ZBasic

Application Note

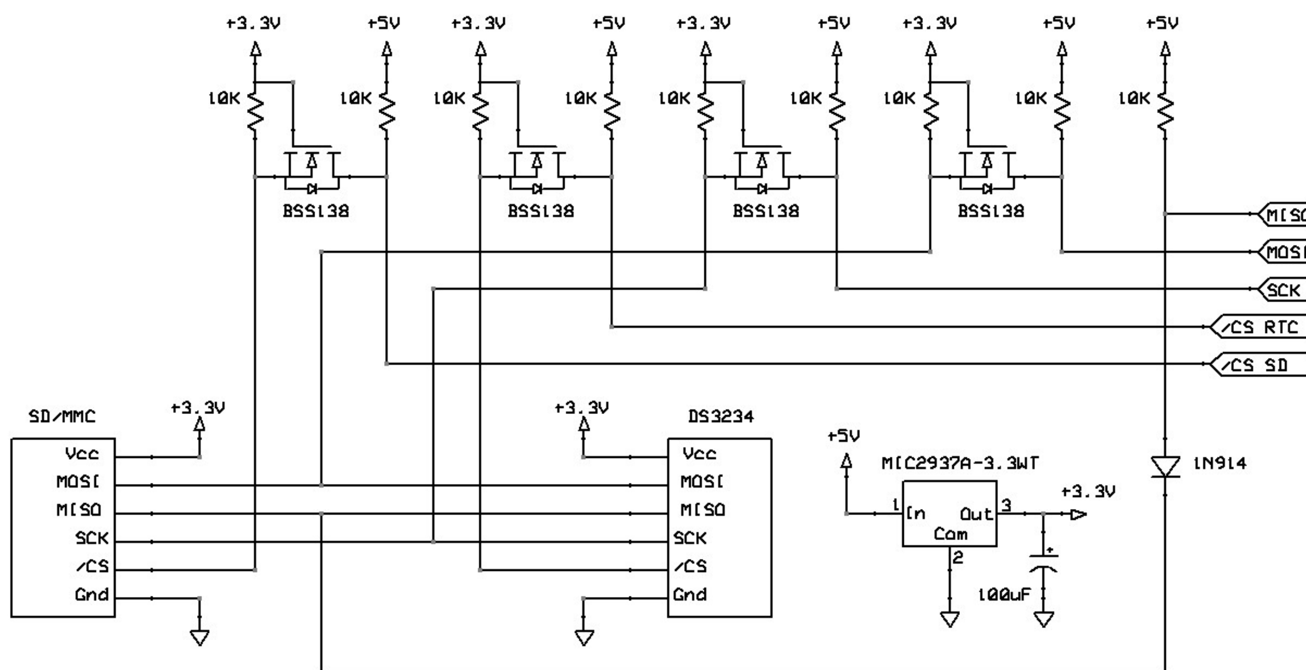
AN-222 Interfacing an SD Flash Drive

Introduction

This application note describes how to interface an SD Flash Drive to a ZBasic device and how to interact with the drive using the FatFS open source file system code base. Because the FatFS code is written in C, advanced features of ZBasic are used to access the FatFS functionality from ZBasic as well as to provide the required low level access code (written in ZBasic). Note that version v4.0.2 or later of the ZBasic compiler is required in order to successfully compile the code provided with this application note.

Hardware Connections

A ZX-24n target device is used in this application note but any ZX device or generic AVR target device could be used as long as it has sufficient Flash memory and an SPI interface. (The application note code compiles to a Flash image size of about 25K bytes.) Breakout boards for the SD card receptacle, battery backed RTC and logic level converters are used to simplify prototype construction. Other required components are a 5 volt supply (not shown) and a 3.3V regulator.

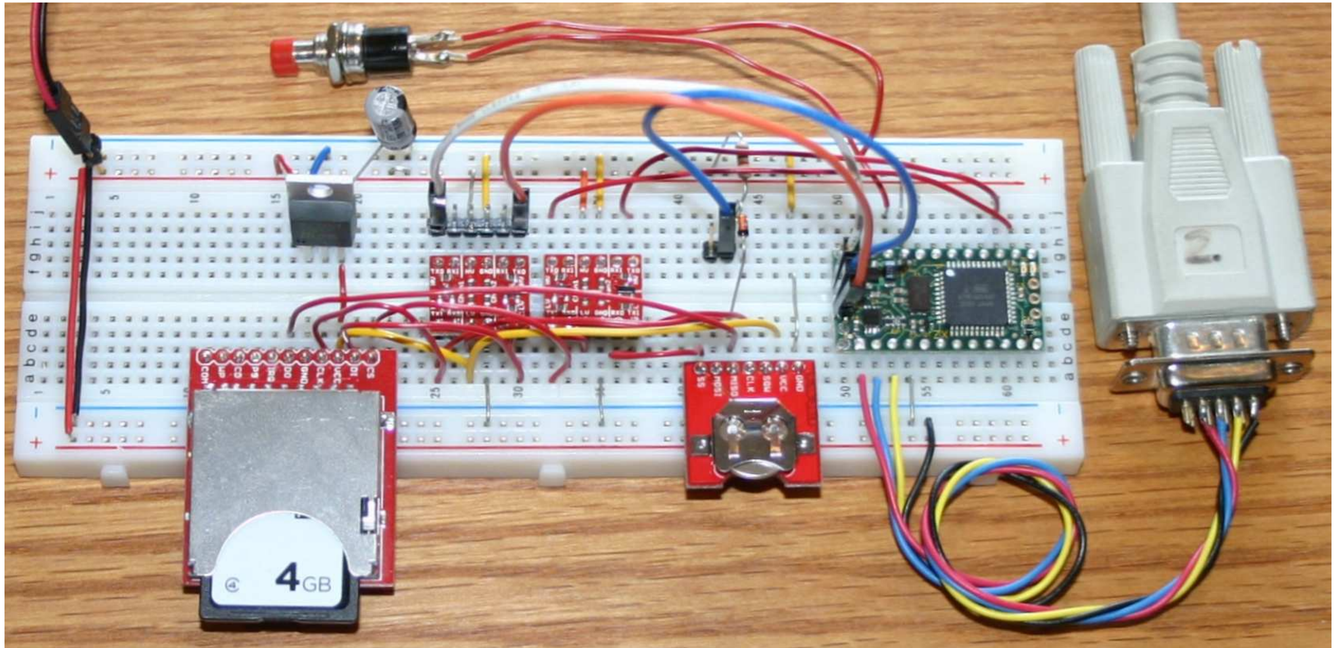


The connections to the ZX-24n are shown on the right side of the schematic above. The MISO, MOSI and SCK signals are connected to square pins soldered into the holes on the "pin 1" end of the ZX-24n while the two SPI chip select signals are connected to pins 14 and 15 of the ZX-24n. Because SD/MMC cards operate at 3.3V and are not 5 volt tolerant, level conversion circuitry is used to translate the 5 volt outputs of the ZX-24n to the lower levels. The sole output from the SD card, MISO, is connected to the ZX-24n through a diode, effectively providing level conversion from 3.3V to 5V.

The DS3234 timekeeping chip can operate at either 3.3V or 5V. We chose to place it on the low voltage side but if it were placed on the 5V side, of course, no level conversion would be needed for the signals to it. The photo

Copyright © 2012 Elba Corp.
Published May 2012

below shows the prototype constructed on a solderless breadboard. The SD card and its adapter are easily identifiable on the left. Just above that is the 3.3V regulator and filter capacitor. In the center of the breadboard are the two level converter breakout boards (of which only the MOSFETs are used). Farther to the right is the DS3234 breakout board (with battery) and above that is the resistor/diode level converter for MISO. The red pushbutton switch is connected to the reset line of the ZX-24n and 5V power is being supplied via the black/red wire pair in the upper left corner.



The table below gives the source and part number for several of the key components. The remaining parts are relatively non-critical. The diode should have a reasonably low forward voltage - a Schottky diode (e.g. BAT42) would be preferred, due to its lower forward voltage, but almost any small signal silicon diode like the 1N914 we happened to have on hand would probably work fine. Note, too, that resistor/diode level converters could be used instead of the MOSFET level converters (resistor connected to 3.3V, cathode connected to the 5V signal).

SD Prototype Parts

Item	Source	Part Number
SD/MMC Breakout Board	SparkFun	BOB-00204
DS3234 Breakout Board	SparkFun	BOB-10160
Level Converter Breakout Board	SparkFun	BOB-08745
3.3V regulator	DigiKey	576-2234-ND

FatFS Project Files

The files comprising the ZBasic FatFS project include ZBasic source code, C source code and headers, and an AVR assembly language file. The files are listed below along with a brief description of each. The last three files in the list (grey background) are not needed for normal operation.

ZBasic FatFS Project Files

File	Description
FatFS.bas	Provides the ZBasic `Main()` subroutine and several example uses of FatFS functions.
FatFS_mmc.bas	Provides the low-level disk access routines required by the FatFS code.
DS3234.bas	This file provides routines for interacting with the DS3234 timekeeping chip.
FatFS/ff.c	Provides the implementation of FAT file system functionality.
FatFS/ff.h	Provides definitions for the higher-level FAT file system functions.

Copyright © 2012 Elba Corp.

FatFS/diskio.h	Provides definitions for the low-level disk interface functions.
FatFS/ffconf.h	This header file specifies configuration information that controls what functionality is provided by the FatFS code.
FatFS/integer.h	This header file provides type mapping that allows FatFS to be compiled for many different types of computers.
FatFS_test.c	This file, derived from the FatFS example file `main.c`, provides a command-driven "monitor" functionality that allows exercising many aspects of the FatFS functionality.
FatFS/xitoa.S	Provides some special-purpose routines needed by the "monitor" functionality.
FatFS/xitoa.h	This is a header file describing the C-callable entry points in `xitoa.S`.

Excepting one, all of the files in the `FatFS` sub-directory are exact copies of those available from the FatFS distribution (see http://elm-chan.org/fsw/ff/00index_e.html) in the `sample/avr` subdirectory. The one file with changes is the configuration file `FatFS/ffconf.h`, which changes are summarized below.

Line	Description
60	Changed the code page to 437.
93	Disabled long filename support (primarily to reduce code size).
128	Set the number of volumes to 1.

FatFS Low Level Disk Interface

The FatFS code is designed to be simple to implement on many different types of physical media, e.g. PATA or SATA hard disks, SD/MMC Flash drives, Compact Flash drives, etc. All that is necessary to allow the higher level FatFS routines to access a particular physical storage medium is to implement several "worker" routines that handle all of the interaction with the drive. A brief description of each of the required routines together with a ZBasic signature is given in the table below. The `DRESULT` type is an enumeration defined in the FatFS header files.

Required FatFS Disk Interface Routines

Description	ZBasic Signature
Initialize a disk drive.	`Function disk_initialize(ByVal drv as Byte) as Byte`
Get disk status.	`Function disk_status(ByVal drv as Byte) as Byte`
Read one or more sectors.	`Function disk_read(ByVal drv as Byte, ByRef buf() as Byte, _ ByVal sector as UnsignedLong, ByVal count as Byte) as DRESULT`
Write one or more sectors.	`Function disk_write(ByVal drv as Byte, ByVal buf() as Byte, _ ByVal sector as UnsignedLong, ByVal count as Byte) as DRESULT`
Control device-dependent features.	`Function disk_ioctl(ByVal drv as Byte, ByVal ctrl as Byte, _ ByRef data() as Byte) as DRESULT`
Get the current time/date in FAT format.	`Function get_fattime() as UnsignedLong`

For this application note, these required low-level functions are implemented in the file `FatFS_mmc.bas`. Consider, for example, the `disk_status()` function, the code for which is shown below. For the most part, this function looks like any normal ZBasic function. The unusual part is the attribute clause following the function type. The attribute clause syntax allows one or more attributes to be given that specify certain characteristics for the routine. The `Used` attribute tells the ZBasic compiler to output code for the function even if it isn't referred to anywhere in the ZBasic code. In this use case, the `disk_status()` routine isn't invoked in any ZBasic code but it still must be output because the FatFS C code *does* refer to it and a link-time error would occur if it weren't present.

```
Public Function disk_status(ByVal drv as Byte) _
      as Byte Attribute(Used, "Alias:disk_status")
   If (drv <> 0) Then
       disk_status = STA_NOINIT
   Else
       disk_status = stat
   End If
End Function
```

Copyright © 2012 Elba Corp.

The second attribute tells the ZBasic compiler to give the function a specific name in the generated code rather than to use the normal ZBasic naming convention for the generated routines. This is necessary because the FatFS code expects to invoke a routine named `disk_status` and a link-time would occur if that routine doesn't exist.

In the routine `disk_status()`, note the use of the identifier `STA_NOINIT`. This identifier along with many others including the `DRESULT` enumeration type mentioned above are defined in the FatFS header files `ff.h` and `diskio.h`. Rather than duplicating those definitions in ZBasic code it is much easier to use the existing definitions and the new C/C++ header importing feature provides this ability. Near the top of the file `FatFS_mmc.bas` there are two lines of code related to this.

```
#import Public "FatFS/diskio.h"
#import Public "FatFS/ff.h"
```

Each of these lines causes the ZBasic compiler to run a specialized utility to process the indicated C/C++ header file to extract information about data types, classes, structures, unions, enumerations, variables and functions that can be referred to in ZBasic code. The importing process also extracts information about identifiers that are defined using #define and these are made available to ZBasic code as if they were created in ZBasic using #define. This new feature makes it much simpler to incorporate existing code written in C/C++ and AVR assembly language into your ZBasic application. More information about importing can be found in chapter 6 of the ZBasic Language Reference manual.

The low-level routines in FatFS_mmc.bas are a relatively straightforward translation to ZBasic of the routines provided in the FatFS download in the file `sample/avr/mmc.c`. Because of some peculiarities in the SD/MMC SPI protocol, the ZBasic routine CmdSPI() could not be used to interact with the SD card. For example, as part of the initialization sequence the SD card's SCK line needs to be clocked for at least 80 cycles with the chip select not asserted. Consequently, the SPI interaction with the SD card is implemented using new ZBasic routines for low-level SPI operations that are similar to the ZBasic low-level I2C routines.

ZBasic Low-Level SPI Routines

Routine	Description
`SPIStart()`	Perform one or more actions for starting an SPI transaction.
`SPIPutByte()`	Send a byte to the slave and retrieve the returned byte.
`SPIPutData()`	Send a stream of bytes to the slave discarding the returned bytes.
`SPIGetByte()`	Send a dummy byte to the slave and retrieve the returned byte.
`SPIGetData()`	Send multiple dummy bytes to the slave and store the returned bytes.
`SPIStop()`	Perform one or more actions for terminating an SPI transaction.

Detailed information on the operation of the low-level SPI routines may be found in the ZBasic System Library Manual (v4.0.1 or later).

Implementing FatFS Disk Operation Timeouts

The AVR sample code provided in the FatFS distribution uses a dedicated timer to implement a timeout capability. The timer is configured to generate an interrupt at 100Hz and in the interrupt service routine (ISR) two variables are decremented (if non-zero) on each interrupt. This gives the ability to realize two independent timeout timers with a 10mS resolution. Rather than using a dedicated timer for this purpose, we chose to implement the timeout mechanism using the ZBasic RTC timer. On most AVR devices, the timer used for the RTC has two "compare match" registers. On the mega644P, the registers are OCR0A and OCR0B on 8-bit Timer0. The OCR0A register is used to realize the desired RTC interrupt frequency (1024Hz in the case of the ZX-24n), being set at 224. The timer is configured to reset the TCNT0 register to zero on compare match so the TCNT register will increment from zero to 224 and then reset to zero again. With the timer's prescaler selector set for divide-by-64 and the main clock running at 14.7456MHz this yields the desired RTC interrupt frequency of 1024Hz (14.7456MHz / 64 / 225).

Given that the TCNT0 register changes at a rate of 230.4KHz (14.7456MHz / 64), realizing a 100Hz interrupt requires dividing that rate by 2304 but that can't be accomplished directly for two reasons. Firstly, the TCNT register ranges from 0 to 224 (thus never reaching 2304) and secondly, 2304 is too large to represent in 8 bits. However, we observe that 2304 can be factored as 144 times 16. Since the 144 factor is less than the range of

Copyright © 2012 Elba Corp.

TCNT0 we can set OCR0B to generate an interrupt at 1600Hz (230.4KHz / 144) and then take action in the CompareMatchB ISR every 16 interrupts thus yielding an effective rate of 100Hz. A simplified representation of the ISR code (contained in `FatFS_mmc.bas`) is shown below. The divide-by-16 aspect is implemented by incrementing the `count` variable on each interrupt and then masking the result to 4 bits with an And operation. The result of this operation will be zero every 16 interrupts and on those occasions the `Timer1` and `Timer2` variables are decremented if non-zero.

```
' These timer variables are decremented by the compare match ISR.
Private Timer1 as Byte Attribute(Volatile)
Private Timer2 as Byte Attribute(Volatile)

ISR TIMER0_COMPB()
   Const CountMask as Byte = &H0f
   Const Delta100Hz as Byte = 144

   ' advance the interrupt counter
   Static count as Byte
   count = (count + 1) And CountMask
   If (count = 0) Then
       ' a 10mS interval has passed
       Timer0 = Timer0 + 1
       If (Timer1 <> 0) Then
           Timer1 = Timer1 - 1
       End If
       If (Timer2 <> 0) Then
           Timer2 = Timer2 - 1
       End If
   End If

   ' advance the compare register for the next interrupt
   Dim delta as Byte
   delta = Register.OCR0A - Register.OCR0B
   If (delta >= Delta100Hz) Then
       Register.OCR0B = Register.OCR0B + Delta100Hz
   Else
       Register.OCR0B = Delta100Hz - delta
   End If
End ISR
```

The second part of the ISR is concerned with arranging for the next CompareMatchB interrupt to occur 144 counts later. This is complicated by the restricted range of the TCNT0 register. The solution is to make the new setting of OCR0B be the existing setting plus 144 counts modulo 225. This is accomplished without actually using the modulo operation by first computing the difference between the OCR0A setting (224) and the current OCR0B setting. If that difference is not greater than 144, the current OCR0B setting can simply be advanced by 144. On the other hand, if the difference is less than 144, OCR0B is set to 144 less that difference thus effectively performing the modulo operation.

It should be noted that the ZBasic low level X-10 functionality also uses the CompareMatchB interrupt to implement the required X-10 timing. Consequently, the low level X-10 functionality cannot be used in an application that uses the CompareMatchB interrupt as depicted above. In such cases, an alternate means of implementing the 100Hz timing would need to be devised. One possibility would be to use an external timing signal connected to an input pin and implement the timing in a pin-change interrupt handler or external interrupt handler. One possible source of an external timing signal is an output from a timekeeping chip like the DS3234 which, for example, can be configured to output a 1024Hz square wave. Dividing this by 10 in a pin change ISR would yield 102.4Hz timing - probably close enough for the purposes of disk operation timeout.

Implementing FatFS Disk Date/Time Functionality

When FatFS is configured to support disk write operations, it is required to supply a function `get_fattime()` that returns the current time and date in FAT format. The implementation of `get_fattime()` in `FatFS_mmc.bas` converts the current time and date maintained by the ZBasic RTC to the required format, shown in simplified form below.

```
Public Function get_fattime() as UnsignedLong Attribute(Used, "Alias:get_fattime")
   Dim time as UnsignedLong
   Dim date as UnsignedInteger
   Atomic
       ' retrieve ZBasic time and date in packed form
       time = GetTimeValue()
       date = GetDateValue()
   End Atomic

   ' combine the date/time values, correct for different year 0
   get_fattime = MakeDword(HiWord(time), date + Shl(1999 - 1980, 9))
End Function
```

The required FAT time format is a 32-bit value with fields for each time/date component as shown in the table below. This can be seen as two 16-bit values where the most significant word contains date information while the least significant word contains time information. The new ZBasic functions `GetDateValue()` and `GetTimeValue()` make it relatively easy to produce the FatFS date/time format. The `GetDateValue()` function returns a 16-bit value whose bit fields exactly match those of the required date word except that the zero year is different, being 1999 for the GetDateValue() function and 1980 for the FatFS date format. This difference is easily handled by applying a correction factor to adjust the year offset.

FatFS Date/Time Format

Bits	Description
31-25	Year (relative to 1980, 0 to 127).
24-21	Month (1 to 12).
20-16	Day (1 to 31).
15-11	Hour (0 to 23).
10-5	Minute (0 to 59).
4-0	Two-second count (0-29)

The ZBasic `GetTimeValue()` function returns a 32-bit value whose most significant word exactly matches the required time format while the least significant word contains an extra "seconds" bit and 15 bits representing a fraction of a second. Given the two-second resolution of the FatFS time value it was chosen to simply discard the least significant word returned by `GetTimeValue()` rather than attempting to round to the nearest two-second interval.

Of course, using the ZBasic RTC date/time requires a means to set the time with reasonable accuracy each time the application begins running. This need is met by retrieving the date/time from the DS3234 timekeeping chip at startup and setting the ZBasic RTC to that time, implemented by the subroutine shown below (contained in `DS3234.bas`). All of the called routines and the definition of the structure `TimeDate` are found in the same file.

```
Sub SetZBasicRTC()
   Call OpenSPI(chan, &H05, csPin)
   Call InitDS3234()

   Dim td as TimeDate
   Call GetTimeDS3234(td)
   Call SetTimeZBasic(td)
End Sub
```

Copyright © 2012 Elba Corp.

Example Operations

To demonstrate use of FatFS functions from ZBasic some simple examples were created. The code below shows how to display a directory listing of the root directory of the disk.

```
Dim fs as FATFS

Sub Main()
   ' initialize the external timekeeping chip, set ZBasic time
   Call SetZBasicRTC()

   ' perform hardware initialization
   Call FatFS_init()

   ' initialize the SD card parameters
   Dim b as Byte
   b = disk_initialize(0)

   ' mount drive 0
   Dim res as FRESULT
   res = f_mount(0, fs)

   ' output a directory listing for the root directory
   Call DisplayDir("/")
End Sub
```

The `DisplayDir()` subroutine uses several FatFS functions to obtain information about directory entries. One of the issues that must be dealt with is that in C/C++ code strings are typically implemented using a series of bytes with a zero byte marking the end of the string. This "null-terminated string" paradigm is equivalent to a RAM-resident ZBasic array of Byte values with a zero at the end. Consequently, a null-terminated string could be passed to a C/C++ function by defining a Byte array of sufficient length, copying characters from the ZBasic String variable to the array and then putting a zero byte at the end. With a routine like `DisplayDir()` below, it is not known beforehand how long the string might be so a Byte array of the maximum string size (255) plus 1 would need to be used to ensure proper operation in all cases.

We elected to take a different approach for this application note. The `NullTermString()` function (not shown here) returns a normal ZBasic string that is guaranteed to reside in RAM and also has a null character at the end of the string. Then, all that is needed is a way to get the address of the first character of the string to pass to the C function. This is implemented by the `StrPtr()` function shown below, defined as returning a reference to a byte (`Byte ByRef`) thus being an exact match with what is expected by a C/C++ function expecting a null-terminated string.

```
Function StrPtr(ByRef st as String) as Byte ByRef
   StrPtr.DataAddress = StrAddress(st)
End Function
```

The `DisplayDir()` subroutine works by calling the FatFS function `f_opendir()` and then repeatedly calling the FatFS function `f_readdir()`, outputting information about each directory entry.

```
Sub DisplayDir(ByVal dirName as String)
   Dim res as FRESULT

   ' display a directory listing of the root
   Dim d as DIR
   Dim s as String
   s = NullTermString(dirName)
   res = f_opendir(d, StrPtr(s))
   If (res = FR_OK) Then
       Dim fileSize as UnsignedLong = 0
```

Copyright © 2012 Elba Corp.

```
        Dim dirCnt as UnsignedInteger = 0
        Dim fileCnt as UnsignedInteger = 0
        Dim finfo as FILINFO
        Do
            ' read the next directory entry
            res = f_readdir(d, finfo)
            If ((res <> FR_OK) Or (finfo.fname(1) = 0)) Then
                Exit Do
            End If
            If (CBool(finfo.fattrib And AM_DIR)) Then
                dirCnt = dirCnt + 1
            Else
                fileCnt = fileCnt + 1
                fileSize = fileSize + finfo.fsize
            End If

            ' output the attribute indicators
            Dim i as Integer
            For i = 1 to SizeOf(fileAttrMask)
                Debug.Print IIF(CBool(finfo.fattrib And fileAttrMask(i)), _
                       Chr(Asc(fileAttrChar, i)), "-");
            Next i
            Debug.Print " ";

            ' output the file date/time
            Debug.Print Shr(finfo.fdate, 9) + 1980;
            Call outputDigits(CByte(Shr(finfo.fdate, 5) And &H0f), "/")
            Call outputDigits(CByte(finfo.fdate And &H1f), "/")
            Call outputDigits(CByte(Shr(finfo.ftime, 11) And &H1f), " ")
            Call outputDigits(CByte(Shr(finfo.ftime, 5) And &H3f), ":")
            Call outputDigits(Shl(CByte(finfo.ftime And &H1f), 1), ":")

            ' output the file size
            Debug.Print Right("          " & CStr(finfo.fsize), 11); "  ";

            ' output the file name (N.B.: fname array is null-terminated)
            Debug.Print finfo.fname
        Loop

        ' output disk statistics
        Debug.Print fileCnt; " file(s), "; dirCnt; " directories, ";
        Debug.Print fileSize; " bytes total"
        Dim freeClusters as UnsignedLong
        res = getFreeClusters(StrPtr(s), freeClusters, fs)
        If (res = FR_OK) Then
            Debug.Print Shr(freeClusters * CULng(fs.csize), 1); "K bytes free"
        End If
    End If
End Sub
```

After the last file in the directory is processed, additional information is displayed about the disk. One interesting aspect of this is the determination of the number of free clusters on the disk. The FatFS function to obtain this information, `f_getfree()`, is the only one of the FatFS functions that cannot be called directly from ZBasic because its third parameter must be a pointer to a pointer to a structure which cannot be represented in ZBasic. To work around this issue, we created an inline C function in ZBasic, shown below, to act as an intermediary.

```
Declare Function getFreeClusters(ByRef dir() as Byte, _
      ByRef freeClusters as UnsignedLong, ByRef fs as FATFS) as FRESULT
```

Copyright © 2012 Elba Corp.

```
#c
FRESULT
getFreeClusters(uint8_t *dir, uint32_t *freeClusters, FATFS *fs)
{
   return(f_getfree((TCHAR *)dir, freeClusters, &fs));
}
#endc
```

When compiled and downloaded, the example produces output similar to that shown below.

```
ZBasic v4.0.1
----A 2012/04/16 17:18:28   12506729  FILE.BIG
D---- 2012/04/30 14:36:58          0  DATA
----A 2012/04/26 11:58:18        302  HELPINFO.TXT
2 files, 1 directories, 12507031 bytes total
3851680K bytes free
```

To further demonstrate some of the capability of FatFS, we can add code to create a file and write some data to it. The example subroutine TestWrite() below demonstrates how this might be done.

```
Sub TestWrite()
   Dim res as FRESULT
   Dim f as FIL
   Dim s as String

   ' create the file, overwriting an existing file
   s = NullTermString("/thefox.txt")
   res = f_open(f, StrPtr(s), FA_CREATE_ALWAYS Or FA_WRITE)
   If (res <> FR_OK) Then
       Debug.Print "f_open() returns "; res
       Exit Sub
   End If

   ' write some data to the file
   s = NullTermString("The quick brown fox jumped over the lazy dog." _
         & Chr(&H0d) & Chr(&H0a))
   Dim dataLen as UnsignedInteger
   Dim writeCnt as UnsignedInteger
   dataLen = CUInt(Len(s)) - 1
   res = f_write(f, StrPtr(s), dataLen, writeCnt)
   If (res <> FR_OK) Then
       Debug.Print "f_write() of "; dataLen; " bytes returns "; res
   ElseIf (writeCnt <> dataLen) Then
       Debug.Print "f_write() reports "; writeCnt; " of "; dataLen; " bytes written"
   End If

   ' close the file
   res = f_close(f)
End Sub
```

Modifying the Main() routine to call TestWrite() before displaying the directory produces this output.

```
ZBasic v4.0.1
----A 2012/05/09 12:20:56         47  THEFOX.TXT
----A 2012/04/16 17:18:28   12506729  FILE.BIG
D---- 2012/04/30 14:36:58          0  DATA
----A 2012/04/26 11:58:18        302  HELPINFO.TXT
3 files, 1 directories, 12507078 bytes total
3851648K bytes free
```

Performance

Using the "fr" command of the testing monitor (in FatFS_test.c) the speed for reading 100,000 bytes from a file was measured at over 200KB/sec. The write speed wasn't tested but it is believed that it should be about half the speed of reading.

ZBasic and C/C++ Type Correspondence

In order to write ZBasic code to interact with functions written in C/C++ it is necessary to know how ZBasic data types map to C/C++ data types. The table below shows the equivalency for fundamental types.

Correspondence between ZBasic and C/C++ Types

ZBasic Type	C/C++ Type
Byte	char, unsigned char, signed char, uint8_t, int8_t
Integer	int, short, short int, int16_t
UnsignedInteger	unsigned int, unsigned short, uint16_t
Long	long, long int, signed long int, int32_t
UnsignedLong	unsigned long, unsigned long int, uint32_t
Single	float, double

Further, a reference to a ZBasic fundamental data type (or user-defined type, e.g. structure, class, enumeration, etc.) is compatible with a pointer to the equivalent type in C/C++. For example, the C/C++ type `uint8_t *` (and `uint8_t&` in C++) corresponds to `Byte ByRef` in ZBasic.

Software

The ZBasic, C, AVR assembly language and associated header files are provided in a .zip archive. The code and documentation in the FatFS subdirectory comes directly from the FatFS distribution; license information for that code may be found in the distribution files and/or at the FatFS website. The ZBasic code to interface to FatFS is provided under a BSD type license, details for which may be found in the source code files themselves. Information about the various FatFS functions is available via the file `FatFS.html`.

Author

Don Kinzer is the founder and CEO of Elba Corporation. He has extensive experience in both hardware and software aspects of microprocessors, microcontrollers and general-purpose computers. Don can be contacted via email at dkinzer@zbasic.net.

Disclaimer: Elba Corp. makes no warranty regarding the accuracy of or the fitness for any particular purpose of the information in this document or the techniques described herein. The reader assumes the entire responsibility for the evaluation of and use of the information presented. The Company reserves the right to change the information described herein at any time without notice and does not make any commitment to update the information contained herein. No license to use proprietary information belonging to the Company or other parties is expressed or implied.

Copyright © Elba Corp. 2012. All rights reserved. ZBasic, ZX-24, ZX-32, ZX-328, ZX-40, ZX-44, ZX-1281, ZX-1280, ZX-32a4, ZX-128a1 and combinations or variations thereof are trademarks of Elba Corp. Other terms and product names may be trademarks of other parties.