Reference Manual for the *usbModel* Software and Co-simulation Host and Device model

(version 1.0.0)

Simon Southwell

January 2024
(last updated 20th August 2024)

Copyright

Copyright © 2023-2024 Simon Southwell (Wyvern Semiconductors)

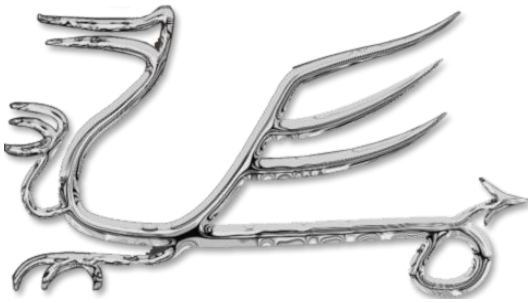
This document may not, in whole or part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior written consent from the copyright holder.

Disclaimers

No warranties: the information provided in this document is “as is” without any express or implied warranty of any kind including warranties of accuracy, completeness, merchantability, non-infringement of intellectual property, or fitness for any particular purpose. In no event will the author be liable for any damages whatsoever (whether direct, indirect, special, incidental, or consequential, including, without limitation, damages for loss of profits, business interruption, or loss of information) arising out of the use of or inability to use the information provided in this document, even if the author has been advised of the possibility of such damages.

Simon Southwell (simon@anita-simulators.org.uk)

Cambridge, UK, January 2024

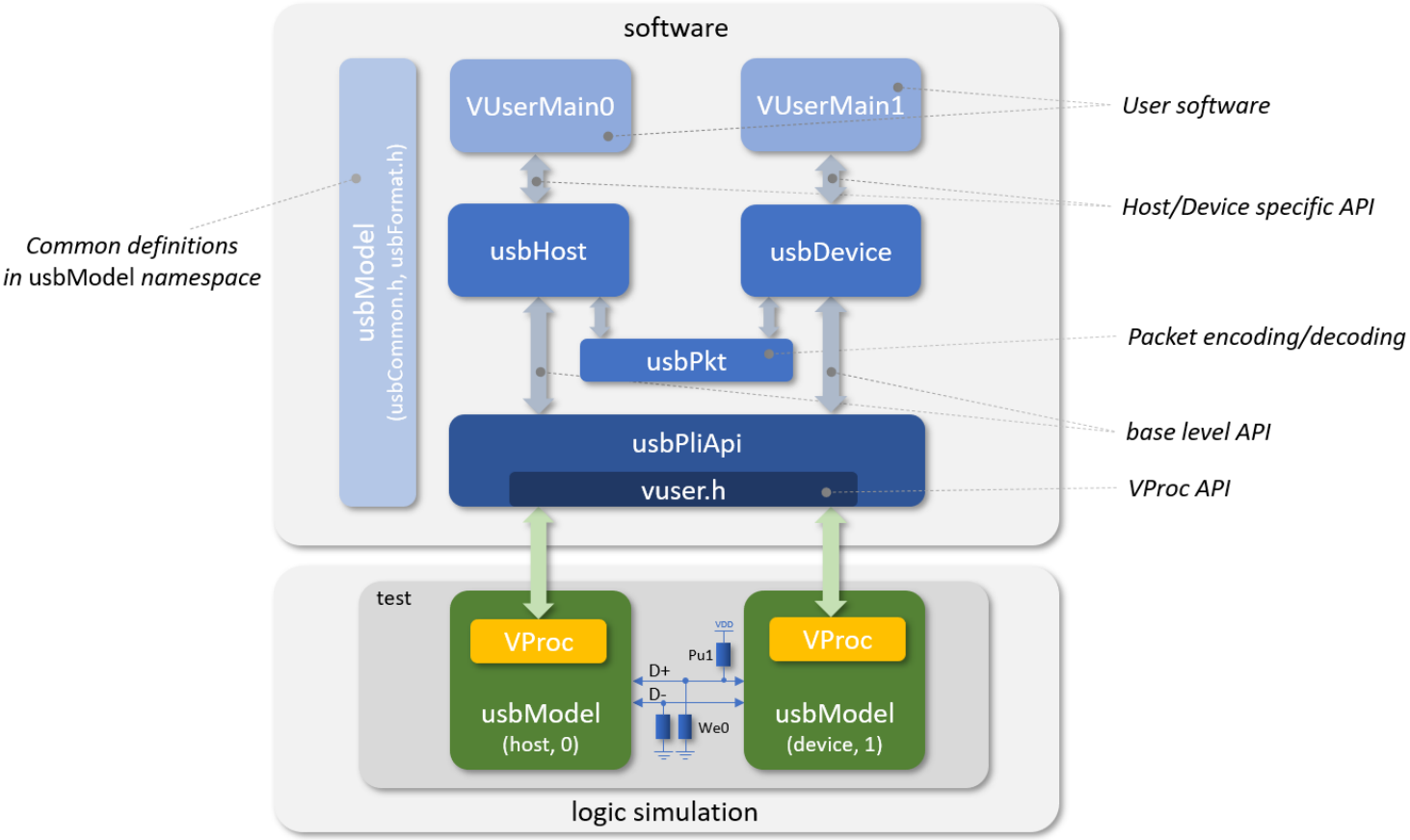


Contents

Introduction

The `usbModel` is a software model of both a USB host and device to standard USB 1.1, with hooks for USB 2.0 enhancements. Each model can be used independently, with the host model a generic implementation, and the device model an example of a specific communications device class (CDC) implementation. The models have been integrated with the [VProc Virtual Processor] to drive USB signals in a Verilog simulation, with scripts for running on ModelSim or Questa. This allows the model to be used to drive device, hub, and host implementations.

A test environment is provided to demonstrate the models in action, with examples of enumeration, data transfer, device reset and suspension. The diagram below shows the main components of this system:



The Models' Features

The Host Features

The following list the features of the `usbModel` host:

- Automatic checks for connection/disconnection.
- Automatic generation of SOF token packets each frame.
- Ability to generate control transactions.
    - Get device, interface, endpoint, string, and class specific descriptors.

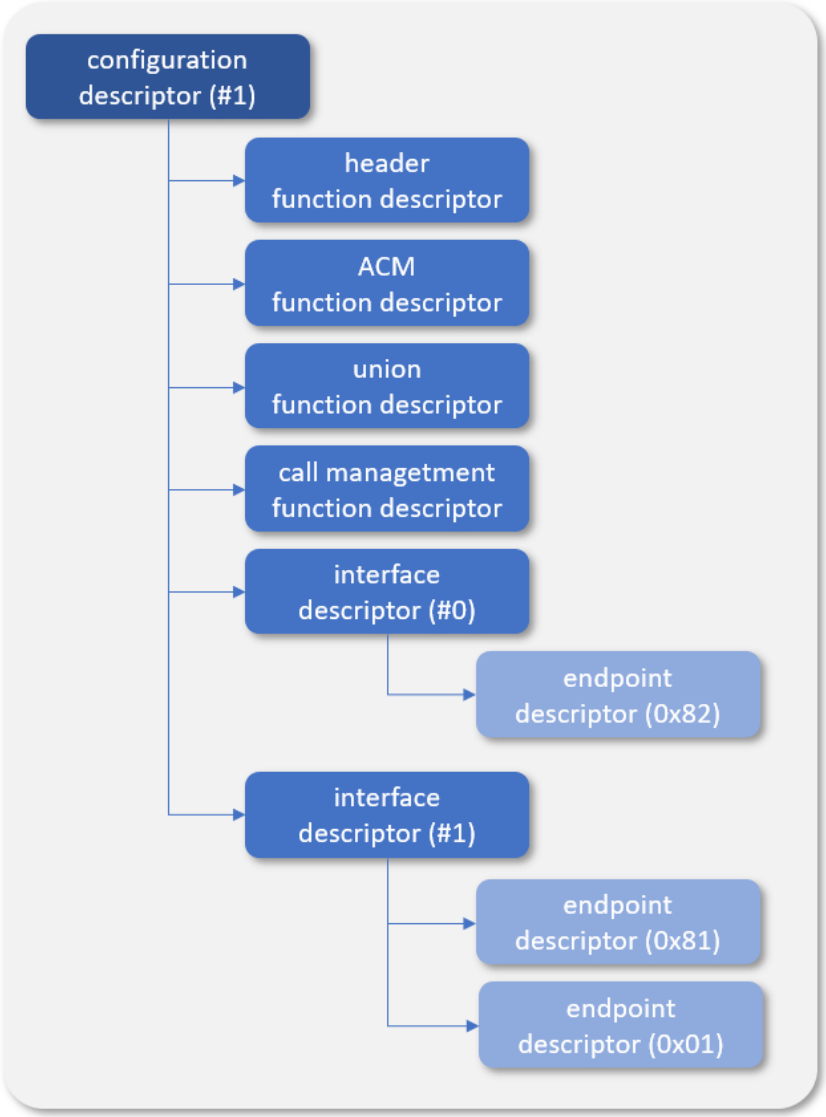
        - Set device address.
        - Get device, interface, and endpoint statuses.
        - Get and set device configuration (enable/disable).
        - Get and set interface alternative.
        - Set and clear device, interface, and endpoint features.
        - Get endpoint synch frame state.
    - Generate bulk and isochronous OUT packets.
    - Generate bulk and isochronous IN packets.
    - Suspend a device.
    - Reset a device.
    - Display formatted output of received packet data.

The Device Features

The device model is that of a communications device class (CDC), set up as a basic data transfer unit. The following list the features of the `usbDevice` model:

- Ability to connect and disconnect from USB line.
- Suspension detection.
- Reset detection.
- Implements a CDC device with:
    - One device configuration.
    - Two interfaces.
    - Three endpoints.
        - One a notify endpoint on one interface.
        - Two data endpoints (IN and OUT) on the other interface.
- Responses to control packets.
- Bulk data transfers IN and OUT.
    - A user callback can be registered at construction, called for each transfer request received.
- Display formatted output of received packet data.

The device implements a configuration descriptor structure as show in the diagram below:



Pre-requisites

The demonstration/test program can be run on both Linux and Windows (requiring MSYS2/mingw-w64 installation), both requiring the `gcc` toolchain (C and C++) to be present.

The Simulations use the VProc Virtual Processor component which is assumed to be checked out in the same folder/directory in which the usbModel repository was checked out. However, if this is not present, then the scripts will automatically check it out in the expected location, so long as the folder has user write permissions.

The simulation scripts, by default, assume that ModelSim or Questa has been installed and the environment set up for one of these simulators. For other simulations, the scripts and `veriuser.cpp` will need to be adapted.

Software Architecture

The usbModel software architecture consists of a set of common layers and then the host/device model specific code. The diagram in the introduction above shows the main layers.

Common to all the code is a set of constant and type definitions for use by all the code and gathered into a namespace called `usbModel`. These are defined in `usbCommon.h`. Also in this name space is defined a set of formatting utilities, defined in `usbFormat.h` and `usbFormat.cpp`.

The lowest level functional code is in the `usbPliApi` class (in `model/src/usbPliApi`). This is a base class meant to be inherited by a higher layer class which gives access to lower-level function to access and drive the USB data lines. Other than the constructor, all the methods are protected and not available to user code. This layer includes functionality such as setting idle or reset on the line for a specified time, enabling or disabling pullups/pulldowns, fetching the current clock tick count, inspecting the current line state, sending a NRZI encoded packet and waiting for a packet. The class is specific to using the VProc API (defined in `VUser.h` in that repository, with C linkage). It is this layer that abstracts away the specific simulation environment from the rest of the code, so, to integrate the model with another environment, it is the API from this class that needs to be reproduced for a different setup (more later).

The next layer comes from the `usbPkt` class (defined in `usbPkt.h` and `usbPkt.cpp` in `model/src`). Like `usbPliApi`, this is a base class meant to be inherited by a higher layer class, and all its methods, except the constructor, are protected. It provides the lowest level encoding, generating basic USB packets—tokens, handshakes, data and SOFs— and NRZI encoding them into a provide buffer, ready for sending over the USB data lines. It also decodes received NRZI packet data into the basic packet types, extracting the information and/or data into a provided buffer. As such, it is a stand-alone class with no dependencies other than the common definitions and formatting utilities of the usbModel `namespace`. Parameters and data are passed in for encoding into a provide buffer, and NRZI data is passed in for decoding into provide argument references and data buffers.

At the highest level of the models are the two USB classes for the host and device. These are dependent on both the `usbPliApi` class and the `usbPkt` class, and these are inherited by them. The two classes provide the user API to the models with various relevant features, discussed below. The host model is defined with the `usbHost` class (defined in `usbHost.h` and `usbHost.cpp`), whilst the device model is defined by the `usbDevice` class (defined in `usbDevice.h` and `usbDevice.cpp`).

The user code, as the environment is based on VProc, has entry point at `VUserMain`*x*, where *x* is the node number used when instantiating the `usbModel` Verilog module. To use the API for the given model, the relevant header just needs to be included.

It is important that the user code in the relevant `VUserMain`*x* uses the correct API and model that's attached to the specific Verilog usbModel module instantiated, as the code is configured for either host or device operation, and each instantiation has a unique node number that must be used by the relevant API when the API object is constructed. I.e., a `VUserMain0` program, for example, (and any code called from

there) uses a `usbHost` or `usbDevice` API object constructed with a node number of 0 and that the `usbModel` Verilog module instantiated with a node number of 0, is configured to be the correct type (host or device) for the API constructed in the user code.

Model Application Programming Interfaces

Both the host and device model provide a user API for performing various actions over the USB interface. Since USB has transactions instigated from a host (possibly via a hub) to which a device responds, the host API has methods for instigating transactions of various kinds, whilst the device is simply run, but with means to call user code on reception of data or a request for data. These APIs will be detailed in this section.

Host API

The constructor for the usbHost class is as shown in the table below:

```
usbHost (
  int           node,
  std::string   name    = std::string(FMT_HOST "HOST" FMT_NORMAL)
);
```

The `node` parameter specifies the Verilog not that the API will be attached to. This would normally be the value matching the `VUserMain`*x* user code entry function (see above). The optional `name` parameter specifies the tag string to use at the start of each formatted output on the display. The default uses a colourised “HOST” tag, with `FMT_HOST` and `FMT_NORMAL` defined in `usbFormat.h`. A whole set of colour definitions have been placed there if a user wishes to customise the formatted output.

Utility Methods

The API has some utility methods related to time and control of the simulation, as defined below:

```
void usbHostSleepUs (
  const unsigned    time_us
);
```

```
float usbHostGetTimeUs (
  void
);
```

```
void usbHostEndExecution (
  void
);
```

```
int usbHostWaitForConnection (
  const unsigned    polldelay  = 10 * usbPliApi::ONE_US,
  const unsigned    timeout    =  3 * usbPlaAPi::ONE_MS
);
```

```
int usbHostFindDescriptor (
    const int          desctype,
    const int          descidx,
    const uint8*       rawdata,
    const int          len,
          uint8*       descdata
);
```

The first of these puts the user code to sleep for the specified amount of time in microseconds, with the line driven to idle by the host. The second returns the current time in microseconds. This is derived from a clock count since time zero and the model's running frequency.

The third method allows a user program to terminate the execution of the simulation at any time, without delay. The final method in this group is used at the beginning of a user program to ensure the simulation is running, out of reset, and that a device has been connected. The method returns the state of the line when a connection is detected, with the D+ signal state in bit 0, and the D- signal state in bit 1.This can be used to decide what speed of device has been connected to the line.

The last method is used to extract descriptors from a buffer containing the entire returned descriptors from a configuration descriptor fetch. The type of descriptor to be extracted is defined in the `desctype` parameter, and can be one of the following:

- `usbModel::USB_DEV_DESCRIPTOR_TYPE`
- `usbModel::USB_CFG_DESCRIPTOR_TYPE`
- `usbModel::USB_STR_DESCRIPTOR_TYPE`
- `usbModel::USB_IF_DESCRIPTOR_TYPE`
- `usbModel::USB_EP_DESCRIPTOR_TYPE`
- `usbModel::USB_FUNC_DESCRIPTOR_TYPE`

The `descindx` parameter selects which of the particular type to extract, when multiple possible descriptors are possible for that type. If only one type possible, then this should be set to `usbModel::NOTVALID`. The length of the data to extract is defined in the `len` parameter, and the method won't return more than this. The data is returned in the buffer pointed to by `descdata`.

The method returns `usbModel::USBOK` if successful, but if the extracted data would overflow the `descdata` buffer then it returns `usbModel::USBERROR`.

Device Control Methods

The device control methods are used to fetch device information, such descriptors, and to configure the device. The device control API methods all have a similar structure:

```
int usbHostXXX (
  const uint8_t      addr,
  const uint8_t      endp,
                        method specific paramters
  const uint8_t      idle     = DEFAULTIDLEDELAY
);
```

Each of the control methods has an address parameter to specify which addressed device is being accessed. Before a device is configured to set an address (part of enumeration) this would be `usbModel::CONTROL_ADDR` to send control packets to the control pipe. An endpoint index is defined by the `endp` parameter, which would usually be set to `usbModel::CONTROL_EP` when doing control accesses. The `endp` parameter include the direction bit in bit 7, and the endpoint index in bits 3 down to 0. So valid endpoint values are `0x00` to `0x0f` and `0x80` to `0x8f`.

The control methods then have one or more methods specific parameters, followed by an optional `idle` parameter. This is used to add a delay before instigating each transaction sent from the host in order to emulate some processing time. The default for this is 4 clock cycles (as defined by `DEFAULTIDLEDELAY`).

The methods all return `usbModel::USBOK` on success. If an error occurred during the transaction, then `usbModel::USBERROR` is returned, or if a device disconnection is detected, then `usbModel::USBDISCONNECTED` is returned. If a valid, but unsupported, response packet is received from the device then the methods return a status of `usbModel::USBUNSUPPORTED`. If a timeout occurred waiting for a response packet, then `usbModel::USBNORESPONSE` is returned.

The rest of the section details the control methods, detailing their specific parameters.

A couple of methods are provided to fetch descriptors:

```
int usbHostGetDeviceDescriptor (
  const uint8_t      addr,
  const uint8_t      endp,
        uint8_t*     data,
  const uint16_t     reqlen,
        uint16_t     &rxlen,
  const bool         chklen   = true,
  const uint8_t      idle     = DEFAULTIDLEDELAY
);
```

```
int usbHostGetConfigDescriptor (
  const uint8_t      addr,
  const uint8_t      endp,
        uint8_t*     data,
  const uint16_t     reqlen,
        uint16_t     &rxlen,
  const bool         chklen   = true,
  const uint8_t      idle     = DEFAULTIDLEDELAY
);
```

This method is used to fetch a device or configuration descriptors from the connected device. The methods have a data buffer (`data`) pointer argument, in which the returned data is placed, and requested data length (`reqlen`). The actual length of data returned is placed in the `rxlen` reference. If the size of the descriptor is less than the requested length, then only the length of the descriptor is returned. If it is greater than the requested length, then it is truncated to the request length. The requested length can be checked against the returned length for a match if the optional `chklen` argument is set to `true`, with a default setting of `false`) where an error is returned if not equal. For the `usbHostGetConfigDescriptor` method, the device will try and return all the descriptor information, including interface, endpoint, and any class specific descriptors. This is where requesting only a truncated set of data is useful, requesting on the amount for the configuration descriptor, and then parsing this to ascertain the length of the entire set of descriptors, which can then be requested in their entirety.

The method for fetching a string descriptor is vary similar to the above, but with a couple of extra parameters.

```
int usbHostGetStrDescriptor (
  const uint8_t      addr,
  const uint8_t      endp,
  const uint8_t      stridx,
        uint8_t*     data,
  const uint16_t     reqlen,
        uint16_t     &rxlen,
  const bool         chklen   = true,
  const uint16_t     langid   = usbModel::LANG_ENG_UK,
  const uint8_t      idle     = DEFAULTIDLEDELAY
);
```

The matching parameters have the same meaning as for the previously mentioned `usbHostGetDeviceDescriptor`, but now a string index (`stridx`) is used to select which string descriptor to fetch. The string descriptor at index 0 has information about how many other string descriptors are present, and at what indexes. In addition an optional `langid` parameter is given to specify which language (of those supported by the device) to fetch for the index string. This defaults to `usbModel::LANG_ENG_UK`.

```
int usbHostSetDeviceAddress (
  const uint8_t      addr,
  const uint8_t      endp,
        uint16_t     devaddr,
  const uint8_t      idle       = DEFAULTIDLEDELAY
);
```

The above method to used set the address of the connected device, as specified in the `devaddr` argument. Although a 16-bit parameter (matching the type of the parameter in the setup packet) valid device addresses are limited to 7 bits, so range from 0 to 127. The method truncates the parameter to 7 bits if outside this range.

```
int usbHostGetDeviceStatus (
  const uint8_t      addr,
  const uint8_t      endp,
        uint16_t     &status,
  const uint8_t      Idle       = DEFAULTIDLEDELAY
);
```

The above method is used to fetch the status of the device and return it in the referenced `status` parameter. This usually contains bits to indicate whether the device is self-powered or not (bit 0) and whether it supports remote-wakeup or not (bit 1).

```
int usbHostGetDeviceConfig (
  const uint8_t      addr,
  const uint8_t      endp,
  const uint8_t      index      = 1,
  const uint8_t      idle       = DEFAULTIDLEDELAY
);
```

```
int usbHostSetDeviceConfig (
  const uint8_t      addr,
  const uint8_t      endp,
  const uint8_t      index      = 1,
  const uint8_t      idle       = DEFAULTIDLEDELAY
);
```

The above two methods are used to get a devices configuration state (enabled/ disabled), or to set it (enabled). The optional index parameter specifies which configuration to access if there are multiple configurations (though this is unusual).

```
int usbHostClearDeviceFeature (
  const uint8_t      addr,
  const uint8_t      endp,
  const uint16_t     feature,
  const uint8_t      idle       = DEFAULTIDLEDELAY
);
```

```
int usbHostSetDeviceFeature (
  const uint8_t     addr,
  const uint8_t     endp,
  const uint16_t    feature,
  const uint8_t     idle       = DEFAULTIDLEDELAY
);
```

The above two methods are used to clear and set features on the device. The `feature` argument specifies which feature to operate on. A device might have such features such as remote wakeup which can be enabled or disabled, or a test mode.

Interface Control methods

There are a smaller number of methods for interface control compared to the device, and these are detailed below.

```
int usbHostGetInterfaceStatus (
  const uint8_t     addr,
  const uint8_t     endp,
  const uint16_t    ifidx,
        uint16_t    &status,
  const uint8_t     idle       = DEFAULTIDLEDELAY
);
```

Like for the device, the interface status can be retrieved with the above method. It requires and addition `ifidx` argument to specify which interface is being operated on if there are multiple interfaces.

```
int usbHostClearInterfaceFeature (
  const uint8_t     addr,
  const uint8_t     endp,
  const uint16_t    ifidx,
  const uint16_t    feature,
  const uint8_t     idle       = DEFAULTIDLEDELAY
);
```

```
int usbHostSetInterfaceFeature (
  const uint8_t     addr,
  const uint8_t     endp,
  const uint16_t    ifidx,
  const uint16_t    feature,
  const uint8_t     idle       = DEFAULTIDLEDELAY
);
```

The interface set and clear features work as for the device methods, but just require an additional `ifidx` argument to select the interface, if there are multiple interfaces in the device.

```
int usbHostSetInterface (
  const uint8_t      addr,
  const uint8_t      endp,
  const uint16_t     ifidx,
  const uint16_t     altif,
  const uint8_t      idle       = DEFAULTIDLEDELAY
);
```

```
int usbHostGetInterface (
  const uint8_t      addr,
  const uint8_t      endp,
  const uint16_t     ifidx,
        uint16_t     &altif,
  const uint8_t      idle       = DEFAULTIDLEDELAY
);
```

The above two methods retrieve or set the interface alternative setting. Both methods require the `ifidx` parameter to select which interface to interact with. The get method has a reference to an `altif` parameter in which the value is returned. For the set method, the new alternative interface setting is passed in as `altif`.

Endpoint Control Methods

The last few control methods belong to controlling endpoints.

```
int usbHostGetEndpointStatus (
  const uint8_t      addr,
  const uint8_t      endp,
        uint16_t     &status,
  const uint8_t      idle       = DEFAULTIDLEDELAY
);
```

The above method fetches an endpoint's status, returning the value in the `status` parameter reference. There can be multiple endpoints in a device, and in this case, this is selected on the `endp` parameter,

```
int usbHostClearEndpointFeature (
  const uint8_t      addr,
  const uint8_t      endp,

  const uint16_t     feature,
  const uint8_t      idle       = DEFAULTIDLEDELAY
);
```

```
int usbHostSetEndpointFeature (
  const uint8_t     addr,
  const uint8_t     endp,
  const uint16_t    feature,
  const uint8_t     idle       = DEFAULTIDLEDELAY
);
```

The above two methods clear or set features in a device's endpoints and work in the same way as for the device feature methods. Again, as for the endpoint status method, the `endp` parameter will select the endpoint being indexed.

```
int usbHostGetEndpointSynchFrame (
  const uint8_t     addr,
  const uint8_t     endp,
  const uint16_t    &framenum,
  const uint8_t     idle        = DEFAULTIDLEDELAY
);
```

The above method fetches an endpoint's frame number, returning the value in the `framenum` argument reference. Again, as for the previous endpoint methods, the `endp` parameter will select the endpoint being indexed.

Data Transfer Methods

Having defined the required standard device control operation in the above control methods, all that remains is to define some data transfer methods. Data transfer can be from the host to the device (an OUT transfer) or from the device to the host (an IN transfer). In all cases, though, the host is the one that initiates the data transfer.

```
int usbHostBulkDataOut (
  const uint8_t     addr,
  const uint8_t     endp,
  const uint8_t*    data,
  const int         len,
  const int         maxpktsize,
  const uint8_t     idle         = DEFAULTIDLEDELAY
);
```

```
int usbHostBulkDataIn (
  const uint8_t     addr,
  const uint8_t     endp,
  const uint8_t*    data,
  const int         len,
  const int         maxpktsize,
  const uint8_t     idle         = DEFAULTIDLEDELAY
);
```

```
int usbHostIsoDataOut (
  const uint8_t     addr,
  const uint8_t     endp,
  const uint8_t*    data,
  const int         len,
  const int         maxpktsize,
  const uint8_t     idle         = DEFAULTIDLEDELAY
);
```

```
int usbHostIsoDataIn (
  const uint8_t     addr,
  const uint8_t     endp,
  const uint8_t*    data,
  const int         len,
  const int         maxpktsize,
  const uint8_t     idle         = DEFAULTIDLEDELAY
);
```

All four method have the same arguments. In the case of the OUT methods, the `data` parameter contains the data to be sent, with its length specified in `len`. The `maxpktsize` parameter specifies the maximum packet size to send. The value to use can be gleaned from the relevant endpoint descriptor (retrieved from the device when fetching the configuration descriptor, along with all the other descriptors, using the `usbHostGetConfigDescriptor` method (see above). The OUT methods will break the packet up into chunks of up to this size in order to send all the data in multiple OUT packets.

Similarly, the IN methods have the requested data returned in the `data` argument, up to the length requested (`len`). The method will break up the IN requests to meet the `maxpktsize` parameter.

Device API

The constructor for the device API is shown in the table below.

```
usbDevice (
  int                       node,
  usbDeviceDataCallback_t   datacb    = NULL
  std::string               name      = std::string(FMT_HOST "DEV " FMT_NORMAL)
);
```

The device constructor is very similar to that for the `usbHost` class, with the `node` and the `name` arguments the same. An additional argument, `datacb`, is provided which is a pointer to a callback function of type `usbDeviceDataCallback_t`. User code can provide a function pointer at construction and the function will be called for ever data transfer to the device, whether an IN or OUT. The `usbDeviceDataCallback_t` type, which sits in the usbDevice class's namespace, defines a function with the following prototype:

```
<userDeviceCBName> (
  const uint8_t      endp,
  uint8_t*           data,
  int                &numbytes
);
```

The callback function is provided with the endpoint being addressed in endp, with the direction of the access in bit 7 (high is an IN access, low is an OUT access). The data for an OUT access is sent in the `data` argument buffer pointer or is updated by the callback to return data for an IN access. The `numbytes` reference contains the data size for an OUT access or is updated by the callback for an IN access, indicating the size of the returned data.

Utility Methods

The device model has some utility methods to access some time information and control the simulation. These can be used both from the main program and in any registered callback function.

```
float usbDeviceGetTimeUs (
  void
);

void usbDeviceSleepUs (
  const unsigned    time_us
);

void usbDeviceDisconnect (
  void
);

void usbDevicReconnect (
  void
);

void usbDeviceEndExecution (
  void
);
```

The first method (`usbDeviceGetTimeUs`) from those about returns the current time in microseconds, with this time based on a clock count started from time 0. The second methods (`usbDeviceSleepUs`) will force the device to set no driving of the bus for the period, in microseconds, passed in with `time_us`. This should be used with caution and would normally only be used before the device has been run (see below). The device will not respond to packets from the host for the duration of the sleep period. The disconnect and reconnect methods basically control the devices pullup resistor to emulate connection or disconnection. The end execution method will terminate all program execution.

Under normal circumstances, the device is simply “run”, and will start responding to host packets. If a user callback function was provided at construction, then data

accesses can be processed with user code via this callback. The run method is show below:

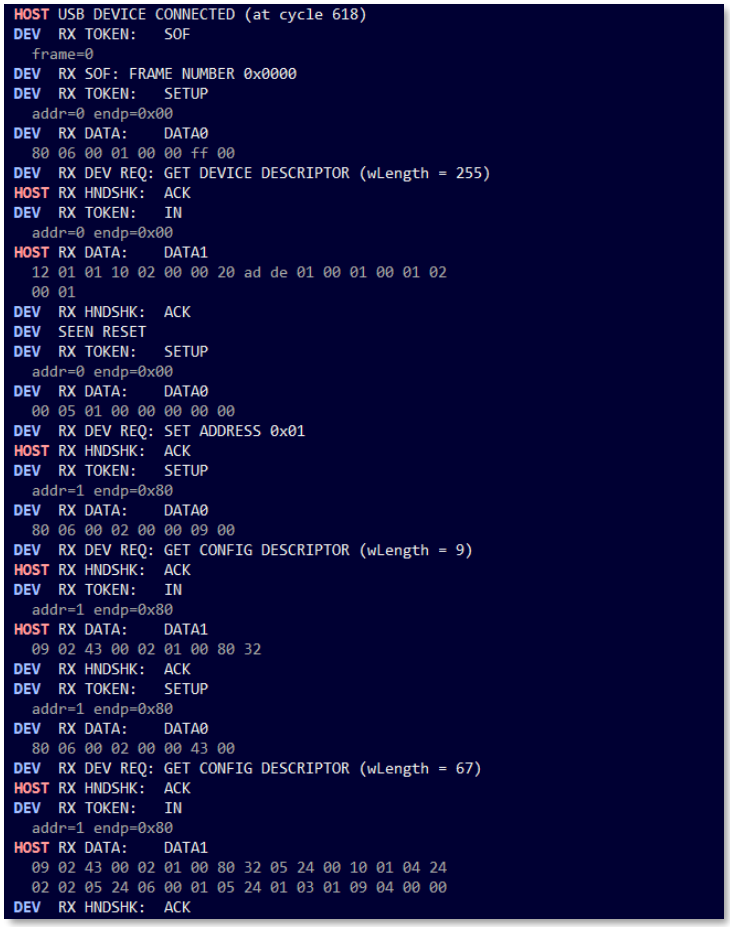
```
int usbDevicRun (
  const int         idle        = DEFAULT_IDLE
);
```

The only parameter is the optional `idle`. This defines a period, in clock ticks, to ide before generating packets on the line, to emulate some processing time . This has a default time of 4 cycles (as defined by `DEFAULT_IDLE`).

The method will not normally return. However, if an error occurred during the transaction, then `usbModel::USBERROR` is returned, or if a valid, but unsupported, response packet is received from the device then the methods return a status of `usbModel::USBUNSUPPORTED`. If a timeout occurred waiting for a response packet, then `usbModel::USBNORESPONSE` is returned. All these are considered as fatal errors.

Formatted Output

The models will produce formatted output of received packets. By default, these are colourised suitable for an xterm terminal. If the models are used in an environment where this will cause an issue then the model can be compiled with `USBNOFORMAT` defined, and the output will be printed as plain text. Colourised output can be an issue if wishing to save the output into a file for logging and future inspection. A script is provided (`model/scripts/decolor.sh`) which will strip the formatting codes to leave plain text when saving output to a file. An example formatted output is shown below:



Formatted outputs from the software models can be disabled by compiling the model code with `DISABLEUSBDISPPKT` defined.

Formatting Functions

In addition to the formatted output from the software models, functions are provided, for use within user code, to format raw bytes of various USB data suitable for printing to a console or text window. These are all defined in `usbFormat.h` and `usbFormat.cpp` and are contained within the `usbModel` namespace.

A set of functions for formatting descriptor data are provided which all have the same prototype:

```
int usbModel::fmt<DESC>Descriptor (
  char*             sbuf,
  const uint8_t*    rawdata,
  const unsigned    indent      = 0,
  const int         maxstrsize  = ERRBUFSIZE
);
```

The *<DESC>* part of the name can be one of `Dev` for device descriptor, `Cfg` for configuration descriptor, `If` for interface descriptor, `Ep` for endpoint descriptor, `HdrFunc` for a header function descriptor, `AcmFunc` for an ACM function descriptor, Union for a union function descriptor, `CallMgmt` for a call management function descriptor or `CfgAll` for a combined configuration descriptor. This last function will decode a configuration and all the other combined returned descriptors from a call to the `usbHostGetConfigDescriptor` method, with some hierarchical indenting.

The parameters consist of a point to an output string buffer (`sbuf`) to place the formatted data, a raw data buffer (`rawdata`) containing the descriptor bytes, an `indent` to specify any additional indenting required on the output and a `maxstrsize` to limit how much output is placed in `sbuf`.

In addition to the descriptor formatting functions there are a few low level functions, primarily used by the descriptor formatting functions, but provided for convenience.

```
const char* usbModel::fmtDescriptortype (
  const uint8_t     desc
);

const char* usbModel::fmtFuncDescSubtype (
  const uint8_t     subtype
);

const char* usbModel::fmtLineState (
  const unsigned    linestate
);
```

The first two take the descriptor type or sub-type and return a text string representing that type/sub-type. The last returns a string for the current line state as either `"J"`, `"K"`, `"SE0"` or `"SE1"`.

String descriptors are stored as 16-bit unicode values, even if the strig is just ASCII values. A couple of functions are provided to convert between the 16-bit and 8-bit representations.

```
int usbModel::fmtStrToUnicode (
  uint16_t*         dst,
  const char*       src,
  const int         maxstrsize  = MAXSTRDESCSTRING
);
```

```
int usbModel::fmtUnicodeToStr (
  char*             dst,
  const uint16_t*   src,
  const int         length,
  const int         maxstrsize  = MAXSTRDESCSTRING
);
```

The first function takes a string buffer (`src`) of 8-bit characters and places then in a 16-bit buffer (`dst`), up to a maximum length `maxstrsize`. The second function takes a 16-bit unicode buffer, with `length` amount of 16-bit data, and places in a character buffer (`dst`), limiting to `maxstrsize`.

Below is show, by way of example, a diagram showing the partial formatted output of the `fmtCfgAllDescriptor`:

```
Configuration Descriptor:

  bLength                 = 9
  bDescriptorType         = USB_CFG_DESCRIPTOR_TYPE
  wTotalLength            = 0x0043
  bNumInterfaces          = 0x02
  bConfigurationValue     = 0x01
  iConfiguration          = 0x00
  bmAttributes            = 0x80
  bMaxPower               = 0x32

..Header Function Descriptor:

    bLength               = 5
    bDescriptorType       = USB_FUNC_DESCRIPTOR_TYPE
    bDescriptorSubType    = HEADER
    bcdCDC                = 110

..Abstract Control Management Function Descriptor:

    bLength               = 4
    bDescriptorType       = USB_FUNC_DESCRIPTOR_TYPE
    bDescriptorSubType    = ABSTRACT CONTROL MANAGEMENT
    bmCapabilities        = 02

..Union Function Descriptor:

    bLength               = 5
    bDescriptorType       = USB_FUNC_DESCRIPTOR_TYPE
    bDescriptorSubType    = UNION
    bControlInterface     = 00
    bSubordinateInterface0 = 01

..Call Management Function Descriptor:

    bLength              = 5
    bDescriptorType      = USB_FUNC_DESCRIPTOR_TYPE
    bDescriptorSubType   = CALL MANAGMENT
    bmCapabilities       = 03
    bmDataInterface      = 01

..Interface Descriptor:

    bLength               = 9
    bDescriptorType       = USB_IF_DESCRIPTOR_TYPE
    bInterfaceNumber      = 00
    bAlternateSetting     = 00
    bNumEndpoints         = 01
```

Using the Models in Other Environments

The usbModel code has an abstraction layer in the `usbPliApi` class, which hides the details of the interface to the VProc virtual processor and the logic environment. The layers above this only interact with this environment through this abstraction layer, and so to integrate with other environments, only the `usbPliApi` class would need replacing. Although the methods of this class are not available to user code, this section will detail them as reference to porting the methods to another simulation setup, whether logic of purely software.

```
unsigned apiGetClkCount (
  int               delta
);
```

This method returns a clock tick count since time zero. So the real time depends on the clock frequency. For a 12MHz full-speed USB, each tick is ~83.3 nanoseconds. When porting, the underlying environment would need to have a model of time and convert to a tick count. The parameter allows for the retrieval of time, without advancing time (delta non-zero). This might be ignored by another implementation.

```
unsigned apiSendIdle (
  const unsigned    ticks
);
```

```
unsigned apiSendReset (
  const unsigned    ticks
);
```

The above two methods advance time by the specified number of clock ticks. The first with nothing driving the line, and the second whilst driving SE0 (normally only a host does this). When porting, the new environment would need to advance its sense of time, whilst allowing detection of the line state by a connected device to suspend or reset as appropriate, if the line state is held sufficiently long.

```
void apiWaitOnNotReset (
  void
);
```

The above method will block, waiting on reset to be removed. If the underlying environment has no concept of a power-on-reset, this method can return immediately.

```
void apiEnablePullup (
  void
);
```

```
void apiDisablePullup (
  void
);
```

The above two methods, in the VProc environment, drive the logic simulation lines with either weak or pull strength signalling to emulate the presence of a pullup (device) or pulldowns (host). When disconnected the lines are driven to Z. The main functionality

required, though, is for the device to indicate that it is connected (a pullup present) or not. This in another environment, the methods on a device need only change state to connected or disconnected. For a host, these could be a ‘don’t care’.

```
void apiHaltSimulation (
  void
);
```

Calling the above function terminates the simulation. For porting to another environment this need only indicate that the program needs to end, however that is affected.

```
void apiReset (
  void
);
```

The `apiReset` method (not to be confused with `apiSendReset`) tells the underlying code to reset all internal state, as at time 0.

```
unsigned apiReadLineState (
  void
);
```

The above method returns the line state as an unsigned value, with bit 0 containing the value on D+ and bit 1 the value on D-. This can be called at any time, and the underlying code must return the correct state of the line of the time as understood by the calling code.

```
void apiSendPacket (
  usbModel::usb_signal_t*    nrzi,
  const int                  bitlen,
  const int                  delay
);
```

```
void apiWaitForPkt (
  usbModel::usb_signal_t*    nrzi
);
```

The above two methods are the methods for sending and receiving NRZI encoded packets. Both use a buffer point of type `usb_signal_t` defined in the `usbModel` namespace. This is simply defined as:

```
typedef struct {
    uint8_t        dp;
    uint8_t        dm;
} usb_signal_t;
```

The `apiSendPacket` method takes a `usb_signal_t` buffer of NRZI encoded data, of length `bitlen`, and sends it over the USB lines, advancing the appropriate amount of time. An additional delay *before* the packet is sent is added, as specified by `delay`.

The `apiWaitForPkt` method will block until a complete NRZI packet has been received, and the packet is returned in `nrzi`.

Some constants are defined in the usbPliApi class that need to be available to the rest of the model code.

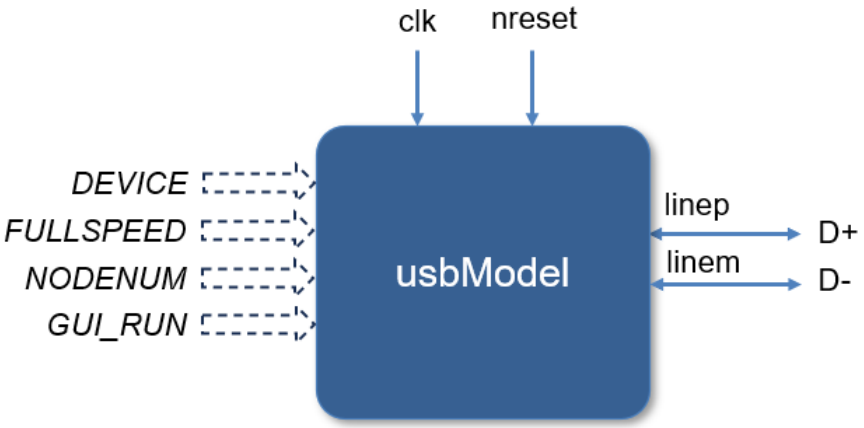
- `ONE_US`: number of ticks for a microsecond.
- `ONE_MS`: number of ticks for a millisecond.
- `IS_HOST`: value to flag if a host (false).
- `IS_DEVICE`: value to flag if a device (true).
- `MINRSTCOUNT`: Minimum clock ticks to constitute a reset on the line.
- `MINSUSPENDCOUNT`: Minimum clock ticks to constitute a suspend on the line.

The last two minimum counts, in the VProc environment, can be altered from the USB specifications, by defining `USBTESTMODE`, which shortens their length for test purposes. The feature is optional.

Verilog Environment

The usbModel Verilog Module

The USB software model use the VProc component to connect to a Verilog simulation environment. The VProc Verilog component is instantiated in a `usbModel` wrapper module, which has ports and parameters as shown in the diagram below:



The ports are very straight forward, with two signals to drive the D+ and D- USB serial data lines. The only other ports are for a clock (`clk`), running at 12MHz, and an active low reset signal (`nreset`).

Model Parameters

The usbModel Verilog module is common to both the host and device software models. The minor differences are controlled through the `DEVICE` parameter. If this is set to 0, then the module behaves for a host model. If it is non-zero, then it behaves as for a device. The selection between host or device is really only in the control of the pullup/pulldown functionality. When a host, the module will pulldown on both lines with a `weak0` (which may be disabled). When a device, the module will pullup on the `linep` port with a strength of `pull1`, which may also be controlled to be enabled or not (when

the line goes to highz, if not driven. It is the control of this pullup that allows emulation of connection or not.

The FULLSPEED parameter sets whether a full speed or low speed component. For a host configured module, this has no effect. For a device configured module, this swaps the pullup from the D+ to D- line. Note that this is for future proofing as the soft model does not support low speed.

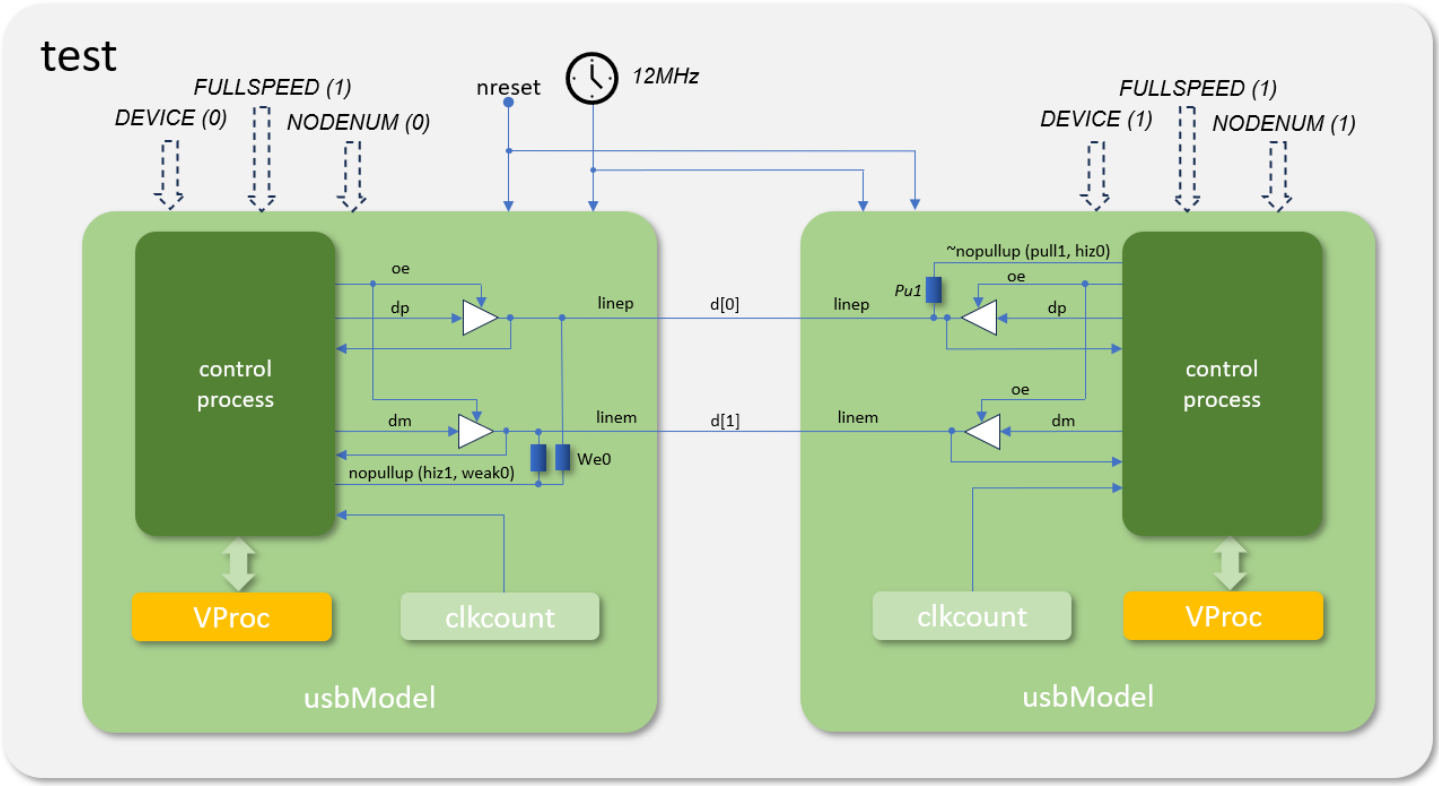
The NODENUM parameter is for the internal VProc instantiation to set which node it is, and thus which entry point code it calls (VUserMain<NODENUM>). These must be unique for each instantiated usbModel (or any other modules with a VProc instantiation).

The GUI_RUN parameter is meant to be 0 when running a batch simulator, and non-zero when running a GUI simulation. The usbModel module has the ability to stop or finish the simulation. If, however, a finish command is issued when GUI_RUN is non-zero, then the simulation will stop rather than finish if a finish command is issued. This allows for inspection of the waveforms before shutting down the GUI, even if the code issues a finish (useful for running a batch simulation).

Test Bench

An example test environment is provided in model/verilog/test. This instantiates both a host and device module and provides example user driver code for both (under usercode, in the test directory).

The diagram below shows the test bench setup, along with some more internal details of the usbModel Verilog wrappers and their parameter settings (GUI_RUN is set at simulation run time).



Compiling and Running Code

In the `test` directory there is a `makefile` that does all the compilation for the software as well as compile the logic and run simulations for a Questa simulation environment, as well as for ModelSim (`make ARCHFLAG=-m32`) and a separate make file for Icarus Verilog (`makefile.ica`) . Using the make command, the following commands (shown for Questa) can be used in a terminal to do various compilations and executions.

- **`make`**: Compile software, but do not compile or run simulation.
- **`make sim`**: Compile software and logic and start batch simulation without running.
- **`make run`**: Compile software and logic and start and run batch simulation.
- **`make simgui`**: Compile software and logic and start GUI simulation without running.
- **`make rungui`**: Compile software and logic and start and run GUI simulation.

The `makefile` will call the VProc make file to compile the code required for that component and will even check out the repository if it is not where it is expected. The commands to run a simulator (both batch and GUI) are useful if a debugger connection is required to debug software before the simulation starts. Note that if using `rungui`, the makefile will look for a `wave.do` file (or `waves.gtkw` for Icarus) to configure the waveform window. If it is not present then the simulation won't be run, allowing the user to add wave traces to the waveform window before proceeding. For Icarus Verilog an additional `debug` target is added which runs the simulation by encounters a `$stop` at time 0. This is like the `sim` target, but Icarus will not have loaded the `VProc.so` shared object if not run at all, so this allows debug attachment at a point where the shared object has been loaded.

Some user-defined `makefile` variables can be overridden on the `make` command line, in addition to the simulation command.

- `USRFLAGS`: default `-DUSBTESTMODE`
- `USRSRCDIR`: default `usercode`
- `USER_CPP`: default `VUserMain0.cpp VUserMain1.cpp`

The `USRFLAGS` variable is for user code to add any compilation command line options specific to user code. The default sets a definition which shorted some timings in the mode from the USB specifications for test purposes. Currently this is the times for detecting a reset and a suspension. To use the USB specification values, this can be set to ""— e.g. `make USRFLAGS="" run`.

The `USRSRCDIR` specifies which directory contains the user source code. The defaults to a directory in the `test` directory called `usercode`. If the code is located elsewhere, then this variable can be overridden. Absolute or relative paths are allowed.

The `USER_CPP` variable is a list of all the C++ user code (not the headers). By default this is just the two mains source files for `VUserMain`*x* code, as used in the test example. If user code has additional files, then this variable can be overridden to list *all* the relevant files, which are all expected to be in the directory specified by `USRSRCDIR`.

Of course, the `makefile` of the example test directory is just that and can be used as a basic to adapt for other environments and retargeted to other simulators. Then the defaults for the user overridable variables can be set to new, more useful, defaults.

The Demonstration Test Code

Some demonstration test code is provided in `model/verilog/test/usercode`. As much as possible, it is meant to give example of the use of each of the host and device API methods, tough the device model does not have any isochronous endpoints, and so the associated methods are not used (they are called in exactly the same way as for their bulk transfer equivalents. There's also no example of interrupt transfers as such, but this is a higher-level concept, where an endpoint is configured as an interrupt endpoint, and specifies a polling interval. The actual transfers still use bulk transfers, it's just that the driver code needs to schedule a transfer at the specified regular interval.

The host example code goes through an initialisation, much as one would expect on a real system (and maybe a bit more), and then does some data transfer. The sequence is as shown below:

1. Wait for a device to be connected.
2. Fetch the device descriptor.
3. Reset the device.
4. Set the devices address to 1.
5. Get the configuration descriptor (initial descriptor only)
6. Extract the total length for all descriptors.
7. Get the configuration descriptor with all associated descriptors.
8. Get the string descriptors.
9. Get the device status.
10. Get and set the device's configuration status.
11. Set and clear features for device, interfaces, and endpoints
12. Get an endpoint's synch frame number.
13. Send some bulk data to one of the endpoints.
14. Fetch some data from one of the endpoints.
15. Suspend the device.

For the device end, the user code is much simpler. On construction of the `usbDevice`, a callback function is registered (`dataCallback`), and then some delay is requested before call the `usbRunDevice` method, which connects the device to the line and starts waiting for packets. The callback is executed for ever data transfer and, in the example case, simply displays OUT data, or generates a incrementing pattern for In data requests.

Summary of APIs

Methods for usbHost

The following table summarises the methods for the `usbHost` model.

method	description
Utility methods	
usbGetVersionStr	Return a string in a buffer with version number of usbModel
usbHostSleepUs	Sleep (go idle) for a specified time
usbHostGetTimeUs	Get current time
usbHostEndExecution	End all execution of the program
usbHostWaitForConnection	Wait for a device to be connected
usbHostFindDescriptor	Fine a specific descriptor in configuration descriptor raw data
Device Control Methods	
usbHostGetDeviceDescriptor	Fetch a device descriptor
usbHostGetConfigDescriptor	Fetch a configuration descriptor
usbHostGetStrDescriptor	Fetch a string descriptor
usbHostSetAddress	Set the address of the connected device
usbHostGetDeviceStatus	Fetch the device status
usbHostGetDeviceConfig	Fetch the device configuration (disabled/enabled)
usbHostSetDeviceConfig	Set the configuration of the connected device (enable)
usbHostClearDeviceFeature	Clear a device feature of the connected device
usbHostSetDeviceFeature	Set a device feature of the connected device
Interface Control Method	
usbHostGetInterfaceStatus	Get an interface's status
usbHostClearInterfaceFeature	Clear an interface's feature
usbHostSetInterfaceFeature	Set an interface's feature
usbHostSetInterface	Set the alternative interface setting of a particular interface
usbHostGetInterface	Get the alternative interface setting of a particular interface
Endpoint Control Methods	

usbHostGetEndpointStatus	Get an endpoint's status
usbHostClearEndpointFeature	Clear an endpoint's feature
usbHostSetEndpointFeature	Set an endpoint's feature
usbHostGetEndpointSynchFrame	Get an endpoint's current synch frame number
Data Transfer Methods	
usbHostBulkDataOut	Send bulk data to an endpoint on the device
usbHostBulkDataIn	Fetch bulk data from an endpoint on the device
usbHostIsoDataOut	Send isochronous data to an endpoint on the device
usbHostIsoDataIn	Fetch isochronous data from an endpoint on the device

Methods for usbDevice

The following table summarises the methods for the `usbDevice` model.

method	description
Utility methods	
usbGetVersionStr	Return a string in a buffer with version number of usbModel
usbDeviceGetTimeUs	Get time in microseconds
usbDeviceSleepUs	Sleep (go idle) for given time in microseconds
usbDeviceDisconnect	Disconnect device (remove pullup)
usbDeviceReconnect	Reconnect device (enable pullup)
usbDeviceEndExecution	End all program execution
Run Methods	
usbDeviceRun	Run the device to process packets

Functions for Formatting Data

In the `usbModel` namespace, a set of functions are provided to process raw data into various printable formatted text. These are used by the model but are available for use in user code. The table below summaries these functions

function	description
Low Level Functions	
fmtFuncDescSubtype	Generate a string representation of a subtype

fmtDescriptorType	Generate a string representation of a descriptor type
fmtLineState	Generate a string representation of the line state ("J", "K", "SE0", "SE1")
Conversion Functions	
fmtStrToUnicode	Convert 8-bit string data to 16-bit unicode
fmtUnicodeToStr	Convert 16-bit unicode data to 8-bit string data
Decsriptor Formatting Functions	
fmtDevDescriptor	Generate a formatted device descriptor
fmtCfgDescriptor	Generate a formatted configuration descriptor
fmtIfDescriptor	Generate a formatted interface descriptor
fmtEpDescriptor	Generate a formatted endpoint descriptor
fmtHdrFuncDescriptor	Generate a formatted header function descriptor
fmtAcmFuncDescriptor	Generate a formatted ACM function descriptor
fmtUnionFuncDescriptor	Generate a formatted union function descriptor
fmtCallMgmtFuncDescriptor	Generate a formatted call management function descriptor
fmtCfgAllDescriptor	Generate a formatted output for all descriptors returned with a configuration descriptor