MMBasic code-pack to Read and Write Winbond Flash Memories

Winbond flash memories provide an excellent method of storing up to 16MB of data in a tiny SPI driven chip. At around $0.50 each (varies) these offer an amazing storage/price density and are used in many devices to store configuration and firmware. Throwing out some old electronics? Check the board - there is likely a Flash memory that can be recovered and re-used.

SPI is very much faster and easier to drive than I^{2}C. The routines here work flawlessly at 20MHz and were inspired by work MatherP published on the BackShed here https://www.thebackshed.com/forum/ViewTopic.php?TID=8492

That work used CSubs to access the Flash and whereas they are very quick, they exclude users of the bigger 'Mite platforms - the MMX and ST's Arm base processors - the H7/F4 platforms. The routines below are a purely MMBasic solution and so can be used on any MMBasic platform supporting the SPI bus.

The approach with the code here is comparable to a sequential file. Data is written using one function and read back with another. When data is exhausted a marker, CHR$(255), is reached. This is analagous with the EOF character of disk filing systems. This code provides a simple system of strings in, strings out. It handles just this - what you actually store is your own decision. Using a structured logging method e.g. Ultra_Compact Logging with Flash Storage on small MicroMites it would be possible to log continuously at one second intervals for eight days straight on a 4MB chip. With arbitrary strings of average length 50 bytes, in excess of 83,000 lines could be stored. `During research for this software pack, some solutions were found where a single string occupies an entire page with corresponding wastage or “slack space”. If such a method were employed, a 1MB flash could store just 4096 strings(!).`

windbond flash

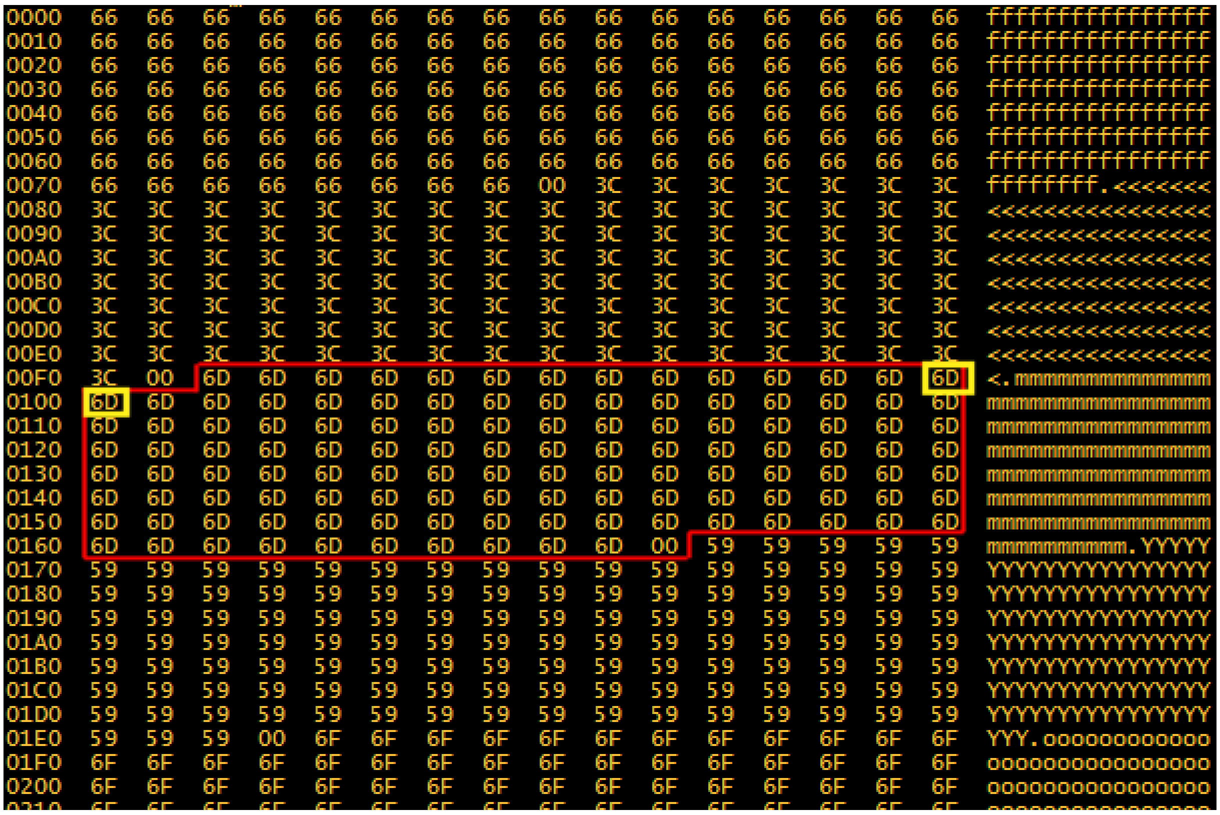
Organisation of Strings in Memory

There are generally two methods of storing strings. Both methods are length+1 and both have pros and cons. The first stores a byte giving the length of the string followed by the payload (the actual characters of the string) - this is how MMBasic stores strings in memory. The second is to store the payload and add a zero byte (00H) to show the end of the string - this is sometimes called C-style or null-terminated as this is how the C programming language generally works with strings. The method chosen in this pack is the latter. The reason for this is that there is a clear demarcation between strings which lends itself to random reading. With C-style strings, you can simply drop into the middle of a block of data, then read forwards or backwards until you get a zero byte, knowing that is the last byte of a string. With the first method, you would have to start at the beginning of memory each time and step through, adding up all the length bytes to work out where any given string begins. In some ways, the above mentioned method, of a single string per page, provides the best of both of these methods but I just can't rationalise the waste of space.

Winbond Flash technology provides continuous reading starting from a single 24 bit address. Once latched, the address increments with each read and so the entire memory could be rapidly read out with only a single address setup - at 20MHz this could be little more than 400nS per byte!

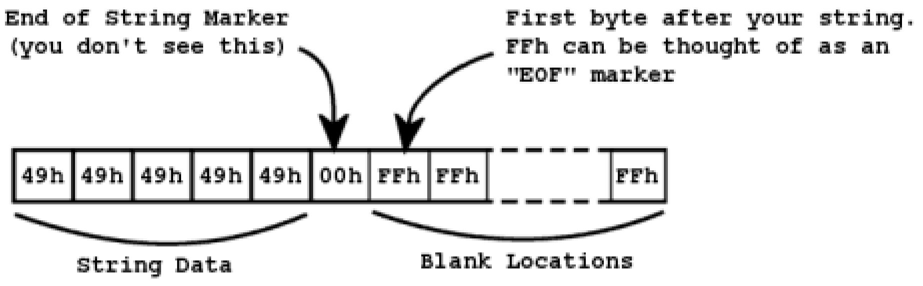
Writing the device is more involved. Internally, Flash memory is arranged into pages of 256 bytes. When writing, you can only do so within a given page and although you have to latch a full 24 bit address, the top 16 bits are fixed so only the bottom 8 bits can be worked with - you are locked into a page. Attempts to write beyond simply roll the bottom eight bits of the address back to zero and overwrite what was already in that page. Note also that because of the way flash memory is constructed, only zero bits can be written. You cannot set a previously zero bit to one without formatting. This is why a formatted chip has all locations sets to 255 (0FFh). When writing a byte to any given location, the result stored is a logical AND of the original value and the value you wrote. So supposing you wish to write 55h: If the memory is 0FFh, FF AND 55 = 55… all good, but now suppose you want to write 0AAh in that same location afterwards; 55 AND AA = 0. So you can see the data becomes unreliable unless you are writing in formatted cells. Any Random access writing you might try needs to be aware of this. Grey beards may remember burning EPROMS years back. It was the same approach there and fast programmers would read out an EPROM then AND it with your target code to see if it could be burned in without a time-consuming erase of the EPROM beforehand. Similar thing really.

The write routine takes care of paging making the process transparent to the user. You can see from the grab below that a string (red box) traverses a page boundary with no problem - the two bytes in the yellow boxes are the last and first bytes in different pages (00FFh = page 0 and 0100h = page 1) but it's all handled by the write function. This means the entire memory is available for use.



```
0000  66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66  ffffffffffffffff
0010  66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66  ffffffffffffffff
0020  66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66  ffffffffffffffff
0030  66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66  ffffffffffffffff
0040  66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66  ffffffffffffffff
0050  66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66  ffffffffffffffff
0060  66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66  ffffffffffffffff
0070  66 66 66 66 66 66 66 66 00 3C 3C 3C 3C 3C 3C 3C  ffffffff.<<<<<<<
0080  3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C  <<<<<<<<<<<<<<<<
0090  3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C  <<<<<<<<<<<<<<<<
00A0  3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C  <<<<<<<<<<<<<<<<
00B0  3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C  <<<<<<<<<<<<<<<<
00C0  3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C  <<<<<<<<<<<<<<<<
00D0  3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C  <<<<<<<<<<<<<<<<
00E0  3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C 3C  <<<<<<<<<<<<<<<<
00F0  3C 00 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D  <.mmmmmmmmmmmmmm
0100  6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D  mmmmmmmmmmmmmmmm
0110  6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D  mmmmmmmmmmmmmmmm
0120  6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D  mmmmmmmmmmmmmmmm
0130  6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D  mmmmmmmmmmmmmmmm
0140  6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D  mmmmmmmmmmmmmmmm
0150  6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 6D  mmmmmmmmmmmmmmmm
0160  6D 6D 6D 6D 6D 6D 6D 6D 6D 6D 00 59 59 59 59 59  mmmmmmmmmm.YYYYY
0170  59 59 59 59 59 59 59 59 59 59 59 59 59 59 59 59  YYYYYYYYYYYYYYYY
0180  59 59 59 59 59 59 59 59 59 59 59 59 59 59 59 59  YYYYYYYYYYYYYYYY
0190  59 59 59 59 59 59 59 59 59 59 59 59 59 59 59 59  YYYYYYYYYYYYYYYY
01A0  59 59 59 59 59 59 59 59 59 59 59 59 59 59 59 59  YYYYYYYYYYYYYYYY
01B0  59 59 59 59 59 59 59 59 59 59 59 59 59 59 59 59  YYYYYYYYYYYYYYYY
01C0  59 59 59 59 59 59 59 59 59 59 59 59 59 59 59 59  YYYYYYYYYYYYYYYY
01D0  59 59 59 59 59 59 59 59 59 59 59 59 59 59 59 59  YYYYYYYYYYYYYYYY
01E0  59 59 59 00 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F  YYY.oooooooooooo
01F0  6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F  oooooooooooooooo
0200  6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F  oooooooooooooooo
```

The below diagram shows the structure of strings in memory in more detail - each cell represents a byte in Flash. Here the string “IIIII” is stored as the first and only string in the memory



Strings may be 0 to 255 characters long and different length strings can be freely mixed in memory but must conform to the following rules for any one character:

- CHR$(0) - Forbidden. This is the string delimiter byte used for organising the stored strings.</li>
- CHR$(1) to CHR$(254) - May be freely used within your strings.</li>
- CHR$(255) - Forbidden. All Flash memory locations are set to 255 during a format. The location of the start of blank memory (where data may be safely stored) is determined from the first such byte. </li>

The initialization routine (WB.Init) performs several useful and vital functions:

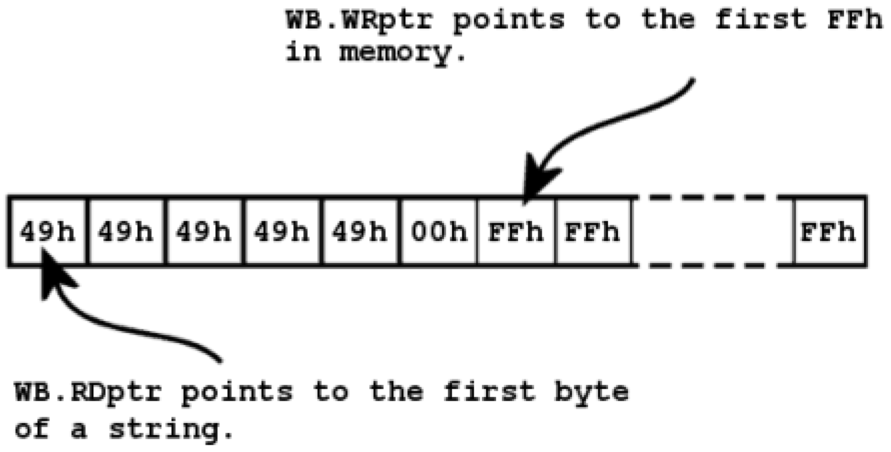
- It identifies the type of Flash attached (if any) and sets the maximum address variable WB.Top to the last available memory location.</li>
- It rapidly determines the first unused byte in the memory (more on this later) and sets the WB.WRptr variable so as not to trample on previously stored data</li>

Method

There are two address pointers, one for read and another for write. This permits an easy access method where the memory can be read and written simultaneously with no special measures. In normal use, the pointers are of no concern and any program will use the WB.WRstr() and WB.RDstr$() routines.

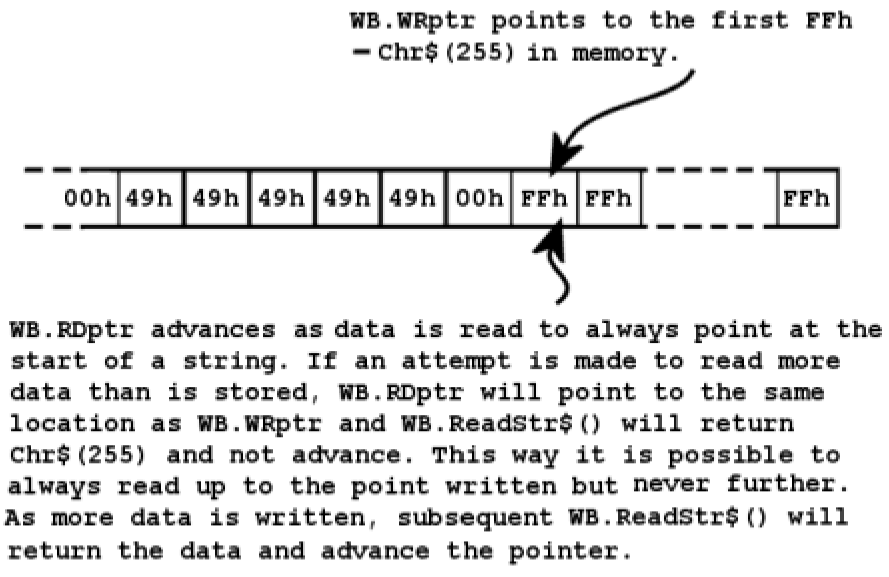
Writing: An erased Flash has all locations set to 255 so it follows the first such byte (lowest address) is the first place we can write data without trampling on earlier data (now it is clear why CHR$(255) is prohibited in strings). NOTE: Brand new Flash chips are not necessarily in an erased state and should be formatted first for reliable use - unless the garbage is of some curiosity value.

WB.WRptr points to the first such memory location i.e. the position new data can be reliably stored. Flash memories retain data for up to 20 years and with such large capacities, it is highly likely that data will be aggregated over power-cycles. At initialization, WB.WRPtr is ascertained by a rapid successive approximation algorithm that will return the first 255 byte in around 80mS on a 4MB chip with 48MHz MicroMite. This happens each time WB.Init is called - you can do this as often as you like but once at the start of your program is usual.


WB.WRptr points to the first FFh in memory.
49h
49h
49h
49h
49h
00h
FFh
FFh
FFh
WB.RDptr points to the first byte of a string.

Reading: Each byte is read from memory and built up in a string until a 0 is read whence the string is returned (the 0 does not form part of that string). The read pointer, WB.RDptr, is incremented to keep track of where reading occurs and will always point to the start of the next string to be read, thus you can continually read from the memory with successive calls to WB.ReadStr$. If an attempt is made to read past valid data, CHR$(255) is returned by WB.ReadStr$ and WB.RDptr is not incremented (because no string was read). When additional data is written, the next read will return the new data and so on…

Thus, WB.RDptr will happily trot along behind WB.WRPtr but never pass it. It is entirely normal for the two pointers to have the same value - when you have read out all the data. This can be used as an end-of-file (EOF) indication if you prefer.


WB.WRptr points to the first FFh
- Chr$(255) in memory.
00h
49h
49h
49h
49h
49h
00h
FFh
FFh
FFh
WB.RDptr advances as data is read to always point at the start of a string. If an attempt is made to read more data than is stored, WB.RDptr will point to the same location as WB.WRptr and WB.ReadStr$() will return Chr$(255) and not advance. This way it is possible to always read up to the point written but never further. As more data is written, subsequent WB.ReadStr$() will return the data and advance the pointer.

Global Variables

Integer **WB.RDptr**
The pointer for the next string READ operation

Integer **WB.WRptr**
The pointer for the next string WRITE operation

Integer **WB.ID**
The JEDEC identifier of the Flash - the manufacturer and 2 byte device ID.

String **WB$**
A human friendly version of the ID e.g. W25Q32 etc.

Integer **WB.Top**
The top-most available memory location

The Routines

WB.Init

Discover what type of Flash and Initialise the internal pointers.

WB.Format

Erase the entire Flash; sets all memory locations to contain 0FFh - does an initialization afterwards to reset all the pointers. New Flash chips should be formatted before use as they may contain random data that may confuse WB.Init and return dubious usage figures.

WB.Stat1RD()

WB.Stat2RD()

WB.Stat3RD()

Read the three status registers.

WB.WaitBusy

Wait for the Flash to become idle. Beware, a format can take several seconds depending on the Flash capacity. If you are using WATCHDOG in your code, you may need to put one in here so your code doesn't restart on long waits.

WB.TestBusy()

Check if the Flash is busy without waiting.

WB.Peek(address)

Peek a byte at the given address in Flash. Flash is erased to FF so when we peek the next address after a string we can tell if we are at EOF.

WB.ReadStr$()

Read a string from the current read pointer and increment the pointer. Returns Chr$(255) if we are at the end of the data.

WB.WriteStr(a$)

Write a string to the current write pointer and increment the pointer. Returns a zero if successful and 1 if the write would exceed available free space.

Support routines - unlikely to be useful in your code:

WB.Addr(address)

Latch a three byte address in the Flash.

WB.WREnable(0|1)

Enable/Disable write function.

WB.SetPage

Set up the page address for a pending write operation.

Example

```
'*****************************************************************
' your setup code goes  here
'*****************************************************************
Init:
    CPU 48
    'Option Autorun On
    'Option Baudrate 9600
    Option Explicit
    Option Base 0

    Const Ver$="0.20 of 24-NOV-2019"

'-------------------------------------------------------------------
'The only config you should need to do:
'change the following to the CS pin for the Flash memory
    Const WB.CS=23
'change the following to the SPI channel where your Flash chip is (it's
usually OK to leave it open
'so long as nothing else is competing and closes it after use - e.g. LCD
panels)
    SPI Open 20000000,0,8'20MHz, mode 0, 8 bits... seems happy at 20MHz but
drop this down if you get probs
'-------------------------------------------------------------------



'mandatory variables in your program:
'-------------------------------------------------------------------
    Dim Integer WB.RDptr,WB.WRptr,WB.Top,WB.ID,x,n
    Dim WB$
'-------------------------------------------------------------------
```

```
'*****************************************************************
' your program goes  here
'*****************************************************************



'A little demonstration code...

    x=WB.Init()    'single function id's the device, sets mem size and finds
first usable address
            'your code should check WB.Top after this call. if no Flash
found or couldn't be identified, WB.Top=-1
    If x<>0 Then
        Print"Flash chip not recognized: ";Hex$(x,6)
        End
    EndIf

    Dim y$
    Dim Integer m

    Print"Found ";WB$;" (";Hex$(WB.ID,6);")"
    Print"memory size is";(WB.Top+1)/1024;"KB (0-"+Hex$(WB.Top)+")"
    Print"First blank address is ";Hex$(WB.WRptr);"h"

    Input"Format before testing (Y/N)?",y$
    If y$="y" then
        Print "Formatting..."
        WB.Format
        Print"First blank address is ";Hex$(WB.WRptr);"h"
    EndIf

    Randomize Timer ' delete this line on MMX/Arm etc

'write some random strings to Flash with timings
    Print
    Print"","WR Time"
    Print"---------------------------------------"

    For n=1 To 10
        Print n,
        Timer=0:x=WB.WriteStr(Date$+" "+Time$+"
"+String$(65,48+Rnd*70)):Print Timer;"mS"
    Next

'read back the strings - demonstrates chr$(255) if attempting to read
    Print:Print"Reading back all strings..."
    Print"Addr","Time","Data"
    Print"---------------------------------------"
    Do
        Print Hex$(WB.RDptr,4);"h",
        Timer=0:y$=WB.ReadStr$():Print Timer,y$
    Loop While y$<>Chr$(255)

    For n=0 To 1023 Step 16
        y$=""
        Print Hex$(n,4);"  ";
        For m=0 To 15
            x=WB.Peek(n+m)
            Print Hex$(x,2);"  ";
            If x<32 Or x>126 Then x=&h2e
            y$=y$+Chr$(x)
        Next
        Print y$
    Next

    Input"Format the Flash (Y/N)?",y$
    y$=UCase$(y$)
    If y$="Y" Then
        Print"wait a moment"
        timer=0:WB.Format:Print timer;"mS"
    EndIf

    End
```

The routines are fairly brisk and numerous testing/adjustment phases have resulted in some tricks to make them as fast as possible. The Write function was a bottle-neck and has been re-written. It is now very quick and because of the SPI Bulk Write methods available, paradoxically, writes substantially out-perform reads by about a factor of 3 worst case (255 character strings) at ~20mS Vs ~70mS. With 3 character strings, Reads take around 5mS and Writes are still around the 20mS mark.

Here are some timings for sequential reads & writes on a 48MHz '170 with 24MHz SPI bus.

Random 254 char strings (worst case)

```
old write method:
<hr>
 1       89mS
 2       97mS
 3       97mS
 4       96mS
 5       97mS
 6       96mS
 7       96mS
 8       97mS
 9       96mS
 10      97mS
```

```
new write method:
<hr>
 1       11mS
 2       19mS
 3       19mS
 4       19mS
 5       20mS
 6       19mS
 7       19mS
 8       19mS
 9       19mS
 10      20mS
```

Read 254 character string

```
<hr>
    73mS
    72mS
    72mS
    71mS
    72mS
    72mS
    72mS
    72mS
```

Read 3 character string

```
<hr>
    5mS
    5mS
    4mS
    5mS
    5mS
    5mS
    5mS
    4mS
    5mS
```

The Code

```
'*****************************************************************
'Flash Subs & Functions

'Discover the attached flash memory and initialize all the pointers and
stuff
'sets up the CS pin and opens the SPI channel
'returns 0 if the chip is IDed with all the variables correctly set up.
'else returns the ID read from the chip = &FFFFFF is open bus (no chip)
    Function WB.Init() As Integer
```

```
        Local Integer n
        WB.Top=-1:WB.WRptr=-1
        'identify the Flash ram by reading the JEDEC ID and setting some key
values
        Pin(WB.CS)=0
            x=SPI(&h9F)
            WB.ID=65536*SPI(0)+256*SPI(0)+SPI(0)
        Pin(WB.CS)=1
        'if your device is not listed or comes up unknown, you'll have to
determine the correct ID
        'from the device PDF and add it below. Good news is, it looks like
all the Winbond W25* are
        '4KB sectors and 256B page size so the code *should* work without
any changes - no guarantees
        'only known problem is the code assumes a 24 bit address which
breaks for >16MB devices and maybe
        'others which might only provide a 16 bit address - haven't looked
at the PDF.
        'The Winbond product selector shows these chips but I couldn't find
PDFs for them:
        'W25Q02JV    256MB
        'W25Q01JV    128MB

        Select Case WB.ID
            '*** Tested
            Case &hEF4018:WB.Top=16384:WB$="W25Q128"'16MB
            Case &hEF4017:WB.Top=8192:WB$="W25Q64"'8MB
            Case &hEF4016:WB.Top=4096:WB$="W25Q32"'4MB
            Case &hEF4015:WB.Top=2048:WB$="W25Q16"'2MB
            Case &hEF4014:WB.Top=1024:WB$="W25Q80"'1MB
            '*** UnTested
            Case &hEF4020:WB.Top=65536:WB.Top=16384:WB$="W25Q512"'64MB -
With these two devices, the code should work as they default to
            Case &hEF4019:WB.Top=32768:WB.Top=16384:WB$="W25Q256"'32MB - 3
byte addresses but you'll only be able to use the bottom 16MB.
                                        'May support the higher capacities
later if there is a demand.
            Case &hEF4013:WB.Top=512:WB$="W25Q40"'512KB
            Case &hEF5012:WB.Top=256:WB$="W25Q20"'256KB
            Case &hEF6011:WB.Top=128:WB$="W25Q10"'128KB
            Case &hEF3010:WB.Top=64:WB$="W25X05"'64KB
            'Case &hBF2642:WB.Top=4096:WB$="SST26VF032"'Microchip 4MB pin
and code compatible - needs work on global enable
            Case Else
                SPI Close:WB.Top=-1:WB.Init=WB.ID:Exit Function'can't
identify
        End Select

        WB.Top=(WB.Top*1024)-1'set the top memory location
```

```
'find the first (=lowest address with) FF byte
        If WB.Peek(0)=255 Then WB.WRptr=0:WB.RDptr=0:Exit Function'blank
Flash; early bath
'otherwise find by successive approximation - very fast, searches 4MB in
80mS
        WB.WRptr=(WB.Top+1)\2:n=WB.WRptr\2'start at the middle and go in
smaller and smaller halves
        Do
            If WB.Peek(WB.WRptr)=255 Then
                WB.WRptr=WB.WRptr-n'still in void so go down by half the
remainder
            Else
                WB.WRptr=WB.WRptr+n'we are in the strings so go up by half
the remainder
            EndIf
            If n<>1 Then n=n\2'smaller and smaller halves
        Loop Until WB.Peek(WB.WRptr)=255 And WB.Peek(WB.WRptr-1)<>255'is the
byte before the FF !FF? if so, we are done
    End Function

'set or clear flash write enable flag
    Sub WB.WREnable(a As Integer)
        If WB.Top=-1 Then Exit Sub
        Local Integer x
        Pin(WB.CS)=0
            If a=0 Then
                x=SPI(4)
            Else
                x=SPI(6)
            EndIf
        Pin(WB.CS)=1
        WB.WaitBusy
    End Sub

'wait while Flash is busy
    Sub WB.WaitBusy
        If WB.Top=-1 Then Exit Sub
        Do While WB.TestBusy()
        'you might want to put a WATCHDOG here. Long operations could break
your program
        Loop
    End Sub

'test BUSY flag in STAT1
    Function WB.TestBusy() As Integer
        If WB.Top=-1 Then Exit Function
        WB.TestBusy=(WB.Stat1RD() And 1)
    End Function

    Function WB.Stat1RD() As Integer
        If WB.Top=-1 Then Exit Function
        Local Integer x
        Pin(WB.CS)=0
            x=SPI(5)
            WB.Stat1RD=SPI(0)
        Pin(WB.CS)=1
    End Function

    Function WB.Stat2RD() As Integer
        If WB.Top=-1 Then Exit Function
        Local Integer x
        Pin(WB.CS)=0
            x=SPI(&h35)
            WB.Stat2RD=SPI(0)
        Pin(WB.CS)=1
    End Function

    Function WB.Stat3RD() As Integer
        If WB.Top=-1 Then Exit Function
        Local Integer x
        Pin(WB.CS)=0
            x=SPI(&h15)
            WB.Stat3RD=SPI(0)
        Pin(WB.CS)=1
    End Function

'send a 3 byte address to Flash
    Sub WB.Addr(a As Integer)
        If WB.Top=-1 Then Exit Sub
        Local Integer x
        x=SPI((a>>16) And 255)
        x=SPI((a>>8) And 255)
        x=SPI(a And 255)
    End Sub

'set the address for page writing
    Sub WB.SetPage
        If WB.Top=-1 Then Exit Sub
        Local Integer x
        Pin(WB.CS)=1
        WB.WaitBusy
        WB.WREnable 1
        WB.WaitBusy
        Pin(WB.CS)=0
        x=SPI(2)
        WB.Addr WB.WRptr'setup the address
    End Sub

'erase the entire chip - Beware; can take several seconds
    Sub WB.Format()
        If WB.Top=-1 Then Exit Sub
        Local Integer x
        WB.WREnable 1
        WB.WaitBusy
        Pin(WB.CS)=0
            x=SPI(&h60)'start the erase - will take some time
        Pin(WB.CS)=1
        WB.WaitBusy
        x=WB.Init()
    End Sub

'Peek any address in Flash
    Function WB.Peek(a As Integer) As Integer
        If WB.Top=-1 Then Exit Function
        Local Integer x
        Pin(WB.CS)=0
            x=SPI(3)'read at the given address
            WB.Addr a
            WB.Peek=SPI(0)'grab one byte
        Pin(WB.CS)=1
    End Function

'Read a string from the Read Address using the global WB.RDptr variable
    Function WB.ReadStr$()
        If WB.Top=-1 Then Exit Function
        Local Integer x,z
        Local a$
        WB.WaitBusy'wait for the Flash to be idle
        Pin(WB.CS)=0
        x=SPI(3)
        WB.Addr WB.RDptr
        For z=1 to 256'stream the data from Flash into a$, 256 covers max
length+chr$(0), we bail early if needed
            x=SPI(0)'get the character
            Select case x
                Case 0 'break out on the delimiter
                    Poke Var a$,0,z-1
                    WB.RDptr=WB.RDptr+Len(a$)+1'bump the address along by
the number of bytes we read +1 for the delimiter
                    Exit For
                Case 255 'break out on EOF, should never happen
                    WB.RDptr=WB.RDptr+Len(a$)
                    a$=Chr$(x)'single char?
                    Exit For
                Case Else
                    Poke Var a$,z,x
            End select
        Next
        Pin(WB.CS)=1
        WB.ReadStr$=a$
    End Function
```

```
'write a string to the next available position. Return 0 if successful
    Function WB.WriteStr(a$) As Integer
        If WB.Top=-1 Then WB.WriteStr=1:Exit Function
        Local Integer n,x
        n=Len(a$)+1
        If WB.WRptr+n>WB.Top Then WB.WriteStr=1:Exit Function
        x=256-(WB.WRPtr Mod 256)' remaining space in this page
        WB.SetPage
        If n<=x Then 'will fit in current page
            SPI Write n-1,a$
            x=SPI(0)
            Pin(WB.CS)=1
            WB.WrPtr=WB.WrPtr+n
        Else 'have to split across pages
            'first half
            Local l$
            l$=Left$(a$,x)
            SPI Write len(l$),l$
            Pin(WB.CS)=1
            WB.WrPtr=WB.WrPtr+x
            'second half
            l$=Mid$(a$,x+1)+Chr$(0)
            x=Len(l$)
            WB.SetPage
            SPI Write x,l$
            Pin(WB.CS)=1
            WB.WrPtr=WB.WrPtr+x
        EndIf
    End Function
'*****************************************************************
```