USB MSD Host Library

2017-01-03

1 Content

2 Introduction

This is a library for making an **USB host** capable of reading and writing sectors from/to **USB memory sticks** (pen drives) 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 also 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...)

The library consists of 2 files:
"`USB_HOST_MSD_Library`" and "`USB_HOST_Common_Library`". The first one, which has to be in a uses clause in your project, uses in turn the second one. Both should be entered in the project manager.

3 Usage in a program

3.1 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, UartUtils;
...

begin
  { Main program }

  InitMain;

  InitUsb;                              // <--- initialisation of the library
  repeat
  until USB_MSD_Device_Ready or         // <--- test for readiness 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 readiness (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.
“InitUsb” has in principle to be called only once, all USB activity is handled under interrupt, also attachment and detachment of the device.

3.2 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.
```

4 Identification of the connected MSD device.

The device connected to the USB host can be identified by in total 6 identifiers (3 numerical and 3 strings). All identifiers are present in the so called USB Devicedescriptor or pointed to by it.
All identifiers are only valid if “`USB_MSD_Device_Ready`” is true.

4.1 Numerical identifiers

The following 3 functions return each a word value:

```
function USB_Device_Vendor : word;

function USB_Device_Product: word;

function USB_Device_DeviceRelease: word; // BCD value
```

You can use these values to check which USB MSD device is connected to the host.

4.2 String identifiers

These 3 identifiers are only exported in the library interface if the “`USB_HOST_STRINGS`” compiler directive in your project is defined. The best way to do this is adding the directive to a .pld file which is part of the project. Make sure the USB host library is also part of the project, it is compiled depending on that directive.

If the directive “`USB_HOST_STRINGS`” is defined in the project, then the program using the USB MSD Host library must define 3 strings in which the identifiers will be presented:

```
{$IFDEF USB_HOST_STRINGS}
var  USB_Device_ManufacturerString,
     USB_Device_ProductString,
     USB_Device_SerialNrString: string[USBHostStringSize];
{$ENDIF}
```

In case a string identifier is not defined in the USB device an empty string will be the result.

5 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 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

6 Interface of the Library

6.1 USB_HOST_MSD_Library

```
uses USB_HOST_Common_Library;

// ------------------------------------------------------------------------
// interface
// ------------------------------------------------------------------------

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; var _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)

function USB_Device_Vendor : word;
// Returns the USB Device Vendor numerical value. Only valid if
USB_MSD_Device_Ready

function USB_Device_Product: word;
// Returns the USB Device Product numerical value. Only valid if
USB_MSD_Device_Ready

function USB_Device_DeviceRelease: word; // output is in BCD!
// Returns the USB Device Release numerical value. Only valid if
USB_MSD_Device_Ready


{$IFDEF USB_HOST_STRINGS}
const USBHostStringSize = __USBHostStringSize;
{$ENDIF}

// ------------------------------------------------------------------------
// ---------------- USB and MSD Error Constant Definitions ----------------
// ------------------------------------------------------------------------

const
      // 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

const // 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
```

6.2 USB_HOST_Common_Library

```
{$IFDEF USB_HOST_STRINGS} const __USBHostStringSize = 30; // string size to be used
for the following 3 strings
var  USB_Device_ManufacturerString,
     USB_Device_ProductString,
     USB_Device_SerialNrString: string[__USBHostStringSize]; external;
// content of the strings is only valid if USB_MSD_Device_Ready is true
{$ENDIF}
```

[end of document]