USB MSD Host Library

2012-10-08

Content

Introduction

This is a library for making an **USB host** capable of reading and writing sectors from/to **USB memory sticks** (pendrives) or **card readers**. At this moment only a version for PIC24 is available.


To read/write actual files from/to the USB memory stick you need e.g. a Fat16 or Fat32 library.

The library is tested with 2 sticks of different brands and one card reader of a third brand. (Still no guarantee...)

Usage in a program

Initialisation and test for readiness

The initialisation of the library and test for readiness of the USB memory stick is done as follows:

```
uses USB_HOST_MSD_Library, Debug;
...

begin
  { Main program }

  InitMain;

  InitUsb;                          // <--- initialisation of the library
  repeat
  until USB_MSD_Device_Ready or     // <--- test for readyness of the USB stick
        (Usb_Error > 0) or
        (Msd_Error > 0);
```

```
  if USB_MSD_Device_Ready then        // the USB memory stick is ready
  begin
    uart_write_line('');
    uart_write_line('MSD Ready');
    uart_write_line('');

    LatA.0 := 1;                      // signal readyness (example)

    USB_MSD_Device_Vendor(TmpS);      // show some MSD stick data
    uart_write_line(TmpS);
    USB_MSD_Device_Product(TmpS);     //
    uart_write_line(TmpS);
    USB_MSD_Device_Version(TmpS);     //
    uart_write_line(TmpS);

    uart_write_line('');
  end
  else
  begin                                  // the USB memory stick gives an error
    uart_write_line_word_hex(USB_Error);
    uart_write_line_word_hex(MSD_Error);
    while true do; // stop all processing
  end;
```

The above example shows some vendor and product info if the device becomes ready, and the error codes if the device does produce an error.

As you can see the initialisation is done with the “InitUsb” routine, the readiness of the stick is tested with “USB_MSD_Device_Ready”. In case you do not want to block the software if “USB_MSD_Device_Ready” stays false, you should also test both “Errors” (USB_Error and MSD_Error). If one of these becomes > 0 then you can stop waiting for “USB_MSD_Device_Ready”, see the example above.

Reading and Writing sectors

Reading and writing a sector (called a “block” in MSD terminology) is done as follows:

```
var Buff      : array[512] of byte;
      Success : boolean;

...
Success := USB_MSD_Read_Sector(2000, Buff); // sector 2000 is read into Buff
// process the buffer content here
...

...
// define the buffer content here
Success := USB_MSD_Write_Sector(5000, Buff); // Buff is written to sector 5000
...
```

As you can see this is very similar to the sector read/write routines of an SD/MMC card.

Assumptions about the USB memory stick

The library assumes a number of things about the USB memory stick connected:

- The stick (or card reader) is powered from an external 5V source
- The MSD device is defined in the first interface in configuration 0 (USB)
- The interface descriptor class, subclass and protocol are checked against the standard
- The devices USB endpoint 0 (default pipe) must have 64 bytes of length
- The devices USB bulk endpoints must have 64 bytes of length
- The device must be a Full speed or High speed type (always Full speed is mode is used)
- The device's "Block size" (= sectorsize) must be 512 bytes

Interface of the unit

```
procedure USB_Interrupt;
// To be called from the main interrupt routine (iv IVT_ADDR_USB1INTERRUPT)

procedure InitUsb;
// To be called once in the initialisation phase of the software

function USB_MSD_Device_Ready: boolean;
// Returns TRUE if an USB MSD device is attached and accessible

function USB_MSD_Read_Sector (Sector: DWord; var Buffer: array[512] of byte):
boolean;
// Reads one sector (number = "Sector")from the USB MSD device into "Buffer"
// Returns TRUE if successful, otherwise returns FALSE

function USB_MSD_Write_Sector(Sector: DWord; var Buffer: array[512] of byte):
boolean;
// Writes one sector (number = "Sector")from "Buffer" into the USB MSD device
// Returns TRUE if successful, otherwise returns FALSE

procedure USB_MSD_Device_Vendor (var S: string[8]);
// Returns the vendor info from the "Inquiry" data

procedure USB_MSD_Device_Product(var S: string[16]);
// Returns the product info from the "Inquiry" data

procedure USB_MSD_Device_Version(var S: string[4]);
// Returns the version info from the "Inquiry" data

procedure USB_MSD_Device_Capacity(var _NrBlocks: DWord; _BlockSize: word);
// Returns the number of blocks (or sectors) in _NrBlocks and
// the block (or sector) size in bytes in _BlockSize

var USB_Error: word;
// Returns the USB Error
// in case USB_MSD_Device_Ready stays FALSE
// The USB_Error signals one error per bit (see below for the possible values)

var MSD_Error: word;
// Returns the MSD Error (see below for the possible values)
// in case USB_MSD_Device_Ready stays FALSE
// The MSD_Error signals one error per bit (see below for the possible values)
```

```
const // Error constants

     // USB_Error constants
      USB_DEVICE_DESCRIPTOR_ERROR              = $001; // bit 0
      USB_CONFIG_DESCRIPTOR_ERROR              = $002; // bit 1
      USB_INTERFACE_DESCRIPTOR_ERROR           = $004; // bit 2
      USB_INTERFACE_DESCRIPTOR_CLASS_ERROR     = $008; // bit 3
      USB_INTERFACE_DESCRIPTOR_SUBCLASS_ERROR  = $010; // bit 4
      USB_INTERFACE_DESCRIPTOR_PROTOCOL_ERROR  = $020; // bit 5
      USB_ENDPOINT_DESCRIPTOR_ERROR            = $040; // bit 6
      USB_ENDPOINT_DESCRIPTOR_ATTRIBUTES_ERROR = $080; // bit 7
      USB_ENDPOINT_DESCRIPTOR_PACKETSIZE_ERROR = $100; // bit 8

      // MSD_Error Constants
      MSD_BLOCK_SIZE_ERROR                     = $001; // bit 0
      MSD_READ_SECTOR_ERROR                    = $002; // bit 1
      MSD_WRITE_SECTOR_ERROR                   = $004; // bit 2
      MSD_READ_CAPACITY_ERROR                  = $008; // bit 3
      MSD_READ_INQUIRY_ERROR                   = $010; // bit 4
      MSD_NOT_READY_ERROR                      = $020; // bit 5
```

[end of document]