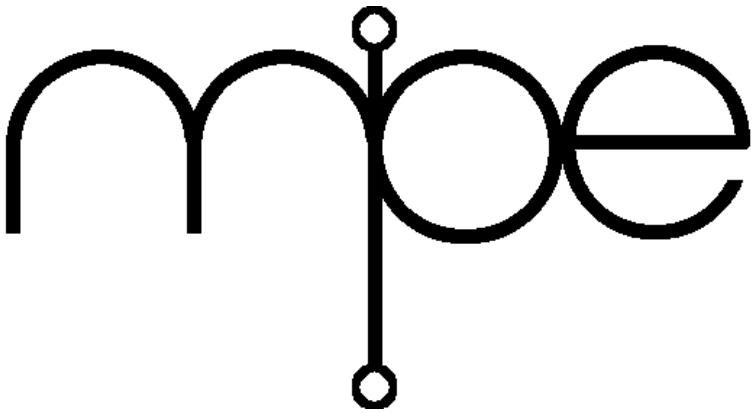
USB Target Code

USB code v3.1, compiler v7.5

Microprocessor Engineering Limited



Published by Microprocessor Engineering

USB Target Code v3.1
User manual
Manual revision 3.1
12 July 2018

Software
Software version 3.1

For technical support
Please contact your supplier

For further information
MicroProcessor Engineering Limited
133 Hill Lane
Southampton SO15 5AF
UK

Tel: +44 (0)23 8063 1441
e-mail: mpe@mpeforth.com
tech-support@mpeforth.com
web: www.mpeforth.com

Table of Contents

1 Introduction

Version 3 of the USB target code has needed a number of detail changes to cope with the addition of a hardware layer for STM32F0x2 devices. The ST USB engine is very different from those of the NXP LPC devices, especially in the interrupt handling.

Version 2 of the USB target code has been refactored, and is not directly compatible with the code in version 1. The major difference is that code dealing with Endpoint 0 has been moved to a separate file which is compiled **after** the class drivers. Other changes include refactoring of the primary USB interrupt to reduce interrupt overhead, provision for DMA-driven systems (recommended), improved error detection (especially for drives) and additional hardware drivers.

To avoid the need for custom host drivers, several standard USB classes are provided. The most common of these are the Mass Storage Class (MSC) used by memory sticks, and the Communications Device Class (CDC) used by USB serial converters which are often referred to as VCOM (Virtual COM) devices. Examples of both classes are provided.

The USB system presented here is built in four main layers

- USB hardware driver - if you you write a new hardware driver, do not change the interface. See *UsbHwSTM32F0x2.fth* for STM32F0x2 devices. See *UsbHwLPC2xxx.fth* for an example for NXP LPC2xxx devices and *UsbHwLPC1xxx.fth* for some LPC1xxx devices. Minor hardware differences are described in the *devxxxx.fth* files, which handle things like pin selection, USB clocking, and USB connect and disconnect. Other drivers are provided in the <*CPU*>*Drivers* folder.
- Base layer - portable code providing generic facilities for the class drivers. See *UsbBase.fth*.
- Class drivers - USB systems, such as memory sticks, use standardised protocols which are referred to as class drivers. By using class drivers you do not have to provide custom drivers on the host PC. An example is the Mass Storage Class (MSC) driver in *MscDrv.fth*.
- Endpoint 0 layer - handles the USB control through endpoint 0, and despatches interrupts to the handlers for the other endpoints. This code is in *UsbEP0.fth*.

In addition to these layers, the code contains configuration options in *MSCconfig.fth* and *CDCconfig.fth*. You can use these as models for other configurations. A sample composite device is configured in *CdcMscConfig.fth*, which also contains code for handling USB serial numbers.

The Mass Storage Class depends on the disk interface used by the *FATfiler* code. The standard example is for an LPC2148 (Olimex P2148 hardware) using the SPI driver in *ARM\Drivers\spiLPC2148hard.fth*. You can read, write and format the drive from a host PC.

The Communications Device Class provides a Generic I/O device which can be used in the same way as any other device. You can even use it as the Forth console. Under Windows, no driver is required, but a ".inf" file is required. An example for NXP processors is provided in `Examples\USB\mpevcom.inf`. Note that the examples use a sample VID and PID pair. For distribution of your products, you must obtain a VID (Vendor ID number) from `http://www.usb.org`.

The code is entirely written in high level Forth for 32 bit systems. The primary tests for hardware

and class drivers are performed using Windows 7 and Windows XP, service pack 3. Windows XP Service Pack 3 is required for reliable use of composite devices that rely on Windows XP services.

1.1 Setting up for USB development

USB is one of those protocols that doesn't work until almost everything works. You **need** a protocol analyser. Just as nobody does anything serious with TCP/IP without learning to use Wireshark (was Ethereal), so nobody who is being paid to develop USB systems can do without a USB analyser. We used a Beagle USB 12 from TotalPhase Systems (`http://www.totalphase.com`) and it was worth every penny.

Every silicon vendor provides sample USB sample code. Don't expect it to be pretty, well commented or bug-free.

There are a number of books about USB device development. The ones we have used are:

USB Design by example, John Hyde,

USB Mass Storage, Jan Axelson.

Jan Axelson has a number of USB books available. Her website is at `http://www.lvr.com`.

1.2 MSC Class driver

Most USB devices are single application. However, most embedded systems use SD or CompactFlash cards for mass storage, and want to use the Mass Storage drive to access the files on the SD card. To do this requires a FAT file system as well as a USB driver. The majority of memory sticks use FAT16 or FAT32 file systems. These are provided by the MPE *FATfiler* which comes with all Developer editions of MPE Forth cross compilers for 32 bit targets.

Most sample MSC drivers do everything in the interrupt routine, including the drive sector read/write routines. This causes excessive time in the interrupt routine, preventing your real-time task from operating properly. To overcome this, we have split the load into a USB interrupt handler and a task controlled by the interrupt handler. The MSC driver only needs raw sector read/write routines, which must be interlocked, e.g. by a semaphore, with the other task(s) that use the sector interface.

Despite all this, we have seen the USB interrupt routine use 80% of the CPU time at peak USB load on a 60MHz ARM interfacing to a slow (2Mbps) SPI driver for an SD/MMC card. For good real-time performane, tune your card and USB hardware routines. Use DMA hardware wherever possible after you have your hardware running reliably. Because of the number of different hardware systems that we have to support, we at MPE have to consider code portability a high priority.

When using SDHC cards (4Gb and above), note that these will normally be formatted using FAT32, for which the FAT table is huge. Windows appears to read the whole FAT table when a drive is recognised. With USB full-speed devices (12 Mbps), this takes a long time. Start up and format for FAT16 are **much** quicker. As an example, the FAT size for a 512Mb FAT16 card

is 242 sectors with 16 sectors per cluster, whereas for a 4Gb FAT32 card, the FAT size is 7658 sectors with 8 sectors per cluster.

Windows does not require a .INF file for MSC devices.

1.3 CDC Class Driver

The Communications Device Class (CDC) can be used for a wide range of devices including modems and Ethernet simulations. The code we supply only supports the ACM subclass, which is the one used for serial port emulations and modems. Only a limited set of control functions are implemented beyond providing a stub. If you need them, expand them to suit your requirements.

There are two USB interfaces, a control interface with an interrupt in endpoint, and a data interface with a bulk in and and bulk out endpoint per COM channel.

The COM channel(s) are implemented as the ends of `CQUEUE` structures. These queues should be large enough to hold the number of bytes accumulated during a USB poll interval. The poll interval is defined in the endpoint descriptor for the control interface. By default, this is set to 1, indicating a poll interval of 1 ms.

Under Windows, a .INF file is required for any USB device that includes a CDC class driver. The PID and VID (see below) must match those in your USB device descriptors, as must the manufacturer string.

1.4 Composite devices

The file *CdcMscConfig.fth* contains a USB configuration for a composite Communications Device Class (CDC) and Mass Storage Class (MSC) application.

The trick for composite devices is to use an Interface Aggregation Descriptor (IAD) for each instance of any class with more than one interface, e.g. CDC. Read the source code in *CdcMscConfig.fth* for the gory details!

Running composite devices on Windows without using a custom driver can be problematic. Vista is fine, but XP SP2 is not. For XP, either upgrade to SP3 or install hotfixes. It seems you need

- Hotfix KB935892 (which brings usbccgp.sys 5.1.2600.3116)
- Hotfix KB918365 (which brings usbser.sys 5.1.2600.2930)

1.5 USB Vendor IDs

USB devices are identified by a Vendor ID (VID), Product ID (PID) and a serial number. The primary source of Vendor IDs is `http://www.usb.org`. If you only need a few, it may be possible to buy a block from a third party, e.g. `http://www.voti.nl/shop/catalog.html?USB-PID-10`.

According to the USB rules, VIDs are not transferable without written permission. For development, MPE uses VID=0D59 with PID=FFF0..FFFF. It is not our intention to use the PID range FFF8..FFFF.

1.6 Compiling the code

Edit the file *xxxConfig.fth* as required. The default is for a Mass Storage Class driver. Depending on the hardware and its driver, you may have to force DMA buffers into a particular memory range. The MSC class driver requires sector read and write routines and the FAT filing system.

```
  include %UsbDir%\UsbDefs          \ USB definitions/equates
  include %UsbDir%\MscConfig        \ configuration file
\  include %UsbDir%\CDCconfig       \ USB configuration file
\  include %UsbDir%\CdcMscConfig    \ USB configuration file
  include %UsbDir%\UsbHwLPC2xxx     \ USB hardware layer for LPC2xxx
  include %UsbDir%\UsbBase          \ USB base operations
CDC? [if]
  include %UsbDir%\CdcDrv           \ Communications Device Class
[then]
MSC? [if]
  include %UsbDir%\MscDrv           \ Mass Storage Class
[then]
  include %UsbDir%\UsbEp0           \ USB Endpoint 0 layer
```

After compiling and flashing the code, power up the target, and type:

```
 startUSBsys
```

to start the USB and mass storage system. You should then be able to transfer files from the PC to the target. To see them, return to the Forth target prompt and type:

```
  dir
```

which will provide a simple directory listing.

1.7 Testing with operating systems

The code here has been tested with Windows 7 and 10, Mac macOS 10.13 High Sierra, Debian Linux v16.04 for x86_64 Linux and Raspbian Jessie for ARM Linux.

Windows takes a lowest common denominator approach to the standard USB classes in order to operate with the largest number of devices. It is probably easiest to get a device working under Windows than any other operating system. However, getting something working under Windows is no guarantee that your device is standards-compliant or that it will work with other operating systems.

Linux expects that what is provided operates to specification and will reject non-compliant devices. However, it is reasonably tolerant of missing features.

Mac OS X is really fussy and makes few concessions. However, there are good USB analysis programs to help debugging. Composite devices can be difficult to write descriptors for. IAD support was only introduced in OS X 10.5.6. At least one of our clients detects an OS X host (first setup packet has wLength=8) and presents different USB descriptors for OS X. Note that recent Macs may no longer support Full Speed devices (the majority of embedded devices) and that a USB 2.0 hub is required. If you are powering the device from the hub, a powered hub is recommended.

2 Generic USB definitions

The file *UsbDefs.fth* contains generic USB defines and tools used to insulate the hardware drivers *UsbHwxxxx.fth* from the core layers in *UsbBase.fth* and *UsbEP0.fth*.

```
0 equ USBMin?   \ -- flag
```
If your control file sets this equate non-zero, much of the USB code is compiled without headers.

```
struct /USBSetupPacket  \ -- len
```
USB Default Control Pipe Setup Packet structure.

```
  byte  usp.bmRequestType       \ bit7=Dir,0=to dev, bit6:5=Type, bit4:0=Recipient
  byte  usp.bRequest
  hword usp.wValue
  hword usp.wIndex
  hword usp.wLength
end-struct
```

```
struct /UsbEpd  \ -- len
```
USB Standard Endpoint Descriptor

```
  BYTE  epd.Len
  BYTE  epd.Type
  BYTE  epd.Addr
  BYTE  epd.Attribs
  HWORD epd./MaxPacket
  BYTE  epd.Interval
end-struct
```

2.1 Support tools

```
: bitn          \ n -- 2^n
```
Generate bit n from n.

```
: lew@          \ addr -- w
```
16 bit unaligned little-endian fetch. USB control data is in little-endian form and may be unaligned.

```
: lew!          \ w addr --
```
16 bit unaligned little endian store. USB control data is in little-endian form and may be unaligned.

```
: lel@          \ addr -- x
```
32 bit unaligned little-endian fetch.

```
: lel!          \ x addr --
```
32 bit unaligned little endian store.

```
: lew@          \ addr -- w
```
16 bit fetch. USB control data is in little-endian form and may be unaligned.

```
: lew!          \ w addr --
```
16 bit store. USB control data is in little-endian form and may be unaligned.

```
: lew,          \ w --
```
Lay unaligned little-endian 16 bit data.

`: lel@ \ addr -- x`
32 bit unaligned little-endian fetch.

`: lel! \ x addr --`
32 bit unaligned little endian store.

`: lel, \ x --`
Lay unaligned little-endian 32 bit data.

`: bew@ \ addr -- w`
Unaligned big-endian 16 bit fetch.

`: bew! \ w addr --`
Unaligned big-endian 16 bit store.

`: bew, \ w --`
Lay unaligned big-endian 16 bit data.

`: bel@ \ addr -- x`
Unaligned big-endian 32 bit fetch.

`: bel! \ x addr --`
Unaligned big-endian 32 bit store.

`: bel, \ x --`
Lay unaligned big-endian 32 bit data.
Strings are laid as little-endian 16 bit Unicode, preceded by a count byte and a string type marker.

`: resetUIFs \ --`
Start the USB interface sequence numbering at 0.

`: NextUIF: \ "<name>" --`
Create an equate with the next USB interface number.

`: #UIFs \ -- u`
Returns the number of interfaces defined so far.

2.2 Hardware isolation

The five `VALUE`s below are used to provide status information for the core layer, but are actually contained here in *UsbDefs.fth* to avoid forward references and to permit easy reference by the class drivers. Note that the endpoint status values are bit masks in a hardware independent form. See `EP>mask` below.

`0 value DevAddr \ -- x`
Set to $80+u where u is the new device address. Used by low level code to set the USB device address at a time suitable for the hardware. After setting `UsbDevAddr` below, `DevAddr` is reset to 0.

`0 value UsbDevAddr \ -- u`
USB device address. Set by the host.

`0 value UsbConfig \ -- x`
USB Configuration set. Non-zero when the USB device has been configured.

`0 value UsbEPmask \ -- x`
Endpoint mask. Bits are set when the endpoint is enabled. Used by the core layer.

`0 value UsbEPhalt \ -- x`
Endpoint halt status. Bits are set when the endpoint is halted/stalled.

`0 value Usb#IfSet \ -- n`
Number of interfaces set.

`: EP>mask \ ep# -- mask`
Decode an endpoint number to a bit mask such that if bit 7 is clear (OUT endpoints), the bit is in the low 16 bits, and if set (IN endpoints) in the upper 16 bits.

2.3 Linked lists

Linked lists are used by the USB driver to avoid forward references to higher level code, especially various reset actions. Note that the chains are managed at compile time. The linked list uses facilities provided in *Common\Kernel62.fth*.

`: ExecChain \ anchor --`
Execute the contents of chain with the following structure:

```
link | xt | ...
```

Each word that is run has the stack effect

```
link -- link
```

Where link is the address of the link field in the structure. Thus, data that follows the xt can easily be accessed.

`create UsbResetChain \ -- addr`
Anchors the chain of words executed at device reset.

`create UsbFrameChain \ -- addr`
Anchors the chain of words executed in the frame interrupt.

`: link, \ addr --`
Extend chain anchored at *addr*.

`: AtUSBreset \ xt --`
Add the word whose *xt* is given to the USB reset chain.

`: AtUSBframe \ xt --`
Add the word whose *xt* is given to the USB frame interrupt chain. Words in this chain are mostly used to enable Bulk or Interrupt endpoint NAK interrupts. Careful use of this chain can significantly reduce USB interrupt handling overhead.

3 USB composite CDC and MSC configuration

The file *CdcMscConfig.fth* contains a USB configuration for a composite Communications Device Class (CDC) and Mass Storage Class (MSC) application. For CDC and MSC only applications see *CdcConfig.fth* and *MscConfig.fth*.

Running composite devices on Windows without using a custom driver for Windows can be problematic. Windows 7, Vista and XP SP3 are fine, but XP SP2 is not. For XP, either upgrade to SP3 or install hotfixes. It seems you need:

- Hotfix KB935892 (which brings usbccgp.sys 5.1.2600.3116)
- Hotfix KB918365 (which brings usbser.sys 5.1.2600.2930)

We installed SP3 for testing and had to copy the new version of *usbser.sys* manually from *SP3.cab*. Use Windows search and copy the new version to replace the one in *Windows\System32\Drivers*.

The trick for composite devices is to use an Interface Aggregation Descriptor (IAD) for each instance of any class with more than one interface, e.g. CDC. Read the source code for the gory details!

You also need to take care with the start up code for composite devices. USB device start up consists of three steps.

1. Initialise data buffers, some of which may be in `UDATA` section, and so will not be initialised by the default MPE initialisation code. This code should also be part of the USB reset action for the class or interface.
2. The primary initialisation of the USB hardware and code is performed by `InitUSB`.
3. Then perform the USB bus connection process using `USBconnect`.

Your start up code for a composite device then consists of

```
initA initB ... InitUSB USBConnect
```

Versions of this stack from November 2010 onwards include in *UsbEP0.fth* the word `startUSBsys` which performs these operations.

3.1 Primary Configuration

Note that interface numbers must be a contiguous set starting at 0.

`0 equ dbgUHW?   \ -- flag`
Set true to get debug information from the USB hardware layer. Some of the debug output **must** be done using polled serial drivers.

`0 equ dbgCore?  \ -- flag`
Set true to get debug information from the USB core layer. Some of the debug output **must** be done using polled serial drivers.

```
0 equ UsbPowered?       \ -- 0/1 ; bus powered?
```
Set to one if bus powered.

```
1 equ UsbDMA?           \ -- 0/1 ; uses DMA?
```

Set to one if any endpoints other than EP0 use DMA. At the moment we assume that EP0 does not use DMA.

```
#64 equ /maxpacket0     \ -- u
```
Maximum size of packets on endpoint 0.

```
#64 equ /maxpacket      \ -- u
```
Maximum size of packets on endpoints other than 0.

```
#32 equ #EPs    \ -- u
```
Number of endpoints we are are going to deal with. On a 32 bit CPU, there are 32 bits per word. We need separate bits for IN and OUT, so there's a practical maximum here of 32, and USB supports 16 endpoints, each with IN and OUT capability. Endpoint 0 is always IN and OUT capable.

```
4 equ #USBifs   \ -- u
```
Number of interfaces we can expand to eventually.

```
0 equ HID?      \ -- flag
```
True if a HID device, e.g. mouse, is to be supported.

```
1 equ CDC?      \ -- flag
```
True if a communications device class (CDC) is to be supported. The most common of these is a USB serial port.

```
1 equ MSC?      \ -- flag
```
True if a mass storage device is to be supported.

```
0 equ AUDIO?    \ -- flag
```
True if an audio device is to be supported.

```
resetUIFs
```
Force the interface numbers to start at zero.

```
NextUIF: CdcCIf#        \ -- b
```
Interface number of the CDC control interface.

```
NextUIF: CdcDIf#        \ -- b
```
Interface number of the CDC data interface.

```
$81 equ CDCIinEP        \ -- ep
```
Optional Interrupt IN event notification endpoint (CIF).

```
$85 equ CDCBinEP        \ -- ep
```
CDC Bulk IN endpoint (DIF).

```
$05 equ CDCBoutEP       \ -- ep
```
CDC Bulk OUT endpoint (DIF).

```
1 equ dbgCDC?           \ -- flag
```
set non-zero to get debug information from the CDC driver.

```
CDCIinEP EP>mask equ CDCIinEPMask       \ -- mask
```
Core layer bit mask for the CDC Interrupt IN endpoint. Used to avoid run-time calculation.

```
CDCBinEP EP>mask equ CDCBinEPMask       \ -- mask
```
Core layer bit mask for the CDC bulk IN endpoint. Used to avoid run-time calculation.

```
CDCBoutEP EP>mask equ CDCBoutEPMask     \ -- mask
```
Core layer bit mask for the CDC bulk OUT endpoint. Used to avoid run-time calculation.

```
NextUIF: MscIf#          \ -- b
```
Interface number of the mass storage class.

```
$82 equ MscBinEP         \ -- ep
```
MSC Bulk IN endpoint.

```
$02 equ MscBoutEP        \ -- ep
```
MSC Bulk OUT endpoint.

```
0 equ dbgMSC?            \ -- flag
```
set non-zero to get debug information from the MSC driver.

```
MscBinEP EP>mask equ MscBinEPMask        \ -- mask
```
Core layer bit mask for the MSC bulk IN endpoint. Used to avoid run-time calculation.

```
MscBoutEP EP>mask equ MscBoutEPMask      \ -- mask
```
Core layer bit mask for the MSC bulk OUT endpoint. Used to avoid run-time calculation.

3.2 USB Descriptors

This section contains the USB descriptors. There is a descriptor for the device, for its configuration, and for various strings. There are various other descriptors, both general and class class specific. Their use is absurdly badly documented and the actual requirement as to which are required is often operating system dependent. If it works with Linux, it will (probably) work with everything else.

Strings are accessed using a "string index". Apart from zero, which has a predefined meaning to define the language, string indices (1..255) are defined by the application. Here the indices are the offset in bytes from the start of a string table. If you need more than 255 bytes of string descriptors, change the index to be the offset into a table of string descriptor addresses.

```
Create CfgDesc  \ -- addr
```
The configuration descriptor has a primary configuration, three interfaces, and five endpoints. If there is more than one configuration, additional ones must follow the first one.

```
create StrDesc  \ -- addr
```
String descriptor and following strings.

- Str0 - identifies the language. This is usually left alone.
- Str1 - the USB serial number, usually a 16 character string. In the default code, the first 8 bytes are fixed and the second 8 bytes can be derived from a 32 bit number. Str2 - the product identification string. Str3 - the manufacturer string.

```
create DevDescDefault    \ -- addr
```
The device descriptor for Windows and Linux.

```
create DevDescOSX        \ -- addr
```
The device descriptor for OSX.

```
0 value DevDescReqLen    \ -- u
```
Holds request length of first device descriptor after reset. It is used to auto-select the device descriptor

```
:noname 0 to DevDescReqLen  ; AtUSBreset
```
Performed at USB reset.

```
: DevDesc        \ -- addr
```

Return the address of the device descriptor. This word uses a kluge to provide operation on Windows, Linux, and versions of OS X before 10.5.6 which recognises the IAD descriptor.

3.3 Serial number formatting

If the word `unit#` is defined, it should return a 32 bit unit serial number. This is used by the USB code to return a serial number to the host. If the word `unit#` has not been defined, a default fixed serial number is returned.

`#34 buffer: UTFbuf  \ -- addr`
Holds a string descriptor with up to 16 little-endian Unicode 16-bit characters.

`: usbSerialNum  \ -- caddr len`
Returns the default device USB serial number if `unit#` is not defined.

`: usbSerialNum  \ -- caddr len`
Returns the default device USB serial number if *unit#* is not defined.

`: USBStr@  \ index -- caddr len`
Return the USB string descriptor corresponding to the given string index. By convention the following order is used:

`0`	language identifier - do not change
`1`	serial number
`2`	product
`3`	manufacturer

If you configure your products with serial numbers extracted from non-USB strings, e.g. MAC numbers, modify this word to perform the string extraction as required.

4 USB hardware layer for LPC1xxx

The file *UsbHwLPC1xxx.fth* contains the USB hardware layer for NXP LPC1xxx and some LPC4xxx devices. The file *UsbHwLPC2xxx.fth* is very similar and contains code for NXP LPC2xxx devices.

DMA operation is supported if the equate `usbDMA?` is set non-zero in the USB configuration file, e.g. *MscConfig.fth*. Use of DMA is **strongly** recommended. For most real-time applications the benefit of DMA operation is the considerable reduction of interrupt overhead at the expense of code size. If even this is unacceptable, consider the following solutions.

- Permit nested interrupts and reduce the priority of the USB interrupt handler.
- Do not use a USB interrupt handler at all. Define a USB task and poll the handler.

4.1 Configuration

`: EPaddr \ ep# -- physaddr`
Translate the endpoint number to a physical address. Bit 7 of the endpoint number *ep#* (0..15) is set if the write buffer is required.

`1 equ usbDMA? \ -- flag`
Set non-zero if the USB hardware is to use DMA for bulk and isochronous transfers. The definition in this file is only used if not previously defined. Use of DMA is **strongly** recommended.

4.2 DMA operations

The USB hardware contains its own DMA controller. Whether this is used is set by the equate `usbDMA?` in the USB configuration file. Each endpoint that uses DMA **must** define at least one DMA descriptor and its type.

`32 cells buffer: UDCA \ -- addr`
USB DMA communication area. `UDCA` is only defined here if it has not already been defined.

`create UDCAcfg \ -- addr`
A table of device descriptors used to initialise the `UDCA`.

`create EpDmaCfg \ -- addr`
A table containing a byte per endpoint. The byte indicates how the endpoint is used with DMA:

`0`	no DMA
`16`	Bulk or interrupt endpoint
`20`	Isochronous endpoint.

DMA use is set at compile time. The following `INTERPRETER` words are used.

`: BulkDD: \ ep# -- ; e.g. $0x or $8x`
Creates a bulk DMA Descriptor in the `URAMP` section, and adds it to the `UDCAcfg` template.

`: IntDD: \ ep# -- ; e.g. $0x or $8x`
Creates an interrupt DMA Descriptor in the `URAMP` section, and adds it to the `UDCAcfg` template.

`: IsoDD: \ ep# -- ; e.g. $0x or $8x`
Creates an Isochronous DMA Descriptor in the `URAMP` section, and adds it to the `UDCAcfg` template.

`: resetDD \ physaddr dd --`
Reset a device descriptor to No_Packet state. The DMA buffer address, length and max packet size are all zero. See LPC23xx User Manual, Chapter 13, 15.5.6.

`: initEPdma \ physaddr --`
Set up the DMA for the given endpoint (0..31) and apply `resetDD`.

`: initUSBdma \ --`
Initialise the USB DMA.

`: setEPdma \ caddr len ep# --`
Set the next transfer for an endpoint ($0x/$8x).

`: #EPtrans \ ep# -- len`
Return the number of bytes transferred on this endpoint. The return value is only valid after a transfer is complete.

`: enEPdma \ ep# --`
Enable endpoint DMA.

`: disEPdma \ ep# --`
Disable endpoint DMA.

4.3 Data and tools

`create ^USBdev \ -- addr`
Holds the address of the USB device peripheral.

`: EpIntSt@ \ -- x`
Read the endpoint interrupt status register.

`: EpIntClr! \ x --`
Write the endpoint interrupt clear register.

`: DevIntSt@ \ -- x`
Read the device interrupt status register.

`: CmdData@ \ -- x`
Read the command data register.

`: EpRe! \ x --`
Write the EpRe register.

4.4 Protocol Engine commands

The USB Protocol Engine is triggered by writing to the `UsbDevCmdCode` register. Commands are in the form:

```
00ppcc00
```

where *pp* is the command phase and *cc* is the command code. For more details, read the source code.

Commands for the LPC2xxx USB controller are 32 bit items of the form $aabb:0500. When data is also written the 8 bit data is written as a 32 bit item of the form $00bb:0100. When a data byte is to be read, the read is performed by reissuing the command in the form $aabb:0200, and waiting for the CDFULL interrupt rather than the CCEMTY interrupt.

```
 : WrCmd         \ cmd --
```
Write *cmd* to the CmdCode register.

```
: WrCmdDat       \ cmd val --
```
Write *cmd* followed by *val* to the CmdCode register.

```
: RdCmdDat       \ cmd -- val
```
Write a command and read the data.

4.5 Handling endpoints

This section contains an array of endpoint handlers and default actions for other operations which are hardware dependent.

```
create EPxts     \ -- addr
```
Holds the xts of the 16 endpoint handler words. The action of each handler must have the stack effect below:

```
  event --
```

```
: EPhandler      \ event u --
```
Run the endpoint handler associated with endpoint *u*, where endpoints are numbered 0..15. Each endpoint handler is passed an event type which allows it to determine what it has to do. The action of each handler must have the stack effect below.

```
  event --
```

```
: SetEPhandler   \ xt ep# --
```
Install the endpoint handler for endpoint *ep#*. Used during interpretation.

```
0 value UsbDevSt         \ -- x
```
USB device Status.

```
: USBsuspend     \ --
```
Suspend operations are performed by the hardware. This is a hook for installing actions such as toggling an LED.

```
: USBresume      \ --
```
Resume operations are performed by the hardware. This is a hook for installing actions such as toggling an LED.

```
: USBconnect     \ --
```
Connect to the USB bus.

```
: USBdisconnect \ --
```
Disconnect from the USB bus.

```
: USBWakeUp      \ --
```
Called automatically on USB Remote Wakeup.

```
: USBSetAddress \ addr --
```
Set the USB assigned address in the protocol engine.

```
: WaitEPrlzed    \ --
```
Wait for the EP_RLZED interrupt status and clear it.

```
: USBConfigure   \ cfg -- ; 0=unconfigure
```
Set the SIE configuration state.

```
: USBConfigEP    \ desc --
```

Configure USB Endpoint according to Descriptor.

`: USBConfig2EP  \ desc ep# -- desc`

Configure USB Endpoint according to Descriptor and EndPoint.

`: USBDirCtrlEP  \ dir -- ; 0=out`

This word is a dummy for this hardware.

`: USBEnableEP   \ ep# -- ; bit7=dir, bits3:0=num`

Enable the endpoint.

`: USBDisableEP  \ ep# -- ; bit7=dir, bits3:0=num`

Disable the endpoint.

`: USBResetEP    \ ep# -- ; bit7=dir, bits3:0=num`

Reset the endpoint.

`variable SieModeMask    \ -- addr`

A shadow variable that holds the last mode mask set.

`: setSIEmode    \ mask --`

Set the USB device mode register in the SIE. A shadow variable is used to hold the current state.

`: +BulkNAKin    \ --`

Enable NAK interrupts on bulk IN endpoints. All other SIE mode bits are left alone.

`: -BulkNAKin    \ --`

Disable NAK interrupts on bulk IN endpoints. All other SIE mode bits are left alone.

`: +BulkNAKout   \ --`

Enable NAK interrupts on bulk OUT endpoints. All other SIE mode bits are left alone.

`: -BulkNAKout   \ --`

Disable NAK interrupts on bulk OUT endpoints. All other SIE mode bits are left alone.

`: USBSetStallEP \ ep# -- ; bit7=dir, bits3:0=num`

Stall the endpoint. This word **must** clear the relevant bit in `UsbEpHalt`.

`: USBClrStallEP \ ep# -- ; bit7=dir, bits3:0=num`

Clear the endpoint stall. This word **must** clear the relevant bit in `UsbEpHalt`, and should ensure that bulk IN/OUT NAKs cause interrupts.

`code portl>     \ caddr len port --`

Read the port into the the memory block a cell at a time. If *caddr* is aligned, the data is stored cell by cell, otherwise it is assembled a byte at a time before being sent to the port.

`: USBReadEP     \ ep# addr -- #read ; bit7=dir, bits3:0=num`

Read endpoint data from endpoint *ep#* to *addr*. It will be faster if *addr* is aligned. Return the number of bytes read. Note that data is transferred in units of four bytes, so the buffer must be large enough to accommodate up to three extra bytes.

`code >portl     \ caddr len port --`

Transfer the memory block to the port a cell at a time. If *caddr* is aligned, the data is fetched cell by cell, otherwise it is assembled a byte at a time before being sent to the port.

`: USBWriteEP    \ ep# addr len -- cnt`

Write data *addr/len* to host from endpoint *ep#*. Return the number of bytes written.

4.6 Initialisation

`: hwUSBreset \ --`
Reset the USB device hardware after access has been enabled.

`: USBswReset \ --`
The action performed for a software reset of the USB, either at reset or in response to a USB reset.

4.7 Endpoint interrupt handling

`: waitCmdData \ -- x`
Wait for CDFULL and read the data

`: doEPslow \ --`
Process an EP_SLOW interrupt.

`: doDmaEOT \ --`
Handle DMA End Of Transfer interrupt.

`: doDmaNDDReq \ --`
Handle DMA New DD request interrupt.
USB Ssytem Errors are AHB bus errors. As yet, the AHB bus configuration registers are undocumented. Consequently there is no point in trying to handle these errors, and the code framework to do so is commented out.

`: doDmaSysErr \ --`
Handle DMA System Error interrupt.

`: doUSBint \ disr -- disr'`
The primary word called from `USBinterrupt` below. Processes DEV_STAT, EP_SLOW and FRAME interrupt sources, ignoring and clearing all others. If DMA is enabled, DMA interrupts are also processed.

`: USBinterrupt \ --`
The USB interrupt handler. It calls `doUSBint` above to permit better factoring.

`' USBinterrupt USB_vec# EXC: USB_ISR \ -- addr`
The entry point for the USB interrupt.

`: USBintInit \ --`
Initialise USB hardware interrupt handler.

`: InitUSB \ --`
Initialise the USB hardware.

`: startUSB \ --`
Start the USB system. This word gives more control to devices that have VBUS control. If such devices are rebooted without power down, they may/will need to perform a `USBdisconnect` operation as part of the start up sequence. A disconnect, delay, connect sequence is part of `startUSB`.

4.8 Development and Gotchas

Under some conditions, normally after testing a faulty USB device, the Windows USB system can become faulty. During device development, rebooting your PC will be a common occurrence.

When using console debug messages, note that you must not use serial drivers with queued output. These use `PAUSE` which **must not** be called from an interrupt handler.

5 USB driver for STM32F0x2 parts

The file *UsbHwSTM32F0x2.fth* provides a USB driver for STM32F0x2 devices. Some primitive code was taken from a public-domain source, and updated for correctness and interrupt operation.

`https://github.com/jeelabs/embello/tree/master/explore/1608-forth/suf`

5.1 Endpoint packet memory and control

```
: even          \ addr -- addr'
```
Force input address to be even.

```
: usb-pma       ( pos -- addr ) _USBSRAM + ;
```
Convert an offset/position in the packet RAM to an absolute address.

```
: usb-pma       ( pos -- addr ) _USBSRAM + ;
```
Convert an offset/position in the packet RAM to an absolute address. Compiler macro.

```
: ep-reg        \ ep n -- addr
```
Return the address in packet RAM of reg n (0..3) of ep (0..7)
Each endpoint has an 8 byte control area of four buffers for transmit and receive control.

```
0 equ EPR_ADDR_TX
1 equ EPR_COUNT_TX
2 equ EPR_ADDR_RX
3 equ EPR_COUNT_RX


: ep-addr       ( ep# -- addr )  $0F and  cells _USBdev usbEP0R + + ;
```
Return the address in the peripheral block of the peripheral register for endpoint ep.

```
\ endpoint status values, both TX and RX
0 equ epDISABLED
1 equ epSTALL
2 equ epNAK
3 equ epVALID

\ endpoint type, both TX and RX
0 equ epBULK
1 equ epCONTROL
2 equ epISO
3 equ epINTERRUPT


: rxstat!       \ ep u --
```
Set stat_rx without toggling/setting any other fields.

```
: txstat!       \ ep u --
```
Set stat_tx without toggling/setting any other fields.

```
: rxclear       \ ep --
```
Clear the endpoint receive CTR_RX bit.

```
: txclear       \ ep --
```
Clear the endpoint transmit CTR_TX bit.

```
: ep-reset-rx#  \ ep --
```

Clear the RX count register.

`: ep-reset-tx#  \ ep --`
Clear the TX count register.
Packet memory is used as shown below. Memory use is inefficient as two 64 byte buffers are reserved for each endpoint, whether needed or not.

```
000..03F   Endpoint Table
040..0BF   EP0 buffers, offset $00 for RX, $40 for TX
0C0..13F   EP1 ...
140..1BF   EP2 ...
1C0..23F   EP3 ...
240..2BF   EP4 ...
2C0..33F   EP5 ...
340..3BF   EP6 ...
```

`: EPioBufRX     \ ep# -- pma`
Return offset address of the receive packet memory buffer.

`: EPioBufTX     \ ep# -- pma`
Return offset address of the transmit packet memory buffer

`: set-EPtab-n   \ ep# --`
Set up EP buffer area in the packet memory.

5.2 Handling endpoints

This section contains an array of endpoint handlers and default actions for other operations which are hardware dependent.

`: USBDirCtrlEP  \ dir -- ; 0=out`
This word is a dummy for this hardware.

`: USBResetEP    \ dir -- ; 0=out`
This word is a dummy for this hardware.

`: USBEnableEP   \ dir -- ; 0=out`
This word is a dummy for this hardware.

`create EPxts    \ -- addr`
Holds the xts of the 16 endpoint handler words. The action of each handler must have the stack effect below:

```
event --
```

`: EPhandler     \ event u --`
Run the endpoint handler associated with endpoint *u*, where endpoints are numbered 0..15. Each endpoint handler is passed an event type which allows it to determine what it has to do. The action of each handler must have the stack effect below.

```
event --
```

`: SetEPhandler  \ xt ep# --`
Install the endpoint handler for endpoint *ep#*. Used during interpretation.

`0 value UsbDevSt        \ -- x`
USB device Status.

`: USBsuspend \ --`
Suspend operations are performed by the hardware. This is a hook for installing actions such as toggling an LED.

`: USBresume \ --`
Resume operations are performed by the hardware. This is a hook for installing actions such as toggling an LED.

`: usbHWconnect \ --`
Connect pull up on USB DP line.

`: usbHWdisconnect \ --`
Disconnect pull up on USB DP line.

`: USBconnect \ --`
Connect to the USB bus.

`: USBdisconnect \ --`
Disconnect from the USB bus.

`: USBWakeUp \ --`
Called automatically on USB Remote Wakeup.

`: USBSetAddress \ addr --`
Set the USB assigned address and enable the device. This should be done during a SOF (frame) interrupt. Note that `USBSetAddress` may not actually set the device address, but may just signal that an action must be taken, e.g. in the next frame interrupt

`: USBConfigure \ cfg -- ; 0=unconfigure`
Set the SIE configuration state.

`: cfg-EPnR \ ep# x --`
Configure the USB endpoint register,

`: cfgIso \ desc ep# -- desc`
Configure an isochronous endpoint. UNTESTED.

`: cfgBulk \ desc ep# -- desc`
Configure a bulk endpoint.

`: cfgInt \ desc ep# -- desc`
Configure an interrupt endpoint.

`: USBConfigEP \ desc --`
Configure USB Endpoint register according to Descriptor.

`: USBConfig2EP \ desc ep# -- desc`
Configure USB Endpoint according to Descriptor and EndPoint.

`: USBDisableEP \ ep# -- ; bit7=dir, bits3:0=num`
Disable the endpoint.

`: USBSetStallEP \ ep# -- ; bit7=dir, bits3:0=num`
Stall the endpoint. This word **must** clear the relevant bit in `UsbEpHalt`.

`: epBulk? \ ep# -- flag`
Return true if the endpoint is BULK.

`: USBClrStallEP \ ep# -- ; bit7=dir, bits3:0=num`
Clear the endpoint stall. This word **must** clear the relevant bit in `UsbEpHalt`, and should ensure that bulk IN/OUT NAKs cause interrupts.

`: .ep0/istr \ --`
Display the EP0 endpoint register and the ISTR

`: .ep0-pma \ --`
Display the four EP0 packet memory registers

`: set-EPnR \ x ep --`
Set the USB endpoint register

`: USBswReset \ --`
The action performed for a software reset of the USB, either at reset or in response to a USB reset.

`: ep-setup \ ep# --`
Trigger a SETUP packet handler.

`: ep-out \ ep# --`
Trigger an OUT packet handler.

`: ep-in \ ep# --`
Trigger an IN packet handler.

`: usb-ctr \ istr --`
Given an interrupt status reading, process the packet.

`: usb-poll \ --`
Main USB driver polling routine. Factored this way to allow `EXIT`s from the branches.

`: USBinterrupt \ --`
The USB interrupt handler. It calls `usb-poll` above to permit better factoring. LED1 is enabled around the handler to give the user or oscilloscope an idea of overheads.

`' USBinterrupt USB_EXTI_18vec# EXC: USB_ISR \ -- addr`
The entry point for the USB interrupt.

`: usbIntInit \ --`
Initialise USB hardware interrupt handler.

`: InitUSB \ --`
Initialise the USB hardware.

`: startUSB \ --`
Start the USB system. This word gives more control to devices that have VBUS control. If such devices are rebooted without power down, they may/will need to perform a `USBdisconnect` operation as part of the start up sequence. A disconnect, delay, connect sequence is part of `startUSB`.

6 USB base layer

The file *UsbBase.fth* contains an (almost) hardware independent USB code base. The hardware layer files, *UsbHwxxxx.fth* provide a standard interface that isolates hardware from the base layer. Class drivers such as *MscDrv.fth* should be compiled after the core layer and can reference the hardware layer. After the class files have been compiled, compile `UsbEP0.fth`, which contain the USB control and initialisation.

The five `VALUE`s below are used to provide status information for the core layer, but are actually contained in *UsbDefs.fth* to avoid forward references and to permit easy reference by the class drivers. Note that the endpoint status values are bit masks in a hardware independent form. See `EP>mask` below.

```
0 value UsbDevAddr      \ -- u
```
USB device address

```
0 value UsbConfig       \ -- x
```
USB Configuration set. Non-zero when the USB device has been configured.

```
0 value UsbEPmask       \ -- x
```
Endpoint mask.

```
0 value UsbEPhalt       \ -- x
```
Endpoint halt status.

```
0 value Usb#IfSet       \ -- n
```
Number of interfaces set.

```
: EP>mask       \ ep# -- mask
```
Decode an endpoint number to a bit mask such that if bit 7 is clear (OUT endpoints), the bit is in the low 16 bits, and if set (IN endpoints) in the upper 16 bits.

```
#USBifs buffer: USBAltSets      \ -- addr
```
Alternate settings array

```
struct /UsbEPdata       \ -- len
```
USB Endpoint transfer data structure.

```
UAlign  /UsbEPdata buffer: EP0Data      \ -- addr
```
Endpoint 0 transfer control structure.

```
UAlign  /USBSetupPacket buffer: SetupPacket     \ -- addr
```
Current setup packet.

```
UAlign  /MaxPacket0 buffer: EP0Buf      \ -- addr
```
Packet buffer for endpoint 0 data.

```
: ReqType       \ -- b
```
Return the first byte in the global `SetupPacket`.

```
: ReqDir        \ -- $80|$00
```
Return the direction bit of the `usp.bmRequestType` field in the global `SetupPacket`.

```
: ReqTypeBits   \ -- %0xx00000
```
Return the type bits of the `usp.bmRequestType` field in the global `SetupPacket`.

```
: ReqRecipient  \ -- %000xxxxx
```
Return the recipient bits of the `usp.bmRequestType` field in the global `SetupPacket`.

`: EP0Addr \ -- addr|0`
The address contained in `EP0Data ep.*Data` field.

`: /EP0Data \ -- len`
The size of the data in `EP0Data` as given by the `ep.Count` field.

`: NoEP0Data \ --`
Reset `EP0data`.

`: DevResetCore \ --`
Reset the core software configuration. This word is part of USB device reset chain.

`: UsbSetupStage \ --`
USB Request - Setup Stage

`: +EP0Data \ len --`
Update the `EP0Data` structure by *len* bytes.

`: UsbDataInStage \ --`
USB Request - Data In Stage. Data is taken from the `EP0Data` global structure.

`: doDataIn \ caddr len --`
Set up the data and do the DataIn stage.

`: ?MoreDataIn \ --`
Performed in the USB frame check and sends more data if required.

`: USBDataOutStage \ --`
USB Request - Data Out Stage. Data is taken from the `EP0Data` global structure.

`: sendZLPin \ ep# --`
Send a zero length packet from the given endpoint.

`: USBStatusInStage \ --`
USB Request - Status In Stage.

`: USBStatusOutStage \ --`
USB Request - Status Out Stage.

`: stall_i \ --`
Stall EP0 input.

`: stall_o \ --`
Stall EP0 output.

`: RTDst \ -- flag ; true=success`
Get REQUEST_TO_DEVICE status.

`: RTIst \ -- flag ; true=success`
Get REQUEST_TO_INTERFACE status.

`: RTEst \ -- flag ; true=success`
Get REQUEST_TO_ENDPOINT status.

`: USBGetStatus \ -- flag ; true=success`
Get Status USB Request. The data is in the global `SetupPacket`.

`: RTDsetf \ sc -- flag ; sc, true=set; flag, true=success`
Set/Clear Device feature.

`: RTEsetf \ sc -- flag ; sc, true=set; flag, true=success`

Set/Clear Endpoint feature.

`: USBSetClrFeature \ sc -- flag ; sc, true=set; flag, true=success`
Set/Clear Feature USB Request. The data is in the global `SetupPacket`.

`: maxEP0resp \ len -- len' ; SFP003`
Clip the given length to the wLength value in the setup packet.

`: GetCfgDesc \ -- flag ; true=success`
Get Configuration descriptor. The data is in the global `SetupPacket`.

`: GetStrDesc \ -- flag`
Get String descriptor. The data is in the global `SetupPacket`. Used when `USBStr@` has been defined and the string index is a small integer. `USBStr@ ( index -- caddr len )` is usually defined in the configuration file. See *CdcMscConfig.fth* for an example.

`: GetStrDesc \ -- flag`
Get String descriptor. The data is in the global `SetupPacket`. Used when `USBStr@` has not been defined and the string index is the byte offset into the string table.

`: GetRTDDesc \ -- flag ; true=success`
Get Device Descriptor USB Request. The data is in the global `SetupPacket`.

`: USBGetDescriptor \ -- flag ; true=success`
Get Descriptor USB Request. The data is in the global `SetupPacket`.

`: DisableAllEPs \ --`
Disable all endpoints except 0.

`: SetCfgConfig \ desc -- desc'`
Process the base configuration descriptor, returning the next descriptor.

`: initEP \ desc ep# mask -- desc`
Initialise an endpoint for use.

`: termEP \ ep# mask --`
Terminate using an endpoint. Only needed if `USBSetIface` is provided.

`: SetEPConfig \ desc -- desc`
Process an endpoint descriptor, returning the same descriptor.

`: SetConfig \ --`
Set the configuration from the descriptor. The `CfgDesc` Configuration descriptor (see *Descriptors.fth* is walked and processed.

`: ClrConfig \ --`
Clear the existing configuration.

`: USBSetConfig \ -- flag ; true=success`
Set Configuration USB Request.

`: USBSetIface \ -- flag ; true=success`
Set Interface USB Request. The data is in the global `SetupPacket`. This word is now unused and is commented out.

7 USB EndPoint 0

The file *UsbBase.fth* contains an (almost) hardware independent USB code base. The hardware layer files, *UsbHwxxxx.fth* provide a standard interface that isolates hardware from the base layer. Class drivers such as *MscDrv.fth* should be compiled after the core layer and can reference the hardware layer. After the class files have been compiled, compile `UsbEP0.fth`, which contains the USB control and initialisation code using endpoint 0.

7.1 Setup operations

`: RTD?` `\ -- flag`
Is this a request to the device? If so return true.

`: RTI?` `\ -- flag`
Is this a request to the device? If so return true.

`: doEP0SuStd` `\ --`
Handle the EP0 REQUEST_TYPE_STANDARD setup operations.

`: doEP0ClassRTI` `\ --`
Handle the class REQUEST_TO_INTERFACE operations.

`: doEP0SuClass` `\ --`
Handle the EP0 REQUEST_TYPE_CLASS setup operations.

7.2 Despatch

`: doEP0setup` `\ --`
Perform the EP0 SETUP operations.

`: doEP0out` `\ --`
Perform the EP0 OUT (to device) operations.

`: doEP0in` `\ --`
Perform the EP0 IN (to host) operations.

`: doEP0` `\ event --`
The despatcher for the actions of endpoint 0.

7.3 USB system initialisation

As of November 2010, start up of the USB code is performed by `startUSBsys ( -- )`, which can be found in *UsbEP0.fth*. `startUSBsys` uses conditional compilation to produce a start up word that initialises all the required classes. Please use `startUSBsys` as your USB start up word from now on.

`: StartCDC` `\ --`
Start the USB and CDC systems. Used for CDC only devices. NOW OBSOLETE - see `startUSBsys` below.

`: StartMSC` `\ --`
Initialise MSC and start USB. NOW OBSOLETE - see `startUSBsys` below.

`: StartCompo` `\ --`
Start the USB CDC and MSC systems. Used for composite devices. NOW OBSOLETE - see `startUSBsys` below.

`: startUSBsys  \ --`
Initialise all the selected USB components and then start the USB connection.

8 Mass Storage Class (MSC) driver

The Mass Storage Class (MSC) driver uses the USB Mass Storage Class Bulk Only Transport (BOT) specification. This, in turn, simulates a SCSI disc controller and so many SCSI terms are used here.

The driver relies on user-supplied sector read/write routines. In order to reduce the time spent in the USB interrupt handler, the sector read/write routines are accessed by a separate task. A consequence of this is that if the USB and the device's application share access to the drive, e.g. through the FAT file system, semaphores **must** be used for disk access.

Note that the file *%FATfiler%\SDspi.fth* of 19 November 2008 or later is required for use with SD/MMC cards that use the generic SPI driver.

The file system needs these words to access the mass storage:

`SecRead` (addr sector dev –) Reads a sector from the specified device. `THROW`s on error. Use *dev=0*.

`SecWrite` (addr sector dev –) Writes a sector to the specified device. `THROW`s on error. Use *dev=0*.

`MassInit` (–) Initializes mass storage access.

`MassTerm` (–) Terminates mass storage access.

`CDin?` (– flag) Return true if card/device ready.

`WPin?` (– flag) Return true if card/device write protected.

To simplify the code, the raw disc read/write interface treats all read/write errors as fatal, and `THROW`s on error. Retries should be accommodated within the sector read/write code.

The code is sensitive to the condition of the equate `usbDMA?`. We **strongly** recommend that this is set non-zero if your hardware supports DMA for USB transfers.

8.1 MSC state machine

The Mass Storage Class (MSC) bulk endpoint actions are controlled by a state machine.

```
: [sm           \ -- 0
```
Starts the definition of a state machine's states.

```
: smState       \ n -- n+1
```
Defines the next state as an `EQU` and increments the state number.

```
: sm]           \ n --
```
Finishes the state machine and defines an equate of the number of states.
The MSC bulk endpoint state machine.

```
[sm
  smState smCBW         \ wait for a CBW from host, idle state
  smState smDOUT        \ receive data blocks (sectors) from host
  smState smDIN         \ transmit data blocks (sectors) to host
  smState smCSW         \ send CSW to host
```

```
  smstate smCIN         \ send command response to host
  smState smStalled     \ both bulk endpoints stalled
sm] #MSCsm      \ -- n
```

`smCBW value MSCsm#      \ -- state`
MSC Bulk endpoint state.

`: .MSCstate     \ --`
Display the current state of the MSC state machine.

8.2 MSC data

`struct /DataPtr \ -- len`
describes the layout of length/addr data pointers.

`2variable mscBulkData   \ -- addr`
The data pointer for the remaining data of the bulk sector transfer. Note that this may be only part of the transfer, especially for multi-sector disk reads and writes.

`variable mscBulkLen     \ -- addr`
Holds the remaining length of the complete transfer. Before the transfer starts, the data corresponds to the Dx value in the "13 conditions" tests.

`variable mscHostLen     \ -- addr`
Holds the transfer size read from the CBW data block. This corresponds to the Hx value in the "13 conditions" tests.

`: initBulk      \ addr len --`
Set the data pointers for a bulk transfer.

`struct /CBW     \ -- len`
SCSI Command Block Wrapper (CBW). The CBW fields are little endian for USB, but the CDB fields are big-endian SCSI data.

`struct /CSW     \ -- len`
SCSI Command Status Wrapper (CSW). Little-endian data for USB.
According to the specs (see the MSC overview), there should only be one CBW outstanding until the corresponding CSW has been sent. Consequently, we do not need to queue commands, and only need one set of buffers.

`MscBoutEP BulkDD:`
Bulk OUT DMA descriptor.

`MscBinEP BulkDD:`
Bulk IN DMA descriptor.

`UAlign  /MaxPacket buffer: CBWbuff      \ -- addr`
Holds a packet expected to be the next CBW. Includes the CDB.

`UAlign  /MaxPacket buffer: MSCbuff      \ -- addr`
Holds miscellaneous data in or out that is not sector data.

`UAlign  /CSW buffer: CSWbuff    \ -- addr`
Holds the CSW being assembled.

`UAlign  #512 buffer: USBSecBuff \ -- addr`
Single sector buffer.

`CBWbuff cbw.CDB constant CDBbuff        \ -- addr`

The base address of the CDB in `CBWbuff`. The first byte contains the operation code.

```
0 value SenseCode       \ -- 00kkccqq
```
The REQUEST SENSE command returns data in the KEY, ASC and ASQ fields. These are merged into the low 24 bits of `SenseCode`.

```
0x030C00 equ SenseWriteError    \ Card fail
0x031100 equ SenseReadError     \ Card fail
0x052000 equ SenseInvalidOp     \ bad command
0x052400 equ SenseInvalidCDB    \ command format bad
0x023A00 equ SenseNotReady      \ not ready, no card
0x0B0C00 equ SenseWriteFail     \ USB fail timeout
0x0B1100 equ SenseReadFail      \ USB fail timeout
```

8.3 Tools

```
: StallMSCin    \ --
```
Stall the bulk IN endpoint.

```
: StallMSCout   \ --
```
Stall the bulk OUT endpoint.

```
: BOTStall      \ --
```
Stall the transaction. Which endpoint to stall is determined by looking at the transfer direction intended by the host.

```
: readMSCOut    \ caddr -- len
```
Read the bulk OUT endpoint into a buffer of at least `/MaxPacket` bytes and return the number read.

```
: writeMSCIn    \ caddr len --
```
Write the buffer to the bulk IN endpoint.

8.4 Disk Read/Write task

Disk read and write is controlled by a task. This is done to prevent the USB interrupt taking too long.

`SecRead` (addr sector dev –) Reads a sector from the specified device. `THROW`s on error.

`SecWrite` (addr sector dev –) Writes a sector to the specified device. `THROW`s on error.

```
-1 value USBdisk?       \ -- flag ; FVD009
```
Returns true if the USB drive should be available to the USB host.

```
0 value DriveBusy?      \ -- x
```
Returns the current drive command and direction.

```
$100 DirIn or equ SecReadCmd    \ -- x
```
Indicates that sectors are being read.

```
$100 DirOut or equ SecWriteCmd  \ --
```
Indicates that sectors are being written.

```
0 value DriveComplete   \ -- x
```
Set non-zero when the USB system has finished a command.

```
0 value FirstSector     \ -- u
```

First sector to be read/written.

```
0 value NumSectors      \ -- u
```
Number of sectors to be read/written.

```
0 value DriveStatus     \ -- x
```
Holds the drive action completion status. This is read by the USB interrupt handler to decide how to terminate the action and is cleared when termination has been handled. Negative values indicate errors, 0 is success

```
0 equ DrvOK     \ -- x
```
Drive status good.

```
-128 equ DrvWriteFail   \ --
```
Fatal write error.

```
-129 equ DrvReadFail
```
Fatal read error.

```
-130 equ DrvBadCmd
```
Invalid drive command.

```
-131 equ DrvAborted.
```
Command aborted by USB

```
-132 equ DrvUSBto
```
Command aborted by USB timeout.

```
: setDriveCmd   \ x --
```
Set the drive command and clear the status.

```
: Sector<>USB   \ -- ior ; 0=success, nz=timeout
```
Send or receive a 512 byte sector data to/from `USBSecBuff`.

```
: -BulkData     \ --
```
Acknowledge a sector transfer by zeroing the transfer address and length.

```
: Disk>USB      \ --
```
Read the given sectors from the disk.

```
: USB>Disk      \ --
```
Write the given sectors to the disk.

```
: DriveIn?      \ -- flag
```
Return true if the drive is in.

```
: ?MSCdrive     \ --
```
Wait for drive insertion after removal.

```
: WaitSecCmd    \ --
```
Wait for a sector command

```
: WaitDrvComplete       \ --
```
Wait for the drive action on the USB side to complete.

```
Task USBdiskTask        \ -- addr
```
The task control block for the MSC sector transfer task.

```
0 value USBdiskTask?    \ -- x
```
Returns non-zero when the task is running.

```
: +USBdiskTask  \ --
```

Start the USB disk task if not already running.

```
: -USBdiskTask  \ --
```

Stop the USB disk task if not already stopped.

8.5 CSW operations

```
: NoResidue     \ --
```

Zero the `CSWbuff csw.Residue` field.

```
: initCSW       \ --
```

Set up the CSW from the CBW. The residue field is set to the CBW data length field. For actions that complete immediately, this should be cleared by `NoResidue` above.

```
: sendCSW       \ --
```

Send the CSW as is.

```
: badRWphase    \ --
```

Perform the bad R/W setup actions, returning a phase error status (0x02).

```
: badRWfail     \ --
```

Perform the bad R/W setup actions, returning a fail status (0x01).

8.6 The 13 cases

The USB specifications define the host and device transactions in terms of 13 cases. The file *usbmassbulk_10.pdf* contains the gory details and can be downloaded from `www.usb.org`. You will probably use that document in conjunction with *INF-8070.pdf* to be found at `ftp://ftp.seagate.com/sff/INF-8070.PDF`.

```
: mscHostDir@   \ -- $00/$80
```

Return the Host direction, where $80=IN to host.

```
: mscHostLen@   \ -- len
```

Return the host length from the CBW.

```
: NoMscDevData? \ -- ior ; 0=success
```

Checks cases 1, 4, and 9. On an error (Hx>0), a pipe is stalled and a failure CSW sent. On success, no action is taken.

```
: MscDevDataIN-1?       \ caddr len -- caddr len' 0 | ior ; 0=success
```

Checks cases 2, 5, 6, 7, and 10. On error, appropriate action is performed. On success, no action is taken. This version is used for short responses less than `/MaxPacket`, when data is transmitted for case 5.

```
: MscDevDataIN-2?       \ len -- ior ; 0=success
```

Checks cases 2, 5, 6, 7, and 10. On error, appropriate action is performed. On success, no action is taken. This version stalls and sends no data for case 5.

```
: MscDevDataOUT?        \ len -- ior ; 0=success
```

Checks cases 3, 8, 11, 12, and 13. On error, appropriate action is performed. On success, no action is taken.

8.7 CBW operations

`2variable mscRespCIN  \ -- addr`
Contains a len/addr pair for command responses,

`: -RespCIN  \ --`
Mark no command response data, and switch to a CSW send.

`: +RespCIN  \ caddr len --`
Trigger a command response phase.

`: doTUR  \ --`
Perform the TEST UNIT READY actions.

`#18 equ /SenseData  \ -- len`
Size of the response to the REQUEST SENSE command.

`create SenseTemplate  \ -- addr`
Holds the 18 byte response to the REQUEST SENSE command. The $FF bytes are filled in with data from `SenseCode`.

`: doRS  \ --`
Respond to the REQUEST SENSE command.

`#36 equ /InqData  \ -- len`
Size of returned INQUIRY data.

`create InqData  \ -- addr`
The returned INQUIRY data.

`: doINQ  \ --`
Return the INQUIRY data.

`: doRC10  \ --`
Performs the READ CAPACITY (10) command operations.

`: initMSCrw  \ --`
Initialise the read/write data from the CDBbuff

`: +DiskRD  \ --`
Start the disk task read.

`: +DiskWR  \ --`
Start the disk task write.

`: doRD10  \ --`
Performs the READ (10) command operations.

`: doWR10  \ --`
Performs the WRITE (10) command operations.

`: doVRFY10  \ --`
Performs the VERIFY (10) command operations. For the moment this is a dummy that always succeeds unless no drive is present.

`create MS6resp  \ -- addr`
The MODE SENSE (6) response.

`: doMS6  \ --`
Performs the MODE SENSE (6) command operations. Taken from other code - I have no idea! This command is not needed by Windows, but is required by Linux.

`: doRFC \ --`
Performs the READ FORMAT CAPACITY command operations. This command is supposed to be obsolete or vendor specific, but Windows may want it. See *INF-8070.PDF* for the details. NO LONGER REQUIRED.

8.8 Endpoint despatcher

This section contains the handlers for the MSC bulk IN and OUT handlers, plus the actions required for set up operations performed on endpoint 0.

`: +DiskData \ u --`
Add u bytes to the bulk transfer control variables.

`: ?AckSector \ --`
If a full sector has been transferred, acknowledge it.

8.8.1 Without DMA

`: ReadCBW \ --`
Read a CBW from the bulk OUT endpoint and process it.

`: TransmitCSW \ --`
Transmit the CSW on the bulk IN endpoint, and switch the state machine to wait for the next CBW.

`: readMSCdata \ --`
Read the bulk OUT endpoint for a transfer operation

`: writeMSCdata \ --`
Write the bulk IN endpoint for a transfer operation. Note that this may receive NAK interrupts as it is a Bulk IN endpoint.

`: DiscardMSCout \ --`
Read and discard data received on the MSC Bulk OUT endpoint.

`: MSCBulkOut \ --`
Handle bulk OUT actions to the device. Note that this includes NAK interrupts.

`: MSCBulkIn \ --`
Handle bulk IN actions to the host. Note that this includes NAK interrupts.

`: doEP2 \ event --`
The action of endpoint 2.

`: MscFrameCheck \ --`
Executed in the frame interrupt. No longer needed as it is done by default for non-DMA drivers.

`: DevMSCreset \ --`
Reset the MSC driver. Performed at start up and as an EP0 action.

8.8.2 With DMA

`: startReadMSCdata \ --`
Prepare the bulk OUT endpoint for a transfer operation

`: endReadMSCdata \ --`
Handle the bulk OUT endpoint data just received.

`: startWriteMSCdata \ --`
Write the bulk IN endpoint for a transfer operation.

`: endWriteMSCdata \ --`
Update after a bulk IN endpoint data transfer operation.

`0 value txCSWdone? \ -- flag`
An interlock used to prevent multiple transmissions of the CSW.

`: startTransmitCSW \ --`
Transmit the CSW on the bulk IN endpoint.

`: endTransmitCSW \ --`
CSW transmit done, switch the state machine to wait for the next CBW.

`: startRespCIN \ --`
Start the command response transfer.

`: endRespCIN \ --`
Mark the end of the command response stage.

`: startReadCBW \ --`
prepare for a CBW read.

`: endReadCBW \ --`
Read a CBW from the bulk OUT endpoint and process it.

`: MSCBulkOutNdr \ --`
Handle bulk OUT requests (data to the device).

`: MSCBulkOutEot \ --`
Handle bulk OUT completions (data to the device).

`: MSCBulkInNdr \ --`
Handle requests for bulk IN actions (data to the host).

`: MSCBulkInEOT \ --`
Handle completion of bulk IN actions (data to the host).

`: doEP2 \ event --`
The action of endpoint 2.

`: MscFrameCheck \ --`
Executed in the frame interrupt. Enables both Bulk endpoints, and seems to be required. We do not yet know why!

`: resetMSCdata \ --`
Reset the MSC driver data. Performed at start up and as an EP0 action.

`: DevMSCreset \ --`
Reset the MSC driver. Performed as an EP0 action.

8.9 EP0 SETUP actions

`: MscReset \ -- flag ; 0=failed`
Reset the MSC driver. EP0 action.

`: MSCGetMaxLUN \ -- flag ; 0=failed`
This device does not distinguish between LUNs. EP0 action.

`: MSCdoEP0ClassRTI \ --`
Handle the MSC class REQUEST_TO_INTERFACE operations.

8.10 Diagnostics and Test code

`: initMSCdata  \ --`
Initialise the MSC data and task.

9 Communications Device Class (CDC) driver

The Communications Device Class (CDC) driver uses the ACM subclass. Only a limited set of control functions are implemented beyond providing a stub. If you need them, expand them to suit your requirements.

There are two USB interfaces, a control interface with an interrupt in endpoint, and a data interface with a bulk in and and bulk out endpoint per COM channel.

The COM channel(s) are implemented as the ends of `CQUEUE` structures. These queues should be large enough to hold the number of bytes accumulated during a USB poll interval. The poll interval is defined in the endpoint descriptor for the control interface. By default, this is set to 1, indicating a poll interval of 1 ms.

9.1 Data and buffers

`-1 value RunCDC?  \ -- x`
Set non-zero when CDC operation is enabled.

`#10 aligned buffer: NotifyBuf  \ -- addr`
Holds CDC notification data.

`7 aligned buffer: LineCodingBuf \ -- addr`
Holds current line coding information.

`2 buffer: SerialState  \ -- addr`
Holds serial state.

`2 buffer: LineControlState  \ -- addr`
Holds Line Control State info.

`create InitNotifyBuf  \ -- addr`
Initialisation data for the notify buffer

`create InitLineCoding`
Initialisation data for the line coding, usbcdc11.pdf 6.2.13 p57.

9.2 CDC Class RTI requests

`: CDCdoEP0ClassRTI  \ --`
Handle the CDC class REQUEST_TO_INTERFACE operations.

9.3 CDC Bulk endpoint handling

`variable #LastIn0  \ -- addr`
Holds the number of bytes in the last packet sent to the host. This is used to prevent a Windows problem that occurs when the last IN packet contains `/MaxPacket` bytes. In this case a following zero-length packet is required.

9.3.1 Non DMA operation

Operation without DMA drivers is portable across a wide range of CPUs and USB engines.

`/MaxPacket buffer: CdcBulkBuf  \ -- addr`
Packet buffer for CDC bulk transfers.

`0 value CDCdeferred?  \ -- flag`
Set true if the serial input buffer is full. The current packet is then retried in the frame interrupt.

`: CdcBulkOut  \ --`
Handle bulk OUT actions to the device. Note that this includes NAK interrupts.

`: CdcBulkIn  \ --`
Handle bulk IN actions to the host. Note that this may include NAK interrupts.

`: doCDCbulkInOut  \ event --`
The action of endpoints which combine CDC BULK IN and BULK OUT.

`: doCDCbulkIn  \ event --`
The action of an endpoint for CDC BULK IN only.

`: doCDCbulkOut  \ event --`
The action of an endpoint for CDC BULK OUT only.

`: CdcFrameCheck \ --`
Executed in the frame interrupt.

9.3.2 DMA operation

Because of the variety of DMA engines used for DMA, isolation of a hardware dependent DMA layer is more complex. As yet, we do not guarantee that the hardware dependent DMA layer is portable to all CPUs and USB engines.

`CDCBoutEP BulkDD:`
Bulk OUT DMA descriptor.

`CDCBinEP BulkDD:`
Bulk IN DMA descriptor.

`/MaxPacket buffer: CdcDmaOutBuff  \ -- addr`
DMA buffer for data OUT from host.

`/MaxPacket buffer: CdcDmaInBuff \ -- addr`
DMA buffer for data IN to host.

`CDCBoutEP EPaddr equ CDCBoutPhys  \ -- u`
Physical address of the CDC Bulk OUT channel.

`CDCBinEP EPaddr equ CDCBinPhys  \ -- u`
Physical address of the CDC Bulk IN channel.

`: initCDCbo  \ --`
Start the next CDC Bulk out DMA transfer.

`: initCDCbi  \ --`
Initialise the CDC Bulk IN DMA transfer.

`0 value CDCdeferred?  \ -- flag`
Set true if the serial input buffer is full. The current packet is then retried in the frame interrupt.

`: CdcBulkDmaOut \ --`
Handle CDC bulk DMA OUT actions to the device.

`: CdcBulkDmaIn  \ --s`
Handle CDC bulk IN actions to the host.

`: doCDCbulkInOut  \ event --`

The action of endpoints which combine CDC BULK IN and BULK OUT.

`: doCDCbulkIn  \ event --`
The action of an endpoint for CDC BULK IN only.

`: doCDCbulkOut  \ event --`
The action of an endpoint for CDC BULK IN only.

`: CdcFrameCheck \ --`
Executed in the frame interrupt. If there is data to be sent to the host, Bulk NAK IN interrupts and channel DMA are enabled.

9.4 CDC Interrupt endpoint

`: doCDCintIN  \ event --`
The action of the CDC interrupt endpoint IN.

9.5 Reset and initialisation

`: resetCDCdata  \ --`
Reset the CDC driver data, initialising CDC specific data to start/restart operation. Performed at start up and as an EP0 action.

`: DevCDCreset  \ --`
Reset the CDC driver, initialising the CDC specific data to start/restart operation. Performed as an EP0 action.

`: CDCreset  \ -- flag ; 0=failed`
Reset the CDC driver. EP0 action.

9.6 Diagnostics and Test code

`: UsbCon  \ --`
Switch the Forth console to the USB channel.

`: DefCon  \ --`
Switch to the default Forth console, normally on UART0.

`: UsbDisCon  \ --`
Switch to the default console (usually a serial device) and disconnect the USB from te host.

9.7 Testing with operating systems

9.7.1 Windows

USB serial devices can be determined using:

Control Panel -> System

Hardware -> Device Manager -> Ports

Once a USB serial device has been found, it will usually use the same port number when reconnected. To use two or more of the same device, they must have different serial numbers.

For testing we use PuTTY or HyperTerm and AIDE's PowerTerm. HyperTerm is a very old program written before USB serial ports were available. It does not have sufficient error recovery

for USB serial ports, and so is not suitable for production use. Despite this, it is available on all Windows PCs and your clients will use it. PuTTY and TeraTerm are widely recommended by USB developers.

Under Windows, a .INF file is required for any USB device that includes a CDC class driver. The PID and VID (see below) must match those in your USB device descriptors, as must the manufacturer string. Sample .INF files are provided, which you can use as the basis of your own. Windows 10, 8, 7, Vista, and XP SP3 work well, but XP SP2 is not so robust. For XP, either upgrade to SP3 or install hotfixes. It seems that you need:

- Hotfix KB935892 (which brings usbccgp.sys 5.1.2600.3116)
- Hotfix KB918365 (which brings usbser.sys 5.1.2600.2930)

We installed SP3 for testing and had to copy the new version of *usbser.sys* manually from *SP3.cab*. Use Windows search and copy the new version to replace the one in *Windows\System32\Drivers*.

The Windows INF file for many CDC implementations is *mpecdcW7.inf*. When installing this on Windows 7 onwards, Windows will probably say that it cannot find a driver automatically. To overcome this, select manual installation. Make sure the INF file is the only INF file in the selected driver directory. Windows will then complain that the driver is unsigned. Tell Windows to carry on regardless and the INF file will install.

9.7.2 Linux

USB serial devices

When using USB serial devices, the name used varies according to your distribution. The most common names appear to be:

```
/dev/ttyUSBx
/dev/ttyACMx
```

There are several methods of finding USB serial ports. The simplest seems to be to unplug the device, then reconnect it, then type the following incantation:

```
dmesg | grep tty
```

where you must have root access. On many systems, e.g. Ubuntu

```
sudo dmesg | grep tty
```

is required. The last few lines should then tell you which USB serial port, e.g. `/dev/ttyUSB0` was selected for your device. If the last tells you that the device is now disconnected, it is probably because of the "brltty bug". Unless you need the Braille TTY access, remove the package *brltty*. Repeat:

```
sudo dmesg | grep tty
```

to check that device remains connected. Some forums suggest that you may also need to create the */dev/ttyUSBx* entries. Do this with:

```
sudo mknod /dev/ttyUSB0 c 188 0
sudo mknod /dev/ttyUSB1 c 188 1
sudo mknod /dev/ttyUSB2 c 188 2
```

Linux serial terminal emulators

The most widely used Linux equivalent to Windows' HyperTerm appears to be *minicom*. It isn't pretty, but it works and is easy to use. There are plenty of others, including GUI ones, but *minicom* is the one we come back to as it is available for nearly all distributions.

9.7.3 Mac OS X

For automatic recognition by Mac OS X, bit 0 of the protocol byte in the relevant descriptors **must** be set. After enumeration, your USB serial sevices will appear in the forms:

```
/dev/tty.usbmodem000031FD1
/dev/cu.usbmodem000031FD1
```

Tty is for "incoming calls" and cu for "outgoing calls". It has to do with behaviour when opening the port, though you can achieve any behaviour you want on either device by opening it non-blocking and reconfiguring the tty settings.

To test a serial device, run a terminal

```
Applications -> Utilities -> Terminal
```

Within the terminal run the **screen** program.

```
screen /dev/tty.usbmodem000031FD1
```

From a terminal you can get the documentation:

```
man screen
```

Index

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W