# Storage Commands and Functions

Data files can be opened using OPEN and read from using INPUT, LINE INPUT, or INPUT$() or written to using PRINT or WRITE. On the SD card data is stored using standard text and can be read and edited in Windows, Apple Mac, Linux, etc. An SD card can have up to 10 files open simultaneously while the internal flash drive has a maximum of one file open at a time.

You can list the programs stored on a drive with the FILES command, delete them using KILL and rename them using NAME. On an SD card the current working directory can be changed using CHDIR. A new directory can be created with MKDIR or an old one deleted with RMDIR.

Whenever specified a file name can be a string constant (ie, enclosed in double quotes) or a string variable. This means you must use double quotes if you are directly specifying a file name.  Eg, RUN "TEST.BAS"

**PROGRAM AND DATA STORAGE**

On the CGMMSTICK and CGCOLORMAX two "drives" are available for storing and loading programs and data:

Drive "A:" is a virtual drive using the PIC32's internal flash memory and has a size of about 190KB.

Drive "B:" is the SD card (if present). MMBasic supports MMC, SD or SDHC memory cards formatted as FAT16 or FAT32 with capacities up to the largest that you can purchase.

File names must be in 8.3 format prefixed with an optional drive prefix A: or B: (the same as DOS or Windows). Long file names and long directory names are not supported. The default drive on power up is B: and this can be changed with the DRIVE command.

On the Maximite MMBasic will look for a file on start up called "AUTORUN.BAS" first in the root directory of the internal flash drive (A:) then in the root directory of the SD card (B:). If the file is found it will be automatically loaded and run, otherwise MMBasic will print a prompt (">") and wait for input.

Note that the video output will go blank for a short time while writing data to the internal flash drive A:. This is normal and is caused by a requirement to shut off the video while reprogramming the memory.

When using drive A: you need to be careful not to wear out the flash (the same applies to SD cards). If drive A: is empty, you could write and delete a file on it every day for 175 years before you would reach the endurance limit - but if the interval was once a minute you would reach the limit in about 6 weeks.

# CHDIR

## CHDIR dir$

Change the current working directory on the SD card to 'dir$'

The special entry ".." represents the parent directory of the current directory and "." represents the current directory.

See also:

| | | |
|---|---|---|
| **COPY** | **FILES** | **NAME** |
| **CWD$** | **KILL** | **RMDIR** |
| **DIR$** | **MKDIR** | |
| **DRIVE** | **MM.DRIVE$** | |

# CLOSE (file)

## CLOSE [#]nbr [,[#]nbr] ...

Close the file(s) previously opened with the file number 'nbr'. The # is optional.

See also:

| | | |
|---|---|---|
| **EOF** | **LINE INPUT** | **WRITE** |
| **INPUT#** | **OPEN** | |
| **INPUT$** | **PRINT** | |

# COPY

## COPY src$ TO dest$

Copy the file named 'src$' to another file named 'dest$'.

'dest$' can be just a drive designation (ie, A:) and this makes it convenient to copy files between drives.

Note that it is not possible to have both the source and destination on the internal flash drive A:.

See also:

**CHDIR**
**CWD$**
**DIR$**
**DRIVE**
**FILES**
**KILL**
**MKDIR**
**MM.DRIVE$**
**NAME**
**RMDIR**

## CWD$

Returns the current working directory on the SD card as a string.

See also:

| | | |
|---|---|---|
| **CHDIR** | **FILES** | **NAME** |
| **COPY** | **KILL** | **RMDIR** |
| **DIR$** | **MKDIR** | |
| **DRIVE** | **MM.DRIVE$** | |

# DIR$

## DIR$( [fspec], [type] )

Will search an SD card for files and return the names of entries found. 'fspec' is a file specification using wildcards the same as used by the FILES command. Eg, "*.*" will return all entries, "*.TXT" will return text files. 'type' is the type of entry to return and can be one of:

VOL Search for the volume label only

DIR Search for directories only

FILE Search for files only (the default if 'type' is not specified)

The function will return the first entry found. To retrieve subsequent entries use the function with no arguments. ie, DIR$( ). The return of an empty string indicates that there are no more entries to retrieve.

This function only operates on the SD card and you must change to the required directory before invoking it.

See also:

**CHDIR**
**COPY**
**CWD$**
**DRIVE**
**FILES**
**KILL**
**MKDIR**
**MM.DRIVE$**
**NAME**
**RMDIR**

# DRIVE

## DRIVE drivespec$

Change the default drive used for file operations that do not specify a drive to that specified in drivespec$.  This can be the string "A:" or "B:".

See also the predefined read only variable MM.DRIVE$.

See also:

| | | |
|---|---|---|
| **CHDIR** | **FILES** | **NAME** |
| **COPY** | **KILL** | **RMDIR** |
| **CWD$** | **MKDIR** | |
| **DIR$** | **MM.DRIVE$** | |

# EOF

## EOF( [#]nbr )

Will return true if the file previously opened for INPUT with the file number 'nbr' 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.

See also:

| | | |
|---|---|---|
| **CLOSE** | **LINE INPUT** | **WRITE** |
| **INPUT#** | **OPEN** | |
| **INPUT$** | **PRINT** | |

## FILES [search_pattern$]

Lists files in the current directory on the SD card.

The SD card (drive B:) may use an optional 'fspec $'. Question marks (?) will match any character and an asterisk (*) will match any number of characters. If omitted, all files will be listed. For example:

*.* Find all entries

*.TXT Find all entries with an extension of TXT

E*.* Find all entries starting with E

X?X.* Find all three letter file names starting and ending with X

See also:

| | | |
|---|---|---|
| **CHDIR** | **DRIVE** | **NAME** |
| **COPY** | **KILL** | **RMDIR** |
| **CWD$** | **MKDIR** | |
| **DIR$** | **MM.DRIVE$** | |

# INPUT #

## INPUT #nbr, list of variables

Allows input from from a file previously opened for INPUT as 'nbr' to a list of variables.

The input must contain commas to separate each data item if there is more than one variable.

For example, if the command is:

```
INPUT #1, a, b, c
```

And the following is in the file:  23, 87, 66

Then a = 23 and b = 87 and c = 66

See also:

| | | |
|---|---|---|
| **CLOSE** | **LINE INPUT** | **WRITE** |
| **EOF** | **OPEN** | |
| **INPUT$** | **PRINT** | |

# INPUT$ (from file)

## INPUT$( nbr, [#]fnbr )

Will return a string composed of 'nbr' characters read from a file previously opened for INPUT with the file number 'fnbr'. This function will read all characters including carriage return and new line without translation.

See also:

| | | |
|---|---|---|
| **CLOSE** | **LINE INPUT** | **WRITE** |
| **EOF** | **OPEN** | |
| **INPUT#** | **PRINT** | |

# KILL

## KILL files$

Deletes the file specified by file$ from the SD card.

Quote marks are required around a string constant and the extension, if there is one, must be specified. Example: KILL "SAMPLE.DAT"

See also:

| | | |
|---|---|---|
| **CHDIR** | **DRIVE** | **NAME** |
| **COPY** | **FILES** | **RMDIR** |
| **CWD$** | **MKDIR** | |
| **DIR$** | **MM.DRIVE$** | |

# LINE INPUT (file)

LINE INPUT #nbr, string-variable$

Reads entire line from the file previously opened for INPUT as ‘nbr’. Unlike INPUT, LINE INPUT will read a whole line, not stopping for comma delimited data items.

See also:

| | | |
|---|---|---|
| **CLOSE** | **INPUT$** | **WRITE** |
| **EOF** | **OPEN** | |
| **INPUT#** | **PRINT** | |

# MKDIR

## MKDIR dir$

Make, or create, the directory 'dir$' on the SD card.

See also:

| | | |
|---|---|---|
| **CHDIR** | **DRIVE** | **NAME** |
| **COPY** | **FILES** | **RMDIR** |
| **CWD$** | **KILL** | |
| **DIR$** | **MM.DRIVE$** | |

## MM.DRIVE$

The current default drive returned as a string containing either “A:” or “B:”.

See also:

| | | |
|---|---|---|
| **CHDIR** | **DRIVE** | **NAME** |
| **COPY** | **FILES** | **RMDIR** |
| **CWD$** | **KILL** | |
| **DIR$** | **MKDIR** | |

## MM.ERRNO

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
16 = Flash memory write failure

See also:

**CHDIR**
**CLOSE**
**COPY**
**CWD$**
**DIR$**
**DRIVE**
**EOF**
**FILES**
**INPUT#**
**INPUT$**
**KILL**
**LINE INPUT**
**MKDIR**
**NAME**
**OPEN**
**OPTION ERROR**
**PRINT**
**RMDIR**
**WRITE**

# NAME

## NAME old$ AS new$

Rename a file or a directory on the SD card from 'old$' to 'new$'

Unlike the other commands that work with file names the NAME command cannot accept a full path name (with directories).

See also:

| | | |
|---|---|---|
| **CHDIR** | **DRIVE** | **MM.DRIVE$** |
| **COPY** | **FILES** | **RMDIR** |
| **CWD$** | **KILL** | |
| **DIR$** | **MKDIR** | |

# OPEN (file)

## OPEN fname$ FOR mode AS [#]fnbr

Opens a file for reading or writing.

'fname' is the file name (8 chars max) with an optional extension (3 chars max) separated by a dot (.). It can be prefixed with a directory path.

For example:

```
OPEN "B:\DIR1\DIR2\FILE.EXT" FOR INPUT AS #1
```

'mode' is INPUT or OUTPUT or APPEND.

INPUT will open the file for reading and throw an error if the file does not exist.

OUTPUT will open the file for writing and will automatically overwrite any existing file with the same name.

APPEND will also open the file for writing but it will not overwrite an existing file; instead any writes will be appended to the end of the file. If there is no existing file the APPEND mode will act the same as the OUTPUT mode (i.e. the file is created then opened for writing).

Note: APPEND is not supported on the flash file system (drive A:).

'fnbr' is the file number (1 to 10). The # is optional. Up to 10 files can be open simultaneously. The INPUT, LINE INPUT, PRINT, WRITE and CLOSE commands as well as the EOF() and INPUT$() functions all use 'fnbr' to identify the file being operated on.

See also OPTION ERROR and MM.ERRNO for error handling.

The length of the path plus the file name (including punctuation) must be less than 127 characters.

The following code will try to open a file. If it is not there, an error occurs.

```
' Stop error from halting program
OPTION ERROR CONTINUE

FileStatus = 1

OPEN "test.txt" FOR INPUT AS #1

    ' Check error condition
```

```
    IF MM.ERRNO = 6 THEN FileStatus = 0

OPTION ERROR ABORT


' If file opened without error...
IF FileStatus = 1 THEN

    ' Process opened file here

ENDIF
```

See also:

| | | |
|---|---|---|
| **CLOSE** | **INPUT$** | **WRITE** |
| **EOF** | **LINE INPUT** | |
| **INPUT#** | **PRINT** | |

# OPTION ERROR

## OPTION ERROR ABORT | CONTINUE

Sets the treatment for errors in file input/output. The option CONTINUE will cause MMBasic to ignore file related errors. The program must check the variable MM.ERRNO to determine if and what error has occurred.

The option ABORT sets the normal behavior (ie, stop the program and print an error message). The default is ABORT.

Note that this option only relates to errors reading or writing from the SD card, it does not affect the handling of syntax and other program errors.

See also:

| | | |
|---|---|---|
| **CHDIR** | **FILES** | **NAME** |
| **CLOSE** | **INPUT#** | **OPEN** |
| **COPY** | **INPUT$** | **PRINT** |
| **CWD$** | **KILL** | **RMDIR** |
| **DIR$** | **LINE INPUT** | **WRITE** |
| **DRIVE** | **MKDIR** | |
| **EOF** | **MM.ERRNO** | |

# PRINT

## PRINT #nbr, expression [[,; ]expression] … etc

Same as PRINT except that the output is directed to a file previously opened for OUTPUT or APPEND as 'nbr'.

See also:

| | | |
|---|---|---|
| **CLOSE** | **INPUT$** | **WRITE** |
| **EOF** | **LINE INPUT** | |
| **INPUT#** | **OPEN** | |

# RMDIR

## RMDIR dir$

Remove, or delete, the directory ‘dir$’ on the SD card.

See also:

| | | |
|---|---|---|
| **CHDIR** | **DRIVE** | **MM.DRIVE$** |
| **COPY** | **FILES** | **NAME** |
| **CWD$** | **KILL** | |
| **DIR$** | **MKDIR** | |

# WRITE

## WRITE [#nbr,] expression [,expression] …

Outputs the value of each 'expression' separated by commas (,). 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.

If '#nbr' is specified the output will be directed to a file on the SD card previously opened for OUTPUT or APPEND as '#nbr'.

WRITE can be replaced by the PRINT command.

See also:

| | | |
|---|---|---|
| **CLOSE** | **INPUT$** | **PRINT** |
| **EOF** | **LINE INPUT** | |
| **INPUT#** | **OPEN** | |

# XMODEM

## XMODEM SEND file$

## XMODEM RECEIVE file$

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.

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 60 seconds.

See also:

| | | |
|---|---|---|
| **CHDIR** | **DRIVE** | **MM.DRIVE$** |
| **COPY** | **FILES** | **NAME** |
| **CWD$** | **KILL** | |
| **DIR$** | **MKDIR** | |

# MMBasic input and output

The CGMMSTICK and CGCOLORMAX are all about interfacing to hardware. MMBasic supports simple control of the data lines as well as asynchronous serial communication, SPI, I2C, PWM, analog input, and other useful interface functions.

**INPUT AND OUTPUT**

You can configure an external I/O pin using the SETPIN command, set its output using the PIN()= command and read the current input value using the PIN() function. Digital I/O uses the number zero to represent a low voltage and any non-zero number for a high voltage. An analogue input will report the measured voltage as a floating point number.

The CGMMSTICK Maximite has 20 I/O pins numbered 1 to 20. (See J1 Pinout) Pins 1 to 10 can be used for analog input and digital input/output with a maximum input voltage of 3.3V. Pins 11 to 20 are digital only but support input voltages up to 5V and can be set to open collector.

Normally digital output is 0V (low) to 3.3V (high) but you can use open collector to drive 5V circuit. This means that the pin can be pulled down (when the output is low) but will go high impedance when the output is high. With a pull up resistor to 5V an output configured as open collector you can drive 5V logic signals. Typical value of the pull up resistor is 1K to 4.7K.

**CGCOLORMAX "ARDUINO SHIELD" CONNECTOR**

In addition to the 20 I/O pins described above the CGCOLORMAX Maximite has an extra 20 I/O pins on the Arduino compatible connector (40 I/O pins in total). These are labeled D0 to D13 and A0 to A5.

You can use the labels D0, D1, etc in the SETPIN and PIN statements or you can use their corresponding numbers (D0 = 21, D1 = 22, etc and A0 = 35, A1 = 36, etc). The digital pins (D0 to D13) have the same characteristics (5V, open collector, etc) as the digital pins 11 to 20 and the analog capable pins (A0 to A5) have the same capabilities as pins 1 to 10.

# Communications

Two serial ports are supported with speeds up to 19200 baud with configurable buffer sizes and optional hardware flow control. The serial ports are opened using the OPEN command and any command or function that uses a file number can be used to send and receive data.

Communications to slave or master devices on an I2C bus is supported with eight commands MMBasic fully supports bus master and slave mode, 10 bit addressing, address masking and general call, as well as bus arbitration (ie, bus collisions in a multi-master environment).

The Serial Peripheral Interface (SPI) communications protocol is supported with the SPI command.

The 1-Wire protocol is also supported.

# Interrupts

Any external I/O pin can be configured to generate an interrupt using the SETPIN command with up to 29 interrupts (including the tick interrupt) active at any one time. Interrupts can be set up to occur on a rising or falling digital input signal and will cause an immediate branch to a specified line number or label (similar to a GOSUB). The target can be the same or different for each interrupt. Return from an interrupt is via the IRETURN statement. All statements (including GOSUB/RETURN) can be used within an interrupt.

If two or more interrupts occur at the same time they will be processed in order of pin numbers (ie, an interrupt on pin 1 will have the highest priority). During processing of an interrupt all other interrupts are disabled until the interrupt routine returns with an IRETURN. During an interrupt (and at all times) the value of the interrupt pin can be accessed using the PIN() function.

Interrupts can occur at any time but they are disabled during INPUT statements. If you need to get input from the keyboard while still accepting interrupts you should use the INKEY$ function. When using interrupts the main program is completely unaffected by the interrupt activity unless a variable used by the main program is changed during the interrupt.

For most programs MMBasic will respond to an interrupt in under 100µS. To prevent slowing the main program by too much an interrupt should be short and execute the IRETURN statement as soon as possible.

Also remember to disable an interrupt when you have finished needing it – background interrupts can cause strange and non-intuitive bugs.

# CAN (Controller Area Network)

These commands provide access to the CAN hardware available on the CGCOLORMAX2.

A CAN module supports up to 32 channels, each channel can be configured to transmit or receive data. For receiving data the CAN module writes received messages into a different FIFO buffer for each channel. One entry in a FIFO buffer is 16 bytes long (4 address bytes, 4 length bytes, 8 data bytes).

These commands allow the user to specify the number of records in each FIFO buffer and the internals keep track of the memory requirements and automatically allocates memory from the available heap. A command is provided to free the allocated memory when the CAN interface is no longer required.

Note that CAN channels are numbered from 0 to 31 and that, per the Microchip documentation, CAN channels should be configured contiguously from 0 upwards. I.e. if 5 channels are required use channels 0, 1, 2, 3 & 4; and not 1, 2, 3, 4 & 5 or 3, 5, 10, 22, 31 or any other combination.

The CAN module is first configured by placing it in configuration mode and then issuing configuration commands. When the desired configuration is set up then the CAN module is enabled. Once enabled the RX and TX commands can be used to receive and send data on the bus. Once the CAN module is finished it should be disabled and the memory associated with the FIFO buffers should be freed (for example at the end of the application)

An RX channel can either have a filter for a specific id or be configured to receive all messages. In the latter case the user can request the id of the received command and then perform processing based on the id of the command. However, on a CAN bus with moderate to high utilization, more reliable operation will be achieved by setting up multiple channels with each channel filtering for one id of interest. This is because of the relatively small buffer sizes and the relatively slow operation of the BASIC interpreter – when trying to receive all messages into one buffer the buffer overruns and data is lost. Furthermore, when a channel is being filtered for a single ID it may be appropriate to set the FIFO buffer size to just 1 record, especially if the goal is to update a single value display. This will always give you the most up-to-date data.

The examples provided were designed for (and tested on) a second generation Toyota Prius (Model years 2004 – 2009).

Throughout this overview "CAN commands" have been referred to in the plural. In fact there is only one true CAN command – the other commands are "sub-commands" from this one master. By structuring the commands this way there is less impact on the limitation for the number of top-level/master commands that MMBasic can support.

EXAMPLE ONE – Minimal example of reading one channel

```
' (c) John Harding, 2012

' Example designed for Gen 2 Prius

' Configures connection speed to 500kbps and
' monitors for CAN id 52Ch when data is received
' we calculate the ECT from the appropriate data
' bytes.

' Note that the period of this message is
' approximately 1Hz

Cls
Dim ok
Dim data(8)
Dim ect

CAN CONFIG ok
CAN SETSPEED 500000, ok
CAN ADDRXCHNL 0,&h52C,0,1,ok
CAN ENABLE ok

Timer = 0
Do
  If (Inkey$ = "q") Then Exit
  CAN RX 0,data(0),ok
  If (ok=1) Then
    Print Timer ": ECT = " (data(1)/2)
  EndIf
Loop

CAN FREE

End
```

EXAMPLE TWO – Example of reading all channels and displaying just one (but don't do this!)

```
' (c) John Harding, 2012
```

```
' Example designed for Gen 2 Prius

' Configures connection speed to 500kbps and
' configures a channel to receive all messages
' into a FIFO buffer with 32 records.
'
' When a message with id 52Ch is received
' we calculate the ECT from the appropriate data
' bytes.

' Note that the period of this message is
' approximately 1Hz but that we receive many
' wrong ids before we get the message we want

' This example is provided to contrast with
' example 1.  It is suggested to use example 1
' as the basis for your code.

Dim ok
Dim data(8)
Dim id
Dim typ
Dim length

CAN CONFIG ok
CAN SETSPEED 500000, ok
CAN ADDRXCHNL 0, 0, 0, 32, ok
CAN ENABLE ok
Timer=0
Do
  q$ = Inkey$
  If (q$="q") Then Exit
  CAN RX 0, id, typ, length, data(0), ok
  If (ok=1) Then
    If (id=&H52C) Then
      Print " "
      Print Timer ": " Hex$(id) " : " length " : ECT= " data(1) / 2 "
C"
    Else
      Print Timer ": " Hex$(id) " ";
    Endif
  EndIf
Loop
```

```
CAN FREE
End
```

EXAMPLE THREE – Reading  manufacturer specific PID's on a Toyota Prius 2005.

```
' Example to query PIDs for the battery ECU
' and convert the received data into engineering values
' JDH 10/8/12


Dim ok
Dim txID : Dim txData(8) : Dim txLen
Dim rxID : Dim rxData(8)

txData(0) = 0 : txData(1) = 0 : txData(2) = 0 : txData(3) = 0
txData(4) = 0 : txData(5) = 0 : txData(6) = 0 : txData(7) = 0
txLen = 8 ' always transmit 8 bytes even though payload may be less

txID = &H7E3    ' Battery ECU module
rxID = txID + 8 ' id of reply is 8 higher than module number

' Check the "ok" result, if it fails print a message and exit
Sub checkOK(okay, failed$, succeeded$, xit)
  If (okay=0) Then
    Print failed$
    If (xit=1) Then Exit
  Else
    Print succeeded$
  EndIf
End Sub

' print formatted hex numbers
Function toHex$(val)
  toHex$=""
  If (val<16) Then toHex$="0"
  toHex$ = toHex$ + Hex$(val)
End Function

' print timer : id : len : data
Sub PrintRawData
    Print Timer ": " Hex$(rxId) " : " rxLen " : " toHex$(rxData(0)) " ";
    Print toHex$(rxData(1)) " " toHex$(rxData(2)) " " toHex$(rxData(3)) " ";
```

```
    Print toHex$(rxData(4)) " " toHex$(rxData(5)) " " toHex$(rxData(6)) " ";
    Print toHex$(rxData(7))
End Sub

' make the PID request
Sub SendModeAndPID(mode, pid)
  Local txOK
  txData(0) = 3
  txData(1) = mode
  txData(2) = pid
  CAN TX 0,txID,0,txLen,txData(0),txOk
  checkOK(txOK, "FAILED TO SEND PID", "SENT PID " + Hex$(pid) + " TO " +
Hex$(txId), 0)
End Sub

' Send the acknowledgement after receiving frame
Sub SendReadyForMore()
  Local txOK
  txData(0) = &H30
  txData(1) = 0
  txData(2) = 0
  CAN TX 0,txID,0,txLen,txData(0),txOk
  checkOK(txOK, "FAILED TO SEND PID", "SENT 0x30 TO " + Hex$(txId), 0)
End Sub

' Chains together the initial PID request and the acknowledgement
Sub RequestPID(mode, pid)
  ' Note, we're "cheating" here and simply waiting 10msec and sending the
  ' acknowlegement.  Strictly speaking we should wait for the response
  ' header first (frame 0x10).  This works because in the CAN setup (below)
  ' we've setup a big enough buffer to receive all the frames from the
  ' longest PID.
  SendModeAndPID(mode, pid)
  Pause 10
  SendReadyForMore
End Sub

Sub DisplayD0
  ' retrieve the data (see DisplayCE for commentary)
  RequestPID &H21,&HD0
  Do
    CAN RX 1, rxData(0), ok
  Loop Until (ok=1)
  PrintRawData
  Local b(rxData(1)+15)
  Local j
  Local i
```

```
  j=0
  For i=4 To 7
    b(j)=rxData(i)
    j=j+1
  Next i
  Do
    CAN RX 1, rxData(0), ok
    If (ok=1) Then
      PrintRawData
      For i=1 To 7
        b(j)=rxData(i)
        j=j+1
      Next i
    EndIf
  Loop Until (ok=0)

  ' Convert to engineering units (see DisplayCE for commentary)
  Print "Block Count         = " b(0)
  Print "Time in LOW         = " (256*b(1)+b(2))
  Print "Time in DC Inhibit  = " (256*(b3)+b(4))
  Print "Time in HIGH        = " (256*(b5)+b(6))
  Print "Time in HOT         = " (256*(b7)+b(8))
  Print "Lowest Block        = " b(9)
  Print "Lowest Voltage      = " (2.56*b(10)+0.1*b(11)-327.68)
  Print "Highest Block       = " b(12)
  Print "Highest Voltage     = " (2.56*b(13)+0.1*b(14)-327.68)
  For i=15 To 28
    Print "Block " toHex$(i-14) " Resistance = " (0.001*b(i)) " Ohms"
  Next i
End Sub

Sub DisplayCF
  ' retrieve the data (see DisplayCE for commentary)
  RequestPID &H21,&HCF
  Do
    CAN RX 1, rxData(0), ok
  Loop Until (ok=1)
  PrintRawData
  Local b(rxData(1)+15)
  Local j
  Local i
  j=0
  For i=4 To 7
    b(j)=rxData(i)
    j=j+1
  Next i
```

```
  Do
    CAN RX 1, rxData(0), ok
    If (ok=1) Then
      PrintRawData
      For i=1 To 7
        b(j)=rxData(i)
        j=j+1
      Next i
    EndIf
  Loop Until (ok=0)

  ' Convert to engineering units (see DisplayCE for commentary)
  Print "Air Intake Temp   = " (4.608*b(0)+0.018*b(1)-557.824) " F"
  Print "Fan Motor Voltage = " (0.2*b(2) - 25.6) " V"
  Print "Aux Batt Voltage  = " (0.2*b(3) - 25.6) " V"
  Print "Battery Charge    = " (b(4) - 64)
  Print "Battery Discharge = " (b(5) - 64)
  Print "Delta SOC         = " (0.01 * b(6)) " %"
  Print "Fan Speed         = " b(8)
  Print "Batt Temp 1       = " (4.608*b(10)+0.018*b(11)-557.824) " F"
  Print "Batt Temp 2       = " (4.608*b(12)+0.018*b(13)-557.824) " F"
  Print "Batt Temp 3       = " (4.608*b(14)+0.018*b(15)-557.824) " F"
End Sub

Sub DisplayCE
  ' send the request
  RequestPID &H21,&HCE

  ' retrieve the first frame
  Do
    CAN RX 1, rxData(0), ok
  Loop Until (ok=1)
  PrintRawData
  ' create an array big enough to hold all the data
  ' the +15 is because there always seems to be 1 more frame than expecte
  ' and if the amount of data is 1 larger than a multiple of 8 we need 7
  ' bytes for the remainder of that frame + 8 bytes for the "extra" frame
  Local buffer(rxData(1)+15)

  ' Copy the data from the first frame into the buffer
  Local i
  Local j
  j=0
  For i=4 To 7
    buffer(j)=rxData(i)
    j=j+1
```

```
  Next i

  ' now process the remaining frames
 Do
    CAN RX 1, rxData(0), ok
    If (ok=1) Then
      PrintRawData
      For i=1 To 7
        buffer(j)=rxData(i)
        j=j+1
      Next i     EndIf
  Loop Until (ok=0)

  ' We have the raw data, now convert it to engineering values
  ' These calculations are from USBSeaWolf's spreadsheet available on the
  ' PriusChat.com forum.
  Print "SOC      = " (0.5 * buffer(0)) " %"
  Print "Current  = " (2.56*buffer(1)+0.1*buffer(2)-327.68) " A"
  j=3
  For i=1 To 14
    Print "Block " toHex$(i) " = " (2.56*buffer(j)+0.1*buffer(j+1)-327.68) "
V"
    j=j+2
  Next i
End Sub

''''''''''''''''''''''''''
' START HERE...

Cls

CAN CONFIG ok               : checkOK(ok, "CAN CONFIG FAILED", "", 1)
CAN SETSPEED 500000, ok     : checkOK(ok, "CAN SETSPEED FAILED", "", 1)
CAN ADDTXCHNL 0,1,ok        : checkOK(ok, "CAN ADDTXCHNL FAILED", "", 1)
CAN ADDRXCHNL 1,rxId,0,10,ok : checkOK(ok, "CAN ADDRXCHNL FAILED", "", 1)
CAN ENABLE ok               : checkOK(ok, "CAN ENABLE FAILED", "", 1)
CAN PRINTCONFIG

' Note the above configuration for channel 1
' (a) we're filtering on the response ID (which is the moduleID + 8)
' (b) we create a 10 record buffer this allows us to capture all the
response
'     frames in one go

Print "q:quit, t:0xCE, u:0xCF, v:0xD0 send pid to " Hex$(txID)

Timer = 0
```

```
Do
  k$ = Inkey$
  If (k$ = "q") Then Exit
  If (k$ = "t") Then DisplayCE
  If (k$ = "u") Then DisplayCF
  If (k$ = "v") Then DisplayD0
  CAN RX 1, rxData(0), ok
  If (ok=1) Then PrintRawData
Loop

CAN FREE
```

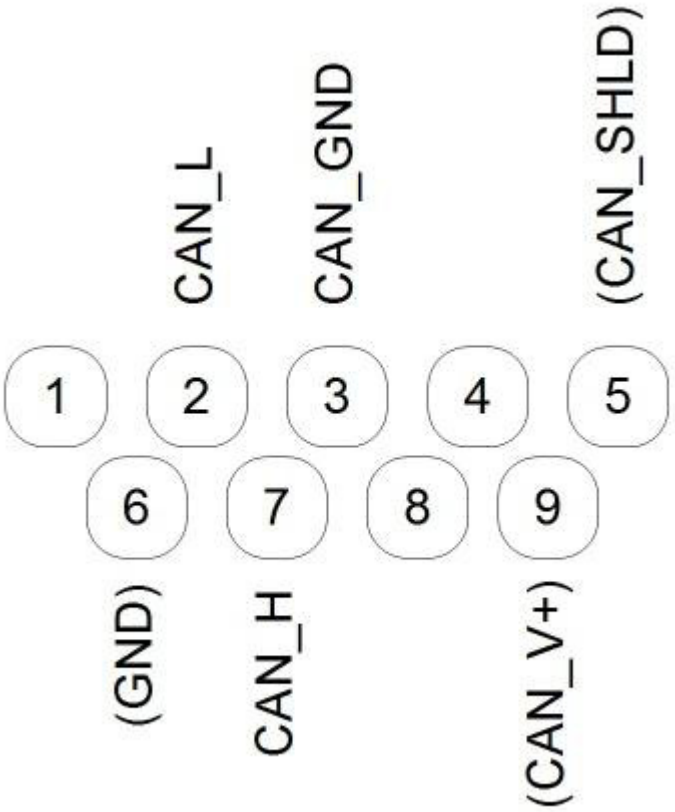



***Illustration 179: CAN-in-Automation connector pin out.***

There are several types of connectors used with CAN. Illustrated here is a standard using a 9-pin D-sub connector. This follows the CiA DS-102 pin out.

| Pin | Signal | Description |
|---|---|---|
| 1 | | |
| 2 | CAN_L | CAN_L bus line |
| 3 | CAN_GND | CAN ground |
| 4 | | |
| 5 | (CAN_SHLD) | CAN shield (optional) |
| 6 | (GND) | Ground (optional) |
| 7 | CAN_H | CAN_H bus line |
| 8 | | |
| 9 | (CAN_V+) | CAN positive supply (optional) |

# Audio and PWM Output

MMBasic there are a number of ways that you can use the sound output. You can play synthesized music, generate tones or generate program controlled voltages (PWM).

**PLAYMOD**

This command will play synthesized music in the background while the program is running. The music must be in the MOD format and the file containing the music must be located on the internal flash drive (drive A:).

The audio is high quality and MMBasic will generate full stereo on the CGCOLORMAX Maximite.

The MOD format is a music file format originating from the MOD file format on Amiga systems in the late 1980s. It is not a recording of the music (like a MP3 file) - instead it contains instructions for synthesizing the music. On the original Amiga this task was performed by dedicated hardware.

MMBasic will read this file and continuously play the music in the background while the program that launched the music will continue running in the foreground. Be aware that synthesizing music is a CPU intensive activity and uses a lot of memory and this could affect the performance of the program.

A description of the MOD format can be found at:
http://en.wikipedia.org/wiki/MOD_(file_format)

A large selection of files that can be played on the Maximite can be found at:
http://modarchive.org (look for files with the .MOD extension). Because the file must be located on drive A: to play it would be wise to select reasonably small files.

You can also create your own music using a tracker. For an example see:
http://www.modplug.com

**TONE**

This command will create two tones for the CGCOLORMAX Maximite that will be outputted separately on the left and right sound channels.  On the monochrome Maximite only one tone is generated. The tone is a synthesized sine wave and can be in the range of 1Hz to 20KHz with a resolution of 1Hz and is very accurate as it is locked to the PIC32's crystal oscillator. When the frequency is changed there is no interruption in the output so the output can be made to glide smoothly across a range of frequencies.

The playing time can be specified in milliseconds and the tone will play in the background (ie, the program continues running).

**SOUND**

The sound command is included only for compatibility with older programs. It generates a single frequency square wave and should be replaced with the tone or PWM command in new programs.

**PWM**

The PWM (Pulse Width Modulation) command allows the Maximite to generate two square waves with programmed controlled duty cycle. By varying the duty cycle you can generate a program controlled voltage outputs for use in controlling external devices that require an analog input (power supplies, motor controllers, etc). he CGCOLORMAX Maximite has two channels while the monochrome CGMMSTICK Maximite has a single channel.

The frequency for both channels is the same and can be from 20Hz to 1MHz. The duty cycle for each channel can be independently set from between 0% and 100% with a 0.1% resolution.

This command uses the sound output for generating the PWM signal so the components on this output may need to be changed to allow this output to work as a PWM output.

# BYTE2NUM

## BYTE2NUM(array(x))

## BYTE2NUM(arg1, arg2, arg3, arg4)

Return the number created by storing the four arguments as consecutive bytes (MMBasic numbers are stored as the C float type and are four bytes in length).

The bytes can be supplied as four separate numbers (arg1 - arg4) or as four elements of 'array' starting at index 'x'.

See the command NUM2BYTE for the reverse of this function.

See also:

| | | |
|---|---|---|
| **I2CDIS** | **I2CSDIS** | **MM.I2C** |
| **I2CEN** | **I2CSEN** | **NUM2BYTE** |
| **I2CRCV** | **I2CSRCV** | |
| **I2CSEND** | **I2CSSEND** | |

## CAN

Displays a list of the available commands.

## CAN PRINTCONFIG

Displays details of the current configuration. The details displayed are module status (online or offline), speed and configured channels.

## CAN CONFIG ok

ok (output) – 1 if successful, 0 otherwise

Clears any pre-existing configuration and puts the CAN module into configuration mode.

## CAN SETSPEED speed, ok

speed (input) – baud rate in kbps from 10,000 to 1,000,000

ok (output) – 1 if successful, 0 otherwise

Sets the baud rate of the CAN connection. Value provided is bits per second with a minimum of 10kbps and a maximum of 1Mbps.

## CAN ADDRXCHNL channel_num, can_id, msg_type, buffer_size, ok

channel_num (input) – a CAN channel from 0 to 31

can_id (input) – a CAN id to filter for (set to 0 to receive all CAN messages)

msg_type(input) – 0 for standard 11-bit IDs, 1 for extended 29-bit IDs

buffer_size(input) – size of the FIFO buffer for this channel expressed as number of records

ok (output) – 1 if successful, 0 otherwise

Configures the specified channel as a receive channel. To filter for an individual id provide the CAN id of interest, to receive all messages pass in a zero id. If you're monitoring a single id (with this channel) and want to act on the latest data set the buffer size to 1. Larger buffer sizes can be set to capture more data – note that no indication is given when buffer overrun occurs (the oldest data is simply discarded).

## CAN ADDTXCHNL channel_num, buffer_size, ok

channel_num (input) – a CAN channel from 0 to 31

buffer_size(input) – size of the FIFO buffer for this channel expressed as number of records

ok (output) – 1 if successful, 0 otherwise

Configures the specified channel as a transmit channel. Normally a buffer size of 1 is sufficient. However, larger sizes allow you to separate the construction and buffering of transmissions from the actual transmission.

# CAN ENABLE ok

ok (output) – 1 if successful, 0 otherwise

Description: Once the configuration is complete call this command to put the CAN module into normal operating mode and ready to receive or transmit data.

# CAN RX channel_num, can_id, msg_type, length, data(8), ok

channel_num (input) – a CAN channel from 0 to 31 for a channel previously configured for RX

data(8) (output) – an array to receive the data from the FIFO record

ok (output) – 1 if successful, 0 if no data available or other failure occurs

This command is intended to read the data only from a channel that has been previously configured to monitor for a given ID (hence the id is already known and doesn't need to be retrieved from the buffer).

Note: The description of the RX and TX commands specify the required size of the data array. However, when calling these functions you will pass in a reference to the first item of the array. Assuming you are using zero-based arrays and an array called data then when you call the function this will be data(0). See the examples for clarification

# CAN RX channel_num, data(8), ok

channel_num (input) – a CAN channel from 0 to 31 for a channel previously configured for RX

can_id (output) – the CAN id of the message read from the FIFO buffer

msg_type (output) – the message type of the message read from the FIFO buffer

length (output) – the amount of data read (between 0 and 8 bytes)

data(8) (output) – an array to receive the data from the FIFO record

ok (output) – 1 if successful, 0 if no data available or other failure occurs

Description: This command is intended to read information from the FIFO buffer of a channel that has been previously configured to receive all messages – hence the need to provide variables to retrieve the full information about the message.

Note: The description of the RX and TX commands specify the required size of the data array. However, when calling these functions you will pass in a reference to the first item of the array. Assuming you are using zero-based arrays and an array called data then when you call the function this will be data(0). See the examples for clarification

# CAN TX channel_num, can_id, msg_type, length, data(8), ok

channel_num (input) – a CAN channel from 0 to 31 for a channel previously configured for RX

can_id (input) – the CAN id to send

msg_type (input) – the message type being sent (0=SID, 1=EID)

length (input) – the amount of data being sent (between 0 and 8 bytes)

data(8) (input) – an array of data bytes to send (values between 0 and 255)

ok (output) – 1 if successful, 0 otherwise

Places data into the FIFO buffer for this channel. Data will be sent on the bus when the CAN module detects the bus is available (i.e. not busy).

Note: The description of the RX and TX commands specify the required size of the data array. However, when calling these functions you will pass in a reference to the first item of the array. Assuming you are using zero-based arrays and an array called data then when you call the function this will be data(0). See the examples for clarification

# CAN DISABLE ok

ok (output) – 1 if successful, 0 otherwise

Description: Puts the module into offline mode, but does not destroy the configuration. This can be used to stop all processing of CAN messages while another processor intensive task takes place. CAN ENABLE can then be called to re-enable the pre-existing configuration.

# CAN FREE

Takes the module off line and frees all memory associated with the existing configuration.

# CLOSE (serial)

## CLOSE [#]nbr [,[#]nbr] ...

Close the serial port(s) previously opened with the file number 'nbr'. The # is optional. Also see the OPEN command.

See also:

| | | |
|---|---|---|
| **CLOSE CONSOLE** | **LOC** | **OPEN** |
| **INPUT$** | **LOF** | **OPEN AS COLSOLE** |

# CLOSE CONSOLE (serial)

## CLOSE CONSOLE

Will close a serial port that had been previously opened as the console.

See also:

| | | |
|---|---|---|
| **CLOSE** | **LOF** | **SETPIN** |
| **INPUT$** | **OPEN** | |
| **LOC** | **OPEN AS COLSOLE** | |

## I2CDIS

Disables the slave I2C module and returns the external I/O pins 12 and 13 to a "not configured" state. Then can then be configured as per normal using SETPIN. It will also send a stop if the bus is still held.

See also:

| | | |
|---|---|---|
| **BYTE2NUM** | **I2CSDIS** | **MM.I2C** |
| **I2CEN** | **I2CSEN** | **NUM2BYTE** |
| **I2CRCV** | **I2CSRCV** | **SETPIN** |
| **I2CSEND** | **I2CSSEND** | |

## I2CEN speed, timeout [, int]

Enables the I2C module in master mode.

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

'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).

'int' 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.

See also:

**BYTE2NUM**
**I2CDIS**
**I2CRCV**
**I2CSEND**
**I2CSDIS**
**I2CSEN**
**I2CSRCV**
**I2CSSEND**
**MM.I2C**
**NUM2BYTE**
**SETPIN**

# I2CRCV

## I2CRCV addr, bus_hold, rcvlen, rcvbuf [,sendlen, senddata [,sendata ....]]

Receive data from the I2C slave device with the optional ability to send some data first.

'addr' 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).

'rcvlen' is the number of bytes to receive.

'rcvbuf' is the variable to receive the data - this is a one dimensional array or if rcvlen 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 'sendlen' and 'senddata'. These parameters are used the same as in the I2CSEND command (ie, senddata can be a constant, an array or a string variable).

```
Examples:
 I2CRCV &h6f, 1, 1, avar

 I2CRCV &h6f, 1, 5, anarray(0)

 I2CRCV &h6f, 1, 4, anarray(2), 3, &h12, &h34, &h89

 I2CRCV &h6f, 1, 3, anarray(0), 4, anotherarray(0)
```

The automatic variable MM.I2C will hold the result of the transaction.

See also:

**BYTE2NUM** **I2CSDIS** **MM.I2C**

**I2CDIS** **I2CSEN** **NUM2BYTE**

**I2CEN** **I2CSRCV** **SETPIN**

**I2CSEND** **I2CSSEND**

# I2CSEND

## I2CSEND addr, option, sendlen, senddata [,sendata ....]

Send data to the I2C slave device.

'addr' 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).

'sendlen' is the number of bytes to send.

'senddata' 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 honored 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 will hold the result of the transaction.

See also:

**BYTE2NUM** **I2CSDIS** **MM.I2C**

**I2CDIS** **I2CSEN** **NUM2BYTE**

**I2CEN** **I2CSRCV** **SETPIN**

**I2CRCV** **I2CSSEND**

## I2CSDIS

Disables the slave I2C module and returns the external I/O pins 12 and 13 to a "not configured" state.  Then can then be configured as per normal using SETPIN.

See also:

**BYTE2NUM**
**I2CDIS**
**I2CEN**
**I2CRCV**
**I2CSEND**
**I2CSEN**
**I2CSRCV**
**I2CSSEND**
**MM.I2C**
**NUM2BYTE**
**SETPIN**

# I2CSEN

## I2CSEN addr, mask, option, send_int, rcv_int

Enables the I2C module in slave mode.

'addr' 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 MMBasic 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.

See also:

**BYTE2NUM** **I2CSEND** **MM.I2C**

**I2CDIS** **I2CSDIS** **NUM2BYTE**

**I2CEN** **I2CSRCV** **SETPIN**

**I2CRCV** **I2CSSEND**

# I2CSRCV

## I2CSRCV rcvlen, rcvbuf, rcvd

Receive data from the I2C master device. 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.

'rcvlen' is the maximum number of bytes to receive.

'rcvbuf' is the variable to receive the data - this is a one dimensional array or if rcvlen is 1 then this may be a normal variable. The array subscript does not have to be zero and will be honored, also bounds checking is performed.

'rcvd' will contain actual number of bytes received by the command.

See also:

**BYTE2NUM** **I2CSEND** **MM.I2C**

**I2CDIS** **I2CSDIS** **NUM2BYTE**

**I2CEN** **I2CSEN** **SETPIN**

**I2CRCV** **I2CSSEND**

# I2CSSEND

## I2CSSEND sendlen, senddata [,sendata ....]

Send the data to the I2C master. This command should be used in the send interrupt (ie in the 'send_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.

'sendlen is the number of bytes to send.

'senddata' is the data to be sent. This can be specified in various ways, see the I2CSEND commands for details.

See also:

| | | |
|---|---|---|
| **BYTE2NUM** | **I2CSEND** | **MM.I2C** |
| **I2CDIS** | **I2CSDIS** | **NUM2BYTE** |
| **I2CEN** | **I2CSEN** | **SETPIN** |
| **I2CRCV** | **I2CSRCV** | |

# INPUT$ (from serial)

## INPUT( nbr, [#]fnbr )

Will return a string composed of 'nbr' characters read from a file previously opened for INPUT with the file number 'fnbr'. 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 'nbr'. If there are no characters waiting it will immediately return with an empty string. The # is optional. Also see the OPEN command.

See also:

| | | |
|---|---|---|
| **CLOSE** | **LOC** | **OPEN** |
| **CLOSE CONSOLE** | **LOF** | **OPEN AS COLSOLE** |

# LOC

## LOC( [#]nbr )

Will return the number of bytes waiting in the receive buffer of a serial port (ie, COM1: or COM2:) that has been opened as #nbr. The # is optional.

See also:

| | |
|---|---|
| **CLOSE** | **LOF** |
| **CLOSE CONSOLE** | **OPEN** |
| **INPUT$** | **OPEN AS COLSOLE** |

# LOF

## LOF( [#]nbr )

Will return the space (in bytes) remaining in the transmit buffer of a serial port (ie, COM1: or COM2:) that has been opened as #nbr. The # is optional.

See also:

| | | |
|---|---|---|
| **CLOSE** | **INPUT$** | **OPEN** |
| **CLOSE CONSOLE** | **LOC** | **OPEN AS COLSOLE** |

## 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)

See also:

**BYTE2NUM** **I2CSEND** **I2CSSEND**

**I2CDIS** **I2CSDIS** **NUM2BYTE**

**I2CEN** **I2CSEN**

**I2CRCV** **I2CSRCV**

## MM.OW

MM.OW variable is set to 1 = OK (one-wire device presence detected, search successful) or 0 = Fail (presence not detected, search unsuccessful).

See also:

**OWCRC8** **OWREAD** **OWSEARCH**

**OWCRC16** **OWRESET** **OWWRITE**

# NUM2BYTE

## NUM2BYTE number, array(x)

## NUM2BYTE number, variable1, variable2, variable3, variable4

Convert 'number' to four numbers containing the four separate bytes of 'number' (MMBasic 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'.

See the function BYTE2NUM() for the reverse of this command.

See also:

**BYTE2NUM** **I2CSEND** **I2CSSEND**

**I2CDIS** **I2CSDIS** **MM.I2C**

**I2CEN** **I2CSEN**

**I2CRCV** **I2CSRCV**

# OPEN (serial)

## OPEN comspec$ AS [#]fnbr

Will open a serial port for reading and writing.  Two ports are available (COM1: and COM2:) and both can be open simultaneously.

'comspec$' is the serial port specification and has the form:

"COMn: baud, buf, int, intlevel"  with an optional ",FC" and/or ",OC" appended.

COM1: uses pin 15 for receive data and pin 16 for transmit data and if flow control is specified pin 17 for RTS and pin 18 for CTS.

COM2: uses pin 19 for receive data and pin 20 for transmit data on the CGMMSTICK and D0 (receive) and D1 (transmit) on the CGCOLORMAX.

If the port is opened using 'fnbr' the port can be written to and read from using any commands or functions that use a file number.

See also:

**CLOSE**
**CLOSE CONSOLE**
**INPUT$**
**LOC**
**LOF**
**OPEN AS COLSOLE**
**SETPIN**

# OPEN AS CONSOLE (serial)

## OPEN comspec$ AS console

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.

See also:

| | | |
|---|---|---|
| **CLOSE** | **LOC** | **SETPIN** |
| **CLOSE CONSOLE** | **LOF** | |
| **INPUT$** | **OPEN** | |

# OWCRC8

## OWCRC8(len, cdata [, cdata...])

Processes the cdata and returns the 8 bit CRC

len - length of data to process

cdata - data to process

The cdata can be a string, array or a list of variables

See also:

| | | |
|---|---|---|
| **MM.OW** | **OWREAD** | **OWSEARCH** |
| **OWCRC16** | **OWRESET** | **OWWRITE** |

# OWCRC16

## OWCRC16(len, cdata [, cdata...])

Processes the cdata and returns the 16 bit CRC

len - length of data to process

cdata - data to process

The cdata can be a string, array or a list of variables

See also:

| | | |
|---|---|---|
| **MM.OW** | **OWREAD** | **OWSEARCH** |
| **OWCRC8** | **OWRESET** | **OWWRITE** |

# OWREAD

## OWREAD pin, flag, length, data [, data...]

pin - the MMBasic I/O pin to use

flag - a combination of the following options:

1 - send reset before command

2 - send reset after command

4 - only send/recv a bit instead of a byte of data

8 - invoke a strong pullup after the command (the pin will be set high and open drain disabled)

length - length of data to send or receive

data - data to send or receive

After the command is executed, the pin will be set to the not configured state unless flag option 8 is used. The data and ser arguments can be a string, array or a list of variables.

The OWRESET and OWSEARCH commands (and the OWREAD and OWWRITE commands if a reset is requested) set the MM.OW variable. to 1 = OK (presence detected, search successful) or 0 = Fail (presence not detected, search unsuccessful).

```
' Read data/temperature command
' Pin 20, send reset first, send two bytes
' &hCC - Skip ROM
' &hBE - read data command
OWWRITE 20, 1, 2, &hCC, &hBE

' Read data/temperature from DS1822
' Pin 20, send reset after, read two bytes
OWREAD 20, 2, 2, LowTemp, HighTemp

' Combine and adjust according to data sheet
Cel = ((HighTemp And &b111) * 256 + LowTemp) / 16

' Print Celsius and convert/print Fahrenheit
Print "Celsius" Cel
Print "Fahrenheit" ( 9 / 5) * Cel + 32;
```

See also:

| | | |
|---|---|---|
| **MM.OW** | **OWRESET** | **SETPIN** |
| **OWCRC8** | **OWSEARCH** | |
| **OWCRC16** | **OWWRITE** | |

# OWRESET

## OWRESET pin [,presence]

pin - the MMBasic I/O pin to use

presence - an optional variable to receive the presence pulse (1 = device response, 0 = no device)

The OWRESET and OWSEARCH commands (and the OWREAD and OWWRITE commands if a reset is requested) set the MM.OW variable. to 1 = OK (presence detected, search successful) or 0 = Fail (presence not detected, search unsuccessful).

```
' Reset the 1w bus on I/O Pin 20
OWRESET 20
```

See also:

**MM.OW** **OWREAD** **SETPIN**

**OWCRC8** **OWSEARCH**

**OWCRC16** **OWWRITE**

# OWSEARCH

## OWSEARCH pin, srchflag, ser [,ser...]

pin - the MMBasic I/O pin to use

srchflag - a combination of the following options:

1 - start a new search

2 - only return devices in alarm state

4 - search for devices in the requested family (first byte of ser)

8 - skip the current device family and return the next device

16 - verify that the device with the serial number in ser is available

If srchflag = 0 (or 2) then the search will return the next device found

ser - serial number (8 bytes) will be returned (srchflag 4 and 16 will also use the values in ser)

The OWRESET and OWSEARCH commands (and the OWREAD and OWWRITE commands if a reset is requested) set the MM.OW variable. to 1 = OK (presence detected, search successful) or 0 = Fail (presence not detected, search unsuccessful).

See also:

**MM.OW** **OWREAD** **SETPIN**

**OWCRC8** **OWRESET**

**OWCRC16** **OWWRITE**

# OWWRITE

## OWWRITE pin, flag, length, data [, data...]

pin - the MMBasic I/O pin to use

flag - a combination of the following options:

1 - send reset before command

2 - send reset after command

4 - only send/recv a bit instead of a byte of data

8 - invoke a strong pullup after the command (the pin will be set high and open drain disabled)

length - length of data to send or receive

data - data to send or receive

After the command is executed, the pin will be set to the not configured state unless flag option 8 is used. The data and ser arguments can be a string, array or a list of variables.

The OWRESET and OWSEARCH commands (and the OWREAD and OWWRITE commands if a reset is requested) set the MM.OW variable. to 1 = OK (presence detected, search successful) or 0 = Fail (presence not detected, search unsuccessful).

```
' Start temperature conversion on DS1822 device
' Pin 20, send reset first, send two bytes
' &hCC - Skip ROM
' &h44 - Start conversion
OWWRITE 20, 1, 2, &hCC, &h44
```

See also:

| | | |
|---|---|---|
| **MM.OW** | **OWREAD** | **SETPIN** |
| **OWCRC8** | **OWRESET** | |
| **OWCRC16** | **OWSEARCH** | |

# PEEK

## PEEK( hiword, loword )

## PEEK( keyword, ±offset)

Will return a byte within the PIC32 virtual memory space.

The address is specifies by ‘hiword’ which is the top 16 bits of the address while ‘loword’ is the bottom 16 bits.

Alternatively 'keyword' can be used and 'offset' is the ±offset from the address of the keyword. The keyword can be VIDEO (CGMMSTICK video buffer) or RVIDEO, GVIDEO, BVIDEO (red, blue and green video buffers on the CGCOLORMAX), PROGMEM (program memory) or VARTBL (the variable table). The input keystroke buffer is KBUF, the position of the head of the keystroke queue is KHEAD and the tail is KTAIL (the buffer is 256 bytes long).

See the POKE command for notes and warnings related to memory access.

See also:

**POKE**

# PIN (command/function)

## PIN( pin ) = value

For a 'pin' configured as digital output this will set the output to low ('value' is zero) or high ('value' 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.

'pin' zero is a special case and will always control the LED on the front panel. A 'value' of non zero will turn the LED on, or zero for off.

## PIN( pin )

PIN returns the value on the external I/O 'pin'. Zero means digital low, 1 means digital high and for analog inputs it will return the measured voltage as a floating point number.

Frequency inputs will return the frequency in Hz (maximum 200KHz). A period input will return the period in milliseconds while a 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' zero is a special case which will always return the state of the bootload push button on the PC board (non zero means that the button is down).

Also see the SETPIN command.

See also:

**PORT** **PWM** **SETPIN**

**PULSE** **PWM STOP**

# PLAYMOD

## PLAYMOD file [, dur]

Play synthesised music or sound effects.  'file' specifies a file which must be located on the internal drive A: and must be in the .MOD format.

'dur' specifies the duration in milliseconds that the audio will play for - if not specified it will play until explicitly stopped or the program terminates.

The audio is synthesized in the background and is not disturbed by the running program.

The command PLAYMOD STOP will immediately halt any music or sound effect that is currently playing.

NOTE:  The file needs to be located on the internal drive A: for performance reasons, it will not play from the SD card.

See also:

**PLAYMOD STOP** **TONE**

**SOUND** **TONE STOP**

## PLAYMOD STOP

The command PLAYMOD STOP will immediately halt any music or sound effect that is currently playing.

See also:

**PLAYMOD** **TONE**

**SOUND** **TONE STOP**

# POKE

## POKE hiword, loword, val

## POKE keyword, offset, val

Will set a byte within the PIC32 virtual memory space.

The address is specified by ‘hiword’ which is the top 16 bits of the address while ‘loword’ is the bottom 16 bits.

Alternatively 'keyword' can be used and 'offset' is the ±offset from the address of the keyword. The keyword can be VIDEO ( CGMMSTICK video buffer) or RVIDEO, GVIDEO, BVIDEO (red, blue and green video buffers on the CGCOLORMAX), PROGMEM (program memory) or VARTBL (the variable table). The input keystroke buffer is KBUF, the position of the head of the keystroke queue is KHEAD and the tail is KTAIL (the buffer is 256 bytes long).

This command is for expert users only. The PIC32 maps all control registers, flash (program) memory and volatile (RAM) memory into a single address space so there is no need for INP or OUT commands.

The PIC32MX5XX/6XX/7XX Family Data Sheet lists the details of this address space.

Note that MMBasic stores most data (including video) as 32 bit integers and the PIC32 uses little endian format.

WARNING: No validation of the parameters is made and if you use this facility to access an invalid memory address you will get an "internal error" which causes the processor to reset and clear all memory.

See also:

**PEEK**

# PORT (command/function)

## PORT(start, nbr) = value

Set a number of I/O consecutive pins simultaneously (ie, with one command).

'start' is an I/O pin number and the lowest bit in 'value' (bit 0) will be used to set that pin. Bit 1 will be used to set the pin 'start' plus 1, bit 2 will set pin 'start'+2 and so on for 'nbr' number of bits. The I/O pins used must be numbered consecutively and configured as outputs before this command is used.

For example;

```
PORT(4, 8) = &B10000011
```

will set eight consecutive I/O pins starting with pin 4. Pins 4, 5 and 12 will be set high while 6 to 11 will be set to a low.

## PORT(start, nbr)

Returns the value of a number of consecutive I/O pins in one operation.

'start' is an I/O pin number and its value will be returned as bit 0. 'start'+1 will be returned as bit 1, 'start'+2 will be returned as bit 2, and so on for 'nbr' number of bits. The I/O pins used must be numbered consecutively and configured as inputs before this command is used.

This command can be used to conveniently communicate with parallel devices like LCD displays. Any number of I/O pins (and therefore bits) can be used from 1 pin up to 23 pins.

See also:

**PIN** **PWM STOP**

**PWM** **SETPIN**

# PULSE

## PULSE pin, width

Will generate a pulse on 'pin' with duration of 'width' mS.

'width' can be a fraction.  For example, 0.01 is equal to 10 µS This enables the generation of very narrow pulses.

The generated pulse is of the opposite polarity to the state of the I/O pin when the command is executed.  For example, if the output is set high the PULSE command will generate a negative going pulse.

Notes:  For a pulse of less than 3 mS the accuracy is ± 1 µS. For a pulse of 3 mS or more the accuracy is ± 0.5 mS. A pulse of 3 mS or more will run in the background. 'pin' must be configured as an output.

See also:

**PIN** **PWM STOP**

**PWM** **SETPIN**

## PWM freq, ch1, ch2

Generate a pulse width modulated (PWM) output for driving analogue circuits.

'freq' is the output frequency (between 20 Hz and 1 MHz) . The frequency can be changed at any time by issuing a new PWM command.

The output will run continuously in the background while the program is running and can be stopped using the PWM STOP command.

'ch1' and 'ch2' are the output duty cycles for channel 1 and 2 as a percentage. If the percentage 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. Both are optional and if not specified will default to the previously used duty cycle for that channel.

The PWM output is generated on the PWM/sound connector and that assumes that the connector has been wired for PWM output. The frequency of the output is locked to the PIC32 crystal and is very accurate and for frequencies below 100 KHz the duty cycle will be accurate to 0.1%.

The original monochrome Maximite has only one PWM/sound output so only 'ch1' can be set on that model.

See also:

**PIN** **PWM STOP**

**PULSE** **SETPIN**

## PWM STOP

Stop PWM output.

See also:

**PIN** **PWM**

**PULSE** **SETPIN**

# SETPIN

## SETPIN pin, cfg [,target]

Will configure the external I/O 'pin' according to 'cfg':

The original Maximite has 20 I/O pins numbered 1 to 20, the Color Maximite adds another 20 I/O pins on the Arduino connector. These are labeled D0 to D13 and A0 to A5.

0 Not configured or inactive

1 Analog input (pins 1 to 10, A0 to A5)

2 Digital input (all pins and 5V tolerant on pins 11 to 20)

3 Frequency input (pins 11 to 14)

4 Period input (pins 11 to 14)

5 Counting input (pins 11 to 14)

6 Interrupt on low to high input change (pins 1 to 20, D0 to D8) for target

7 Interrupt on high to low input change (pins 1 to 20, D0 to D8) for target

8 Digital output (all pins)

9 Open collector digital output to 5V (pins 11 to 20, D0 to D13) In this mode the function PIN() will also return the output value.

The starting line number of the interrupt routine is specified in the third parameter 'target'. This mode also configures the pin as a digital input so the value of the pin can always be retrieved using the function PIN().

See also IRETURN to return from the interrupt.

See also:

**IRETURN** **PORT** **PWM**

**PIN** **PULSE** **PWM STOP**

## SOUND freq [, dur]

Generate a single tone of ‘freq’ (between 20Hz and 1MHz) for ‘dur’ milliseconds. The sound is played in the background and does not stop program execution.

If 'dur' is not specified the sound will play forever until turned off. If ‘dur’ is zero, any active SOUND statement is turned off.

The command has been replaced with the TONE command that generates a pure sine wave (not a square wave).

See also:

**PLAYMOD** **TONE**

**PLAYMOD STOP** **TONE STOP**

# SPI

## SPI( rx, tx, clk[, data[, speed[, mode[, bits]]]] )

Sends and receives a byte using the SPI protocol with MMBasic as the master (ie, it generates the clock).

'rx' is the pin number for the data input (MISO)

'tx' is the pin number for the data output (MOSI)

'clk' is the pin number for the clock generated by MMBasic (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.

'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.

'mode' is optional and is a single numeric digit representing the transmission mode. The default mode is 3.

'bits' is optional and represents the number of bits to send/receive. Range is 1 to 23 (this limit is defined by how many bits can be stored in a floating point number). The default is 8.

Mode CPOL CPHA Description

Mode: 0 (Clock polarity: 0 Clock phase: 0) Clock is active high, data is captured on the rising edge and output on the falling edge

Mode: 1 (Clock polarity: 0 Clock phase: 1) Clock is active high, data is captured on the falling edge and output on the rising edge

Mode: 2 (Clock polarity: 1 Clock phase: 0) Clock is active low, data is captured on the falling edge and output on the rising edge

Mode: 3 (Clock polarity: 1 Clock phase: 1) Clock is active low, data is captured on the rising edge and output on the falling edge

See also:

**SETPIN**

# TONE

## TONE left [, right [, dur]]

Generates a continuous sine wave on the sound output. 'left' and 'right' are the frequencies to use for the left and right channels. The tone plays in the background (the program will continue running after this command) and 'dur' specifies the number of milliseconds that the tone will sound for. If the duration is not specified the tone will continue until explicitly stopped or the program terminates.

The command TONE STOP will immediately halt the tone output.

The frequency can be from 1Hz to 20KHz and is very accurate (it is based on the PIC32 crystal oscillator). The frequency can be changed at any time by issuing a new TONE command.

In the monochrome Maximite and compatibles only the left frequency will play but a dummy or empty value is still required for the right channel.

See also:

**PLAYMOD** **SOUND**

**PLAYMOD STOP** **TONE STOP**

## TONE STOP

The command TONE STOP will immediately halt the tone output.

See also:

**PLAYMOD** **SOUND**

**PLAYMOD STOP** **TONE**