HIDDEN BUT USEFUL TFBASIC FUNCTIONS

for the original TFX-11 only, not the TFX-11v2
Tuesday, March 15, 2005

Since the original TFX-11 was released, we have received requests from customers about how to perform certain tasks that were either not available in TFBASIC or were not documented. This document explains how to perform these tasks.

Tasks explained in this document

- Load a program into TFX-11 RAM and run the program
- Erase the datafile using serial port commands
- Off-load the datafile using serial port commands
- Flushing data to the EEPROM in a TFBASIC program
- Change the value of DFPNT in a TFBASIC program
- Erase the datafile from a TFBASIC program
- Read data from the datafile in a TFBASIC program
- Prevent corrupt datafiles after a power failure or reset

Load a program into TFX-11 RAM and run the program

Why you may want to do it

You may want to write your own program to load a TFBASIC program to the TFX-11. The program will have to have been compiled and saved in binary format. See "Additional notes" below.

How to do it

	Desk-top computer	TFBASIC action
1)	Send Ctrl-L character	Echoes Ctrl-L if ready
2)	Send byte specifying number of data bytes to follow (1 to 255)	no action
3)	Send the block of bytes	no action
4)	If more bytes to send, go to step 2	no action
5)	Send a zero byte to stop transfer	Sends CR/LF then '#'
6)	Send Ctrl-R character	Runs the program

Additional notes

This assumes you've used TFTools to create a compiled binary file of the program that will be loaded into the TFX-11. With the program loaded in an editor window, select the item "Compiler options" under the Tattletale menu in TFTools. When you get the Compiler Options dialog, select "Create binary file" and hit "OK". Then select "Syntax check" under the Tattletale menu. This will compile the program and store it in a file with a .bin extension.

There is no time-out on this protocol nor is there any error checking.

Erase the datafile using serial port commands

Make sure you understand the consequences
Normally, the parallel port is used to erase the datafile because this is part of the process of loading a new copy of the operating system and loading a program into the EEPROM. Another advantage of using the parallel port for erasing the datafile is that the parallel port is the only way to check a flag that is set when the datafile has been off-loaded. Also, the parallel port is the only way to set the TFBASIC read-only variable DFERASED. If you erase the datafile using the instructions, you will be able to neither set the DFERASED flag nor check if the datafile has been off-loaded. Taking this all into account, if you still need to off-load using the serial port, here is the protocol. The '#' prompt must be displayed by the TFX-11 before this protocol can begin.

How to do it

	Host computer action	TFBASIC action
1)	Send Ctrl-E	Echoes Ctrl-E
2)	Send FA hex (within one second	Erases datafile and Sets datafile pointer to 0
3)	no action	Sends '0' on no errors or sends '1' if error

Additional notes
The Tattletale actually sends a '0' (zero) character if the erase succeeds or a '1' (one) character if the erase fails.

To restart the program, once the '#' sign is showing again, just send a Ctrl-R character.

Off-load the datafile using serial port commands

Make sure you understand the consequences
Normally, the parallel port is used to off-load the datafile because it is over 6 times faster than off-loading the datafile serially at 38400 baud. Another advantage of using the parallel port for off-loading is that only the parallel port can be used to set a flag on the TFX-11 to indicate that the datafile has been off-loaded. If you attempt to erase this datafile using the parallel port, and this flag indicates that the datafile has not been off-loaded, you will get a warning. Taking this all into account, if you still need to off-load using the serial port, here is the protocol. The '#' prompt must be displayed by the TFX-11 before this protocol can begin.

How to do it

	Host computer action	TFBASIC action
1)	Ctrl-O	Echoes Ctrl-O
2)	Ctrl-T	Sends total size of datafile as 8 hex characters followed by CR/LF. Bytes are sent MSB first.
3)	Ctrl-N	Sends current datafile pointer as 8 hex characters followed by CR/LF. Sends MSB first.
4)	Send number of bytes to off-load as 8 hex char followed by CR/LF. Send MSB first.	
5)	no action	Ctrl-Z (query for Block 0)
6)	Ctrl-B - to request Block 0 or any other character for NO. There is a < 1sec time-out on this.	Echoes character
7)	Start XMODEM protocol	Responds to XMODEM

Additional notes
To restart the program, once the '#' sign is showing again, just send a Ctrl-R character.

Flush data to the EEPROM in a TFBASIC program

Why you may want to do it
When you Store to the datafile, you are not writing directly to the EEPROM. Instead, the bytes are written to a RAM buffer. Since the EEPROM is segmented into 32-byte blocks, the TFBASIC buffer is 32 bytes wide. When this buffer fills, the 32 bytes are automatically written to the EEPROM. The buffer is also flushed when the Stop command is executed or when a Ctrl-C is detected at the serial port (and is not re-vectored with CBREAK) or if a “How?” error occurs (and is not re-vectored with ONERR). Other than this, the data remains in the buffer which can make some people nervous. You may want to flush the EEPROM buffer before doing a long Sleep or Hyb. You may want to flush the EEPROM buffer before reading data from the datafile. You would certainly want to flush the buffer before you execute a Halt command.

How to do it
There is a user-accessible assembly routine at hex address FD88 that flushes the buffer. This function is only available in TFBASIC version 1.05 and above. The mnemonic that will be used in future versions of the manual will be FEESBF (flush EEPROM store buffer). You would use the Call command like this:

```
call &hFD88,0
```

That’s all there is to it.

Additional notes
You should be able to call this function as often as you want but be aware of two things:

1) this function can use up to 15 mA (depends on the amount of real data in the buffer)
2) this function can take up to 5 mSec (usually less than 500 µSec)

User-accessible assembly routine used in the example

FEESBF Writes the contents of the 32-byte datafile buffer to flash EEPROM

Address: FD88

Args: None
Return: None
Errors: Returns with Overflow Flag (in Condition Code Register) set if time-out accessing the EEPROM. Otherwise, Overflow Flag is clear.

Change the value of DFPNT (datafile pointer) in a TFBASIC program

Why you may want to do it
Normally, the datafile pointer (DFPNT) is only changed when:

1) it is cleared to zero at program launch (via the parallel port)
2) it is cleared to zero if the TFX-11 loses power
3) it is incremented by the STORE command

DFPNT is a read-only variable in TFBASIC. If the programmer wants to save a section of EEPROM for later use or if the datafile is erased programmatically (see an example of this later in this document), you would want to change the value of DFPNT in the TFBASIC program.

How to do it
DFPNT, the address that will receive the next data from STORE, is located at addresses B0, B1, B2 and B3 hex where the most significant byte is at B0 hex. Before changing the datafile address, though, you must flush the EEPROM buffer to the EEPROM (see the previous application note). Then the datafile pointer would be set by assigning the four-byte value to addresses B0 - B3 hex. Finally, the new datafile pointer must be registered with the operating system by calling the user-accessible assembly function SBKADR (at address FD52 hex). This clears the EEPROM buffer and addresses that page on the EEPROM.

Now, the next STORE instruction will store data starting at the new datafile address. The value of DFPNT will be incremented automatically. The functions FEESBF and SBKADR are only available in TFBASIC version 1.05 and above.

User-accessible assembly routine used in the example

SBKADR Sets up EEPROM store buffering system. Assumes the datafile storage pointer (addr B0 to B3) has been set to the correct value. Sets the address of the physical EEPROM and fills the EEPROM storage buffer with FF's.

Address: FD52

Notes: Because the FF's can safely be 'stored' over any pre-existing data in the datafile, this routine should work for any number in the datafile pointer. Power drain for the TFX-11 is higher when writing the EEPROM.

Args: None
Return: None
Errors: None

TFBASIC example for changing the datafile pointer

```
store 1,2,3,4      // store some data in the datafile
print dfpnt        // print the current datafile pointer
call &hFD88,0      // flush data in buffer to EEPROM (FEESBF)
                   // clear the datafile pointer
poke &hB0,0        // most significant byte
poke &hB1,0
poke &hB2,0
poke &hB3,0        // least significant byte
call &hFD52,0      // set up buffer and its address for further Stores (SBKADR)
print dfpnt        // print the new datafile pointer
```

Right-click paperclip icon and Open or Save example

Erase the datafile from a TFBASIC program

How to do it
The main function is EEErase. This does the actual erasing but you first need to select the EEPROM by executing SelEE. Then you have to enable the EEPROM for writing by calling EWEEPR. Then call EEErase to erase each 4K EEPROM block. After the last call to EEErase, you must disable the EEPROM from writing by calling DWEEPR. Then call SelNone to deselect the EEPROM. The functions used in this note are only available in TFBASIC version 1.05 and above.

Additional notes
Function EEErase requires a block number argument. The table starting at address 13B hex in the system area contains the block numbers of the flash EEPROM blocks used by the datafile. The table is terminated with a value of 127 (FF hex) because block 127 is never used by the datafile. It is VERY important to use this list to get block numbers because erasing the wrong block could erase system information, TFBASIC and/or any program you have loaded in EEPROM. Also, there may be bad blocks in the EEPROM and an erase operation will fail for these blocks. The list starting at 13B hex contains only good blocks that are part of the datafile.

This operation will not change the datafile pointer used for storing. The datafile pointer is located at addresses B0 to B3 and will need to be updated. Also, this new value will need to be registered and the RAM buffer prepared by calling the function SBKADR (address FD52H) explained later. The PIC coprocessor also saves a copy of the datafile pointer. You will need to store zero in all four bytes at B0 to B3 and then stop the program. When a program stops, the datafile pointer is copied out to the PIC.

Two important notes about the consequences of erasing the datafile yourself:
There is a read-only variable named DFERASED in TFBASIC that can only be set when the datafile is erased by a host computer using the parallel port. When you erase the datafile using the instructions here, you cannot set this flag. It won't adversely affect your program. It just takes away some useful information that your program can use. Also, there is a read-only variable that the TFTools program can read that specifies that the datafile has been off-loaded. If you off-load the datafile any way except using the parallel port, there is no way to set this flag. Again, it won't adversely affect your program. It just removes some useful information from the system.

User-accessible assembly routines used in the example

SelEE	Activate chip select for flash EEPROM
Address:	FD8B
Note:	Use SelNone to deselect EEPROM when done.
Args:	None
Return:	None
Errors:	None

EWEEPR	Enable writing to flash EEPROM
Address:	FD7F
Note:	Call this after SelEE and before first call to EEErase. Use DWEEPR to disable EEPROM writing when done.
Args:	None
Return:	None
Errors:	None

EEErase Erase an EEPROM 4K block.

Address: FD85

Notes: Must call SelEE and EWEEPR before using this routine and call DWEEPR and SelNone after the last call to this routine.

To get block numbers for the argument to this routine, see the list starting at address 13B. The list is terminated with the value FF because this block number is never used.

Power drain for the TFX-11 is higher when erasing the EEPROM.

Args: Block number in B register
Return: None
Errors: Overflow set if time-out error, clear if no time-out Carry flag set if command fails, clear if pass

---

DWEEPR Disable writing to flash EEPROM

Address: FD82

Notes: Call this after last call to EEErase but before SelNone.

Args: None
Return: None
Errors: None

SelNone Deselect flash EEPROM

Address: FD8E

Args: None
Return: None
Errors: None

Example program
An example program that clears the entire datafile follows the documentation for the six functions needed to erase EEPROM blocks.

```
print
print "Erasing datafile"
TFErr = 0        // used to signal error (non-zero if error)

asm $
SelEE   equ     H'FD8B       ; function to select flash EEPROM
SelNone equ     H'FD8E       ; function to unselect flash EEPROM
EEErase equ     H'FD85       ; function to erase a block of flash EEPROM
EWEEPR  equ     H'FD7F       ; function to enable writing flash EEPROM
DWEEPR  equ     H'FD82       ; function to disable writing flash EEPROM
SBKADR  equ     H'FD52       ; function to set up buffer and its address
DFBlks  equ     H'13B        ; address of list of datafile blocks

        jsr     SelEE        ; select flash on SPI bus
        jsr     EWEEPR       ; write enable flash (needed for erasing, too)
        ldx     #DFBlks      ; X register holds address of datafile blocks
_EEClr  ldab    0,x          ; load next datafile block # in B register
        cmpb    #H'FF        ; check if this is end of list
        beq     _xeec        ; branch to _xeec if this is end of list
        jsr     EEErase      ; erase the flash block whose # is in B reg
        bvs     _eecerr      ; branch if time-out error on erase
        bcs     _eecerr      ; branch if block did not erase
        inx                  ; point to next position in datafile block list
        bra     _EEClr       ; loop back to do this block
_xeec   jsr     DWEEPR       ; write disable flash EEPROM
        jsr     SelNone      ; deselect flash EEPROM on SPI bus
        rts
_eecerr inc     TFErr+3      ; here if error, make value non-zero
        bra     _xeec        ; finish up as normal
        end

if TFErr = 0
        print "OK"
        // note - no need to flush the buffer to EEPROM in this example
        poke &hB0,0          // clear the datafile pointer
        poke &hB1,0
        poke &hB2,0
        poke &hB3,0
        call SBKADR,0        // set up buffer and its address for more Stores
else
        print "No good"
endif
stop    // when program stops, datafile pointer is copied to PIC
```

Right-click paperclip icon and Open or Save example program

Read data from the datafile in a TFBASIC program

Do you need this information?
In versions of TFBASIC previous to version 1.07, there was no documented way to read data from the datafile in your program. With version 1.07, the GET and GETS commands were introduced. If you have version 1.07 or above, use the GET function to read integers and floating point numbers and use GETS to read strings from the datafile. If you have an older version, use the method outlined here.

How to do it
The address of the next byte that will be read from the datafile is stored at addresses B4 to B7 (B4 is MSB). Fill this four byte value with the address to be read and call function GETMEM. The B4 to B7 value will have been incremented. So, if you are reading incremental values from the datafile, B4 to B7 will not have to be changed again.

Complications
Accessing flash EEPROM is slow. When TFBASIC needs to store data in the datafile (which is flash EEPROM), it stores the data in a RAM buffer. When this buffer is full, TFBASIC writes it to the EEPROM. When you read data, you read from a separate buffer. This buffer only reads from the EEPROM when an address is requested that is not already in the buffer. This can lead to two problems:

1) The EEPROM read buffer is not initialized on power up. You can get around this by reading a byte from a part of the datafile that is at least 32 bytes away from the area you want to read. Then, when you read the datafile in the area of interest, TFBASIC will fill the buffer correctly. Once you do this at the beginning of a program, you will not have do it again until the next power-up.

2) You will get incorrect values if you attempt to read data from the same 32-byte range the datafile pointer is pointing to. This is because TFBASIC has not written the data in the write buffer to EEPROM yet. When you read from this part of the datafile, TFBASIC fills the read buffer directly from the unwritten EEPROM. To check the datafile pointer, look at the TFBASIC read-only variable DFPNT. To see if the address you are trying to read is in the 32-byte buffer range, compare your read address with:

   DFPNT & &HFFFFFFE0

   If your address is greater than or equal to this, the data may be in the write buffer and not be available for reading. The example below uses this idea. Optionally, you can flush the EEPROM buffer to ensure that the data you are reading is not hidden in the buffer.

Additional notes
If the value you are reading is a floating point number, be sure that the return variable is a floating point variable. The example does not use this but it would look like:

```
call GetDF4,0,FltVar!
```

You must use the 4-byte assembly routine (GetDF4) provided in the example. You can also read strings from the datafile using the GetDFS assembly routine in the example. This requires that you pass the address of a string variable to GetDFS like this:

```
call GetDFS,varptr(StrVar$)
```

Notice there is no return argument in the CALL. We used varptr() to pass the address of string variable StrVar$ to the routine. Then GetDFS stores data directly to the string variable.

User-accessible assembly routine used in the example

GETMEM:	Get one byte of data from datafile at address held by DFPNT (addresses B4-B7) and increment datafile pointer.
Address:	FD91
Notes:	This routine assumes all data has been stored and flushed to the EEPROM. For instance, if you try to read the EEPROM page that the store function is currently writing to, this function will miss any data that has not been flushed to the EEPROM (up to 32 bytes). No need to use SelEE or SelNone around this routine.
Arg:	None
Return:	Byte in A register
Errors:	Calls TFBASIC error routine with 'How' error #5. Will be displayed before next TFBASIC command is executed.

TFBASIC example for reading data from the datafile

```
asm $
        rts             ; don't execute these until called from TFBASIC code

DFAddM  equ  H'B4       ; datafile address, most significant word
DFAddL  equ  H'B6       ; datafile address, least significant word
GETMEM  equ  H'FD91     ; address of function to read datafile

        ; function to set the datafile read pointer
SetDFA  stx  DFAddM     ; update MS word of DF Read Pointer
        std  DFAddL     ; update LS word of DF Read Pointer
        rts

        ; function to get the datafile read pointer (not used in example)
GetDFA  ldx  DFAddM     ; get MS word of DF Read Pointer
        ldd  DFAddL     ; get LS word of DF Read Pointer
        rts

        ; function to read one byte from datafile
GetDF1  jsr  GETMEM     ; read byte from datafile
        tab             ; move byte to LS byte of return value
        clra            ; clear upper bytes
        ldx  #0
        rts

        ; function to read two bytes from datafile
GetDF2  jsr  GETMEM     ; read MS byte from datafile
        psha            ; save MS byte on stack
        jsr  GETMEM     ; get LS byte in A register
        tab             ; move LS byte to B register
        pula            ; restore MS byte in A register
        ldx  #0         ; MS word is zero
        rts
```

```
        ; function to read four bytes from datafile
GetDF4  jsr  GETMEM      ; read MS byte from datafile
        psha             ; save MS byte on stack
        jsr  GETMEM      ; read next byte from datafile
        pulb             ; get MS byte back for now
        psha             ; reorder these two bytes so they
        pshb             ;   can be PULled into X register
        jsr  GETMEM      ; read next byte from datafile
        psha             ; this is MS byte, save on stack
        jsr  GETMEM      ; read LS byte from datafile
        tab              ; save LS byte in B register
        pula             ; A register has next more significant byte
        pulx             ; X register as MS word of 4-byte data
        rts

        ; function to read string from datafile
GetDFS  xgdx             ; move pointer to string variable to X register
        jsr  GETMEM      ; first byte is length of string that follows
        staa 0,x         ; store length byte in string variable
_gdfs   tsta             ; check if length byte is zero
        beq  _xgdfs      ; exit if no more characters to read
        psha             ; save string length byte
        pshx             ; save string variable address
        jsr  GETMEM      ; get next string character from datafile
        pulx             ; retrieve string variable address
        inx              ; point to next location in string variable
        staa 0,x         ; store character from datafile there
        pula             ; get length byte
        deca             ; count down one character
        bra  _gdfs       ; check if more characters to read
_xgdfs  rts              ; done, no return value

        end
print
print "Reading data from datafile"

call SetDFA,dfmax       // set the address to be read from datafile
call GetDF1,0,value     // do a dummy read to force filling buffer

GetAddress:
      input "Datafile address: "dfAddr

      // check if this address is out of range for the datafile
      if dfAddr > dfmax
            print "Address out of range: 0 to ",dfmax
            goto GetAddress
      endif

      // check if the current datafile pointer (for STORE) is
      // in a position where the data buffer may not have been
      // written to the EEPROM
      if dfAddr >= (dfpnt & &hfffffffe0)
            print "Data at this address may not been written yet"
      endif
```

```
GetNumBytes:
      input "Number of bytes to read (1,2 or 4): "numBytes
      if numBytes <> 1 & numBytes <> 2 & numBytes <> 4
            print "This example only works with 1, 2 or 4 bytes"
            goto GetNumBytes
      endif
      call SetDFA,dfAddr       // set the address to be read from datafile
      if numBytes = 1
            call GetDF1,0,value     // read one byte from datafile
            print "Data = ",#02h,value
      endif
      if numBytes = 2
      call GetDF2,0,value     // read two bytes from datafile
      print "Data = ",#04h,value
      endif
      if numBytes = 4
            call GetDF4,0,value     // read four bytes from datafile
            print "Data = ",#08h,value
      endif
stop
```

Right-click paperclip icon and Open or Save example program

Prevent Corrupt Datafiles after a Power Failure or Reset

It is possible to see corrupted data in the datafile of a TFX-11 after a power failure or after resetting the TFX-11 by using the RESET line (pin B-21).

The datafile pointer is volatile
This all comes from the fact that the datafile pointer in TFBASIC is a volatile value - whenever power is lost, the value is lost and on next start-up will be cleared to zero. This seems strange because the datafile itself is non-volatile. The reason is that the flash EEPROM, used to store the datafile and make it non-volatile, has a limited write capacity. After 100,000 or more erase-write cycles, the flash EEPROM cells begin to fail. This is fine for data because you will probably never write to any one location more than that over the lifetime of the TFX-11. But, if we wrote the current datafile pointer to flash EEPROM every time it was changed, you would ruin those few flash cells by the time you had filled the datafile once.

So, the datafile pointer must be stored in RAM and when power is lost or if the TFX-11 is reset, the datafile pointer is cleared to zero. If you've previously stored data in the datafile, your next write to the datafile will be writing to memory that already has data stored in it. You might think that you will just overwrite the old data with new data but that is incorrect.

Overwriting data corrupts the data
Flash EEPROM is not like normal memory. It must be erased before you store data. The reason for this is that flash EEPROM memory acts like a series of fuses. Erasing the memory resets (or closes) all the fuses and when you read them, they read as a 1. When you store data to a flash EEPROM, any bits that are 1 just leave the fuse in its closed state. Only bits that are 0 really change anything - a bit that is 0 opens (or burns) the fuse. Then when you read data back, fuses that are closed read as 1 and fuses that are burned (or open) read as 0. One of the limitations of flash EEPROM is that you cannot write a 1 into a cell that already has been written to 0 (open or burned). The ONLY way to close the fuse again is to erase the flash and this must be done in pages (or groups) of bytes. In the case of the TFX-11, this page size is 512 bytes.

An example
If you write the byte value 72 hex to a flash memory, the eight cells that comprise this byte look like:

0 1 1 1 0 0 1 0

where each 0 is an open (or burned) memory cell and each 1 is a closed (thus, unchanged) memory cell. If we now try to write the byte value 94 hex to this same flash memory byte, you want to write:

1 0 0 1 0 **1** 0 0

but you can't change the left-most bit (in bold face) back to a 1 - it's a burned fuse. Likewise, you can't change the third bit from the right (in bold face) back to a 1. The only way to change a 0 bit to a 1 bit is to erase the entire 512 byte page. If we do write 94 hex to this byte without erasing it first, only bits that are being changed to 0 will be written. So, we'll end up with:

0 0 0 1 0 **0** 0 0

Notice again the bits in bold face. If we read this data, it reads as 10 hex. It looks like garbage because it doesn't look like either of the bytes we wrote to that location. When the datafile pointer gets reset to 0, and you write data starting again at 0, you are only writing new 0 bits to your data and making it look like garbage.

Protect your data
We've included a way to keep this from happening. There are two read-only values available to your TFBASIC program that you need to check when your program starts and before you try to store data to the datafile. Read DFPNT and DFERASED. If DFERASED is true (or non-zero), it means the datafile has been erased and it is safe to write to the datafile. If DFERASED is false (or zero) and DFPNT is 0, it means you are going to be storing data over previously stored data. You have two options at this point:

1) Stop your program, off-load the data and erase memory OR

2) Find where your previously stored data ends and set the datafile pointer to one byte beyond that and continue with your program

Option 1 is simple and requires no more explanation. Option 2 is more complex. If you find that DFPNT and DFERASED are both 0 (meaning you will corrupt data if your use the STORE command), read the datafile bytes (starting at 0) until you find a block of bytes (a record) that read as FF hex - meaning they are erased. Then you have to set the DFPNT value to the beginning of the erased area. We have included a program that does this for you. The tricks to using this are that you must supply a record length to the program (that is, how many bytes are stored each time you call the STORE command) and there must be at least one non-FF hex byte in every record you write. Try running the example program. It works best if you Launch the program (select the Launch item under the Tattletale menu in TFTools) so that the program runs automatically on power-up. After the program ends, cycle the power so the program starts again. Do this a few times so you see how the program works. Notice that **this** program has a record length of 132 bytes. Your program will probably be different.

TFBASIC example for finding erased portion of datafile

```
print "Program to find where erased datafile starts"
RecordLength = 132                  // store 4 bytes 33 times
print "DFPNT = ",DFPNT
print "DFERASED = ",DFERASED
print "RecordLength= ",RecordLength

if (DFPNT = 0) & (DFERASED = 0)     // check for trouble
    print "Searching for erased area of datafile"
    gosub FindErasedDF
    NewDFPointer = iErasedArea
    gosub SetDFPointer
    print "Updated DFPNT = ",DFPNT
endif
if (DFPNT >= 0) & (DFPNT < DFMAX+1) // if datafile pointer is valid
    print "Storing data"
    for i = 1 to 33
        store #4,i
    next i
else
    print "Unable to find an erased area"
endif
stop
```

```
//*********************************************************************
// This section of assembly code is only needed for TFBASIC versions
// previous to version 1.07.
//
// InitDFR - Prepares the datafile read buffer by setting the read
//           address to a high number and doing a dummy read. Then
//           it resets the read address to zero. When the next read
//           is done, the read buffer will be handled correctly.
//
// DFRead - Reads the next byte from the datafile and then updates
//          the TFBASIC variable iGetPointer used by this example
//          to keep track of where it is reading from.
//*********************************************************************
asm $
         rts   ; don't execute these until called from TFBASIC code

DFAddM   equ   H'B4          ; datafile address, most significant word
DFAddL   equ   H'B6          ; datafile address, least significant word
GETMEM   equ   H'FD91        ; address of function to read datafile

InitDFR  ldd   #0            ; these four instructions will
         std   DFAddM        ;  set the datafile read pointer
         ldd   #H'FF         ;   to a high value
         std   DFAddL
         jsr   GETMEM        ; this is a dummy read
         ldd   #0            ; these three instructions
         std   DFAddM        ;  reset the datafile read pointer
         std   DFAddL        ;   to zero so next GETMEM reads correctly
         rts

DFRead   jsr   GETMEM        ; read next byte from datafile
         psha                ; save byte read from datafile
         ldd   DFAddM        ; get updated datafile read pointer
         std   iGetPointer   ;  and update the TFBASIC variable
         ldd   DFAddL        ; do the same thing for the
         std   iGetPointer+2 ; least significant word
         pulb                ; will return datafile byte in B reg
         clra                ; clear all other returns
         ldx   #0
         rts
         end


//*********************************************************************
// FindErasedDF
//
// Returns with iErasedArea set to start of erased area in datafile or
// equal to DFMAX + 1 (impossible datafile address) if datafile is
// entirely filled
//
// Before calling this function, RecordLength MUST be set to the number
// of bytes you store to the datafile in each record.
//*********************************************************************
FindErasedDF:
    iErasedArea = DFMAX+1   // start by assuming we can't find an erased area
    iGetPointer = 0         // start looking at beginning of datafile area
```

```
    if vers < 107           // if using TFBASIC version 1.06 or earlier
        call InitDFR,0      //   initialize datafile reading
    endif
    iCount = RecordLength   // the length of data we store, in bytes

    while iGetPointer <= DFMAX   // while we haven't reached the end of datafile
        if vers < 107
            call DFRead,0,dfvalue           // use this for version 1.06 or earlier
        else
            dfvalue = get(iGetPointer)      // use this for current versions
        endif
        if dfvalue = &HFF                   // if in an erased area
            if iErasedArea = (DFMAX + 1)    // if first byte of erased area...
                iErasedArea = iGetPointer-1 // mark as start of erased  area
            endif
            iCount = iCount - 1             // count down
        else                                // else NOT in an erased area
            iErasedArea = DFMAX + 1         // mark that we haven't found it
            iCount = RecordLength           // reset erased byte count
        endif
        if iCount = 0
            return
        endif
        if (iGetPointer % 1024) = 0 print iGetPointer,\13;
    wend
    return

//*********************************************************************
// SetDFPointer
//
// This subroutine will set the datafile pointer (DFPNT) to the value
// in NewDFPointer. The subroutine checks that NewDFPointer is a valid
// number from 0 to DFMAX. If the value is not valid, this subroutine
// returns without doing anything (except printing an error message).
//*********************************************************************
SetDFPointer:
    if NewDFPointer < 0 | NewDFPointer > DFMAX
        print "Invalid datafile pointer"
        return
    endif

    call &HFD88,0                   // flush data in buffer to EEPROM (FEESBF)
    pDFP = &HB3                     // location of datafile pointer
    poke pDFP, NewDFPointer & &HFF  // set LSB of new DFPNT
    pDFP = pDFP - 1                 // point to next byte of datafile pointer
    NewDFPointer = NewDFPointer/256
    poke pDFP, NewDFPointer & &HFF  // set next byte of new DFPNT
    pDFP = pDFP - 1                 // point to next byte of datafile pointer
    NewDFPointer = NewDFPointer/256
    poke pDFP, NewDFPointer & &HFF  // set next byte of new DFPNT
    pDFP = pDFP - 1                 // point to next byte of datafile pointer
    NewDFPointer = NewDFPointer/256
    poke pDFP, NewDFPointer & &HFF  // set MSB of new DFPNT

    call &HFD52,0                   // set up buffer and address for STORE
    return
```


Right-click paperclip icon and Open or Save example program