number of characters in x$.

Syntax:
LEN(x$)

Comments:

x$ is any string expression. Non-printing characters and blanks are counted.

Example:

```
10 X$="PORTLAND, OREGON"
20 PRINT LEN(X$)
RUN
 16
```

Note that the comma and space are included in the character count of 16.

3.47 LOAD

Purpose:
To load a custom font by using the `FONT` command.

Syntax:
`(FONT )LOAD "fontfile" AS [#]n`

Comments:

See `FONT` command for details

Function `LOAD` is never used to load a Basic file; this is the command `LOAD`.

3.48 BLOAD

Purpose:
To load a custom binary font by using the `FONT` command.

Syntax:
`(FONT )BLOAD "fontfile" AS [#]n`

Comments:

See `FONT` command for details.

`FONT BLOAD "fontfile" AS [#]n` is for system level developers only.

3.49 LOC

Purpose:
To return the number of bytes waiting in the receive buffer of a serial port (ie, COM1:, COM2:, COM3:or COM4:).

Syntax:
`LOC([#]file-number)`

Comments:

`file-number` is the file number used when the Serial port was opened. The `#` is optional.

When transmitting or receiving a file through a communication port, `LOC` returns the number of characters in the input buffer waiting to be read. The default size for the input buffer is 127 characters.

If there are more than 127 characters received only the last 127 remain in the buffer, `LOC` returns 127.

If fewer than 127 characters remain in the buffer, then `LOC` returns the actual count.

Examples:

```
200 IF LOC(#1)>5 THEN 300 'process the input message when
5 are read
```

The program jumps to line 300 after 5 characters are read.

3.50 LOF

Purpose:
To return the free space in the transmit buffer.

Syntax:
`LOF([#]file-number)`

Comments:

`file-number` is the number of the file that the file was opened under.

With communications files, `LOF` returns the amount of free space in the input buffers.

Examples:

The following example reads message only when the buffer is half filled:

```
10 OPEN "COM1:9600" AS #1
20 IF LOF(#1) < 64 THEN MSG$=INPUT$(64,#1)
```

3.51 LOG

Purpose:
To return the natural logarithm of `x`.

Syntax:
`LOG(x)`

Comments:

`x` must be a number greater than zero.

Examples:

```
PRINT LOG(2)
 .693147

PRINT LOG(1)
 0

PRINT LOG(0.0001)
 -9.21034
```

3.52 MID$

Purpose:
To return a string of m characters from x$ beginning with the n th character.

Syntax:
MID$(x$,n[,m])

Comments:

n must be within the range of 1 to 255.

m must be within the range of 0 to 255.

If m is omitted, or if there are fewer than m characters to the right of n, all rightmost characters beginning with n are returned.

If n > LEN(x$), MID$ function returns a null string.

If m equals 0, the MID$ function returns a null string.

If either n or m is out of range, an "Number out of bounds" is returned.

Example:

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

Line 30 concatenates (joins) the A$ string to another string with a length of eight characters, beginning at position 8 within the B$ string.

3.53 MM.BLANK

Purpose:
To return the time in seconds when the display will be turned off (screen saver)

Syntax:
MM.BLANK

Comments:

Screen saver must be enabled in SETUP (option n).

MM.BLANK can be given any value by the MM.BLANK command.

Example:

```
100 IF MM.BLANK<10 GOTO 200
…
200 PRINT
210 PRINT CHR$(13);"Display will go down in";MM.BLANK;" seconds";
220 PAUSE 1000
230 IF MM.BLANK=0 THEN END
240 GOTO 210
```

3.54 MM.BOOTUP

Purpose:
To get the bootup status

Syntax:
`MM.BOOTUP`

Comments:

`MM.BOOTUP` has the status after a new boot. Cold start, warm start, start after illegal action, start after sleep, etc.

The table with possible values can be found in `SOFTRESET.`

The decimal value of `MM.BOOTUP` is the same as the hexadecimal Status in the bootup message.

`MM.BOOTUP` was available in the original DMBasic, however not documented.

3.55 MM.DRIVE

Purpose:
For the dummy who forgot that the drive name is a string

Syntax:
`MM.DRIVE`

Comment:

See next page for `MM.DRIVE$`.

3.55 MM.DRIVE$

Purpose:
To get the name of the actual drive

Syntax:
`MM.DRIVE$`

Comment:

Only 2 possibilities: A: or B:

Example:

```
PRINT MM.DRIVE$
B:
```

3.56 MM.ERRNO

Purpose:
To get the kind of error if an error occurs during file access.

Syntax:
`MM.ERRNO`

Comments:

Is set to the error number if a statement involving the SD card fails or zero if the operation succeeds. This is dependent on the setting of `OPTION ERROR`. The possible values for `MM.ERRNO` are:

0 = No error
1 = No SD card found
2 = SD card is write protected
3 = Not enough space
4 = All root directory entries are taken
5 = Invalid filename
6 = Cannot find file
7 = Cannot find directory
8 = File is read only
9 = Cannot open file
10 = Error reading from file
11 = Error writing to file
12 = Not a file
13 = Not a directory
14 = Directory not empty
15 = Hardware error accessing the storage media

See `OPTION ERROR` for an example of how to use it.

3.57 MM.FNAME$

Purpose:
To get the name or the file that will be used as the default for the SAVE command.

Syntax:
MM.FNAME$

Comments:

It is set by the commands LOAD, RUN and SAVE.

An empty string will be returned as long as none of these commands are executed.

Example:

```
PRINT "The actual file name is ";MM.FNAME$
The actual file name is DEMO.BAS
```

3.58 MM.HRES

Purpose:
To get the horizontal resolution of the current video display screen in pixels

Syntax:
`MM.HRES`

Comments:

Resolution is dependent on the settings of `SETUP`.

Resolution can also be influenced by the `OLED` command.

3.59 MM.I2C

Purpose:
To get the result of an I2C operation.

Syntax:
`MM.I2C`

Comments:

Following possibilities for `MM.I2C:`

0 = The command completed without error.
1 = Received a NACK response
2 = Command timed out
4 = Received a general call address (when in slave mode)

3.60 MM.SETUP

Purpose:
To get the status of some parameters from SETUP.

Syntax:
MM.SETUP

Comments:

Five parameters of SETUP are available. They are available in binary as following:
<CUSDH>

<C>: Serial Console (If enabled 16; disabled 0)
<U>: Usb Port (If enabled 8; disabled 0)
<S>: SD Card (If enabled 4; disabled 0)
<D>: Date Format (MM/DD/YY 2; DD/MM/YY 0)
<H>: Hardware (Maximite 1; Duinomite 0; this parameter is always 0)

Example:

```
PRINT MM.SETUP
 12
```

This is the default setting, meaning Usb Port and SD Card enabled; no serial console and Date Format DD/MM/YY

3.61 MM.SLEEP

Purpose:
To notice which event awakened the system from sleep.

Syntax:
`MM.SLEEP`

Comments:

The values can be found at the `SLEEP` command.

3.62 MM.VER

Purpose:
To get the version number of the software

Syntax:
`MM.VER`

Comments:

The version number of the firmware in the form aa.bbcc where aa is the major version number, bb is the minor version number and cc is the revision number (normally zero but A = 01, B = 02, etc).

In the CAN version of the software this has not been updated anymore. This means it has been frozen on: 2.7014 (major 2; minor 70; revision 14). Use `COPYRIGHT` *for the actual version numbers.*

3.64 MM.VRES

Purpose:
To get the vertical resolution of the current video display screen in pixels

Syntax:
`MM.VRES`

Comments:

Resolution is dependent on the settings of `SETUP`.

Resolution can also be influenced by the `OLED` command.

3.65 MOD

Purpose:
Operator for modulus function

Syntax:
MOD

Comments:

Modulus is the remainder of a division

Example:

```
PRINT 7/2;7\2;7 MOD 2
 3.5 3 1
```

7/2 =3.5 and as integer division 3 with a remainder of 1.

3.66 NOT

Purpose:
Operator for logical NOT function

Syntax:
`NOT`

Comments:

`NOT` is a logical inverse of the value on the right.

The `NOT` operator is highest in precedence so it will bind tightly to the next value. For normal use the expression to be negated should be placed in brackets.

Example:

```
IF NOT (A=3 OR A=8) THEN
```

3.67 OCT$

Purpose:
To convert a decimal value to an octal value.

Syntax:
`OCT$(x)`

Comments:

`x` is rounded to an integer before `OCT$(x)` is evaluated.

`OCT$` converts decimal values within the range of ±1677100 to an octal string expression.

Octal numbers are numbers to the base 8 rather than base 10 (decimal numbers).

See the `HEX$` function for hexadecimal conversion.

Examples:

```
10 PRINT OCT$(18)
RUN
 22
```

Decimal 18 equates to Octal 22.

3.68 OR

Purpose:
Operator for logical OR function

Syntax:
```
OR
```

Comments:

`OR` is a bitwise operation

Example:

```
PRINT 3 OR 6
 7
```

3 has the first and second bit high; 6 has the second and third bit high; so bit 1, 2 and 3 gets high in at least one of the numbers; 111 results in 7 decimal.

3.69 PEEK

Purpose:
To read from a specified memory location.

Syntax:
`PEEK(hiword,loword)`

Comments:

Returns the byte (decimal integer within the range of 0 to 255) read from the specified memory location. `hiword` is the top 16 bits of the address while `loword` is the bottom 16 bits.

See the `POKE` command for notes and warnings related to memory access.

`PEEK` is the complementary function to the `POKE` statement.

3.70 PIN

Purpose:To read from a specified external GPIO port.

Syntax:
PIN(n)

Comments:

If the GPIO ports is initialized as Digital Input with the SETPIN command, zero means digital low, read 1 means digital high.

For GPIO ports initialized with SETPIN command as analogue inputs, it will return the measured voltage as a floating point number between 0 and 3.3 which corresponds to 0V to 3.3V.

For GPIO ports initialized with SETPIN command as Frequency inputs, will return the frequency in Hz (maximum 200KHz).

For GPIO ports initialized with SETPIN command as Period inputs, will return the period in milliseconds.

For GPIO ports initialized with SETPIN command as Count input, will return the count since reset (counting is done on the positive rising edge). The count input can be reset to zero by resetting the pin to counting input (even if it is already so configured).

PIN(0) is a special case which will always return the state of the user push button on the PC board (non zero means that the button is down).

In the CAN version of the software it is also possible to read back the values of GPIO ports which have been configured as output (8, 9, 18, 19, 28, 29).

In the CAN version it is also possible to read a number of sequential digital inputs.

Also see the SETPIN and PIN()= commands.

Example:

In the following example, a LED connected to ARDUINO.A0 and GND will light ON when User button is pressed.

```
10 SETPIN 1,8
20 PIN(1) = PIN(0)
30 GOTO 20
```

3.71 PIXEL

Purpose:
To return the current value of a pixel on the VGA or composite screen.

Syntax:
`PIXEL(x,y)`

Comments:

Zero is off, 1 is on. Most upper-left coordinate is 0,0 the most right-down coordinates are `MM.HRES, MM.VRES`.
See the statement `PIXEL(x,y)=` for setting the value of a pixel.

Example:

```
CLS
PRINT PIXEL(100,100)
 0
```

3.72 POS

Purpose:
To return the current cursor position.

Syntax:
POS

Comments:

The leftmost position is 1.

Example:

```
10 CLS
20 A$ = INKEY$ :IF A$ = "" THEN GOTO 30 ELSE PRINT A$;
30 GOTO 20
40 IF POS > 10 THEN PRINT CHR$(13);CHR$(10);
50 GOTO 30
```

Causes a carriage return and line feed after the 10th character is printed on each line of the screen.

3.73 RIGHT$

Purpose:
To return the rightmost n characters of string x$.

Syntax:
RIGHT$(x$,n)

Comments:

If n is equal to or greater than LEN(x$), RIGHT$ returns x$.

If n equals zero, the null string (length zero) is returned (see the MID$ and LEFT$ functions).

Example:

```
10 A$="DISK BASIC"
20 PRINT RIGHT$(A$, 5)
RUN
BASIC
```

Prints the rightmost five characters in the A$ string.

3.74 RND

Purpose:
To return a random number between 0 and 0.99999.

Syntax:
`RND[(x)]`

Comments:

The same sequence of random numbers is generated each time the program is run unless the random number generator is reseeded (see `RANDOMIZE` statement).

`x` value is ignored if supplied.To get a random number within the range of zero through `n`, use the following formula:

```
INT(RND*(n+1))
```

Examples:

```
PRINT RND
 0.513871

PRINT RND (1) 'prints two numbers due to the space between
RND and (
 0.175726 1

PRINT RND(-1)
 0.94763

PRINT INT(RND*101) 'prints random number 0 - 100
 53
```

3.75 SGN

Purpose:
To return the sign of x.

Syntax:
SGN(x)

Comments:

x is any numeric expression.

If x is positive, SGN(x) returns 1.

If x is 0, SGN(x) returns 0.

If x is negative, SGN(x) returns -1.

Examples:

```
10 INPUT "Enter value", x
20 ON SGN(X)+2 GOTO 100, 200, 300
```

MM-BASIC branches to 100 if X is negative, 200 if X is 0, and 300 if X is positive.

3.76 SIN

Purpose:
To calculate the trigonometric sine of `x`, in radians.

Syntax:
`SIN(x)`

Comments:

To obtain `SIN(x)` when `x` is in degrees, use
`SIN(x*π/180)`.

Examples:

```
PRINT SIN(1.5)
 0.997495
```

The sine of 1.5 radians is 0.997495.

3.77 SPACE$

Purpose:
To return a string of `x` spaces.

Syntax:
`SPACE$(x)`

Comments:

`x` is rounded to an integer and must be within the range of 0 to 255. If the number is outside this range error "`Number out of bounds`" results.

Example:

```
10 FOR N=1 TO 5
20 X$=SPACE$(N)
30 PRINT X$; N
40 NEXT N
RUN
  1
   2
    3
     4
      5
```

Line 20 adds one space for each loop execution.

3.78 SPC

Purpose:
To skip a specified number of spaces in a PRINT.

Syntax:
SPC(n)

Comments:

n must be within the range of 0 to 255.

If n is greater than the defined width of the printer or the screen, the value used will be n MOD width.

This function is similar to the SPACE$() function and is only included for Microsoft compatibility.

Examples:

```
PRINT "OVER" SPC(15) "THERE"
OVER               THERE
```

3.79 SPI

Purpose:
To send and receive a byte using SPI protocol.

Syntax:
`SPI(rx, tx, clock[, data][, speed])`

Comments:

MM-BASIC is the master (i.e. it generates the clock).

`rx` is the `GPIO` port number for the data input (aka `MISO` master-input slave-output)

`tx` is the `GPIO` port number for the data output (aka `MOSI` master-output slave-input)

`clock` is the `GPIO` port number for the clock generated by MM-MASIC (aka `CLK`)

`data` is optional and is an integer representing the data byte to send over the data output pin. If it is not specified the `tx` pin will be held low as if the `data` is 0.

`speed` is optional and is the speed of the clock. It is a single letter either `H, M` or `L` where `H` is 500KHz, `M` is 50KHz and `L` is 5KHz. Default is `H`.

Examples:

```
10 SETPIN 1,2 'define Rx as input
20 SETPIN 2,8 'define Tx as output
30 SETPIN 3,8 'define clock as output
40 PRINT SPI(1,2,3,255,H) 'send FF and receive one byte
RUN
 255
```

3.80 SQR

Purpose:
Returns the square root of `x`.

Syntax:
`SQR(x)`

Comments:

`x` must be greater than or equal to 0.

Example:

```
10 FOR X=10 TO 25 STEP 5
20 PRINT X; SQR(X)
30 NEXT
RUN
 10 3.16228
 15 3.87298
 20 4.47214
 25 5
```

3.81 STEP

Purpose:
To define the step for the next value in `FOR … TO … STEP … NEXT`.

Syntax:
`STEP`

Comments:

See `FOR` for the use of this function.

3.82 STR$

Purpose:
To return a string representation of the decimal (base 10) value of `x`.

Syntax:
`STR$(x)`

Comments:

`STR$(x)` is the complementary function to `VAL(x$)`.

Examples:

```
10 INPUT "TYPE A NUMBER: ", N
20 PRINT "THIS IS A";LEN(STR$(N));" DIGIT NUMBER"
RUN
TYPE A NUMBER: 123
THIS IS A 3 DIGIT NUMBER
```

3.83 STRING$

Purpose:
To return

- a string of length `n` whose characters all have ASCII code `j`, or
- the first character of `x$`

Syntax:

```
STRING$(n,j)
STRING$(n,x$)
```

Comments:

`STRING$` is also useful for printing top and bottom borders on the screen.

`n` and `j` are integer expressions in the range 0 to 255.

Example:

```
10 X$ = STRING$(10, 45)
20 PRINT X$ "MONTHLY REPORT" X$
RUN
----------
MONTHLY REPORT
----------
```

45 is the decimal equivalent of the ASCII symbol for the minus (-) sign.

3.84 TAB

Purpose:
Spaces to position `n` on the screen.

Syntax:
`TAB(n)`

Comments:

If the current print position is already beyond space `n`, `TAB` goes to that position on the next line.

Space 1 is the leftmost position. The rightmost position is the screen width.

`n` must be within the range of 1 to 255.

It is as though the `TAB` function has an implied semicolon after it.

`TAB` may be used only in `PRINT` or `PRINT #` statements.

Examples:

```
10 PRINT "HELLO" TAB(3) "WORLD"
RUN
HELLO
  WORLD

10 PRINT “HELLO” TAB(10) “WORLD”
RUN
HELLO    WORLD
```

3.85 TAN

Purpose:
To calculate the trigonometric tangent of `x`, in radians.

Syntax:
`TAN(x)`

Comments:

To obtain `TAN(x)` when x is in degrees, use `TAN(x*pi/180)`.
Examples:

```
10 Y = TAN(X)
```

When executed, Y will contain the value of the tangent of X radians.

3.86 THEN

Purpose:
To activate the `THEN` command in a single line `IF … THEN … [ELSE]` statement

Syntax:
`THEN`

Comments:

A statement starts with a command and further only functions, variables, constants, etc. The `IF … THEN … [ELSE]` statement however starts a new statement after the `THEN` [and `ELSE`], also if it is a single line statement. That is why the pseudo `THEN` and `ELSE` functions are created. When they are detected, the `THEN` and `ELSE` commands are executed.

See further the `IF` command.

3.87 TIME$

Purpose:
To retrieve the current time.

Syntax:
`TIME$`

Comments:

The current time is fetched and assigned to the string variable if `TIME$` is the expression in a `LET` or `PRINT` statement.

`TIME$` returns an 8-character string in the form `hh:mm:ss`.

The time is set to “00:00:00” at power up.

Examples:

```
PRINT TIME$
08:00:05
```

3.88 TIMER

Purpose:
To read TIMER value.

Syntax:
`TIMER`

Comments:

`TIMER` function returns the elapsed time in milliseconds (e.g. 1/1000 of a second) since reset. If not specifically reset this count will wrap around to zero after 49 days.The timer is reset to zero on power up and you can also reset it by using `TIMER =` command.

3.89 TO

Purpose:
To define the end value in `FOR … TO … STEP … NEXT`.

Syntax:
`TO`

Comments:

See `FOR` for the use of this function.

3.90 UCASE$

Purpose:
To return `x$` converted to uppercase characters.

Syntax:
`UCASE$(x$)`

Comments:

`x$` may be empty string.

Examples:

```
PRINT UCASE$("qWeRTyUIop")
QWERTYUIOP
```

3.91 UNTIL

Purpose:
To define the end of a `DO ... LOOP`

Syntax:
`UNTIL`

Comments:
`UNTIL` is used at the end of a `DO ... LOOP`

See DO for details.

3.92 VAL

Purpose:
Returns the numerical value of string x$.

Syntax:
VAL(x$)

Comments:

The VAL function also strips leading blanks, tabs, and line feeds from the argument string. For example, the following line returns -3:

```
VAL(" -3")
```

The STR$ function (for numeric to string conversion) is the complement to the VAL(x$) function.

If the first character of x$is not numeric, the VAL(x$) will return zero.

This function will recognize the &H prefix for a hexadecimal number, &O for octal and &B for binary.

Example:

```
10 INPUT "ENTER ZIP:"; ZIP$
20 IF VAL(ZIP$)<1000 OR VAL(ZIP$)>9000 THEN PRINT
"INVALID ZIP"
30 IF VAL(ZIP$) = 4000 THEN PRINT "THIS IS THE ZIP POST
CODE FOR THE CITY OF PLOVDIV WHERE DUINOMITE WAS BORN :)"
```

3.93 WHILE

Purpose:
To define the end of a `DO ... LOOP`

Syntax:
`WHILE`

Comments:
`WHILE` is used at the beginning of a `DO ... LOOP`

See DO for details.

3.94 XOR

Purpose:
Operator for logical XOR function

Syntax:
`XOR`

Comments:

`XOR` stands for exclusive or.

`XOR` is a bitwise operation

Example:

```
PRINT 3 XOR 6
 5
```

Both 3 and 6 have the 2nd bit equal (so 0) and the 1st and 3rd different from each other (so 1).
Result 5