Preface

Introduction to USB

Introducing the USB stack

Finding a USB Target

Getting to know your USB Target

Talking with your USB Target

Preface

What you have before you now, is a guide to developing USB drivers for he AmigaOS4 USB stack. The intention of this guide is not to give full insight into the inner workings of the USB bus. In fact, it propably won't even give give you enough general USB knowledge to form your own transfer requests for the USB bus. This job is already taken by the USB specification found at www.usb.org, which contain all the information needed on that subject - and a lot of other information as well, which you can just skip. Afterall, as a driver developer you don't need to know about USB cable specs, connector tolerances or electrical signal timings.

What this guide aims at is teaching you how to develop USB drivers for the AmigaOS4 USB stack. The USB API documentation gives a description of each API function call. The includes files gives comments on bits and pieces. This guide is intended to bind these informations together to get a clear understanding of what to do (and what not to do).

You should be somewhat familiar with the USB bus before you start reading this guide. Not that you won't succeede without it, but you'll have a much easier job learning what needs to be learnt if you know a little of USB beforehand.

If sections of this guide is unclear or contains erroneous information, please contact the author so that it can be corrected in a future revision.

Thomas Graff Thøger
graff@amigausb.dk

Date	Version	Description
19-09-2004	0.1	Initial draft. Covers USB and USB stack introductory chapters and information on locating, claiming and configuring USB Functions and Interfaces
09-10-2004	0.2	Added section on performing IO with USB EndPoints.
17-10-2004	0.3	Added clarification of the content of the EndPoint number to give to USBGetEndPoint().

Introduction to USB

Introduction

Although the welcoming text for this guide states that you should be familiar with the USB bus before reading, I’m sure some of the readers never did take the time to familiarize yourself with USB anyhow. So a little USB brush-up is in place at this early stage. Those who have just laid down the USB spec may find that skipping this section alltogether is not that much of a loss.

What is the USB bus?

The USB bus is a serial bus capable of hot-swapping devices in and out of the bus topology at any given time. The bus is based on a four wire connection, consisting of two power lines, and two data lines.
The USB system is based on a structure consisting of a single main host controlling the bus, and a multitude of slave devices. The host device is responsible for initiating all transactions on the USB bus - nobody ever speaks without being asked by the host.
The bus topology of the USB bus can be seen as a logical star topology. The host sits in the middle, and all the connected devices sits around it, waiting for commands from the host. Physically, however, the USB bus has a tree-like topology: The host holds a number of USB ports, to which USB devices can be connected. If there is not enough ports to hold the devices required, hubs can be attached to the ports. Hubs have one connection towards the host (the upstream connection), and several ports for connecting with other USB devices (the downstream connections).

An important role of hubs in the USB bus, is their handling of device attachment and detachment. Hubs are responsible for signalling to the USB System Software (USB stack) when devices are attached or detached from the topology. Once signalled he USB stack handles the software side of attachment/detachment, which includes launching drivers, signalling drivers to terminate, as well as other background tasks. An important thing to note about this is, that due to the hot-swap nature of USB, a driver must expect that the USB device it is controlling can disappear from the USB bus at any given time.

Anatomy of a USB device

Each device connected on the USB bus is also called a USB Function- a single USB entity connected on the USB bus.
Each Function basically hosts a number of Interfaces, which might be thought of as logical units within the Function. For instance a USB keyboard may hold in it an Interface for the keyboard itself, but also an Interface for a mouse pad or similar built-in, but functionally different, functionality.
Each Interface has a descriptor describing the type and properties of the Interface, so you can destinguish interfaces from one another. To actually communicate to a USB Function (or, usually the Interfaces within it) you use EndPoints. Each EndPoint is a communication channel to the USB Function/Interface. Sometimes EndPoints are also refered to as pipes.
One EndPoint is special. EndPoint 0 (zero), also known as the Default Control Pipe

(DCP), is always present in a USB Function. Using this EndPoint you control generic features of the USB Function - configuring, status, etc which are general to the USB Function. Other than the DCP, all EndPoints belong to individual Interfaces. So talking to anything but EndPoint 0 will have you talking with an Interface of the USB Function.
There is much more to say about Functions, Interfaces and EndPoints. I'll refer you to the USB specification for that, or I could fill many pages here with something already described elsewhere.

Transfers on the USB bus

All transfers on the USB bus is performed between the host, and an EndPoint at a Function. The USB bus describes four types of transfers:

- Control
- Interrupt
- Bulk
- Isochronous

Without going into too much detail the following can be said about the four transfer types:

Control xfer is generally used for sending immediate commands to a USB EndPoint, such as reading status, setting or clearing switches, reading descriptors etc.

Interrupt xfer is used for xfers needing a regular service interval. E.g. checking if something has been attached or detached from a USB hub, which must be done regularly. Another example is reading keyboard keypress data, which must also be done regularly.

Bulk xfer is a transfer type designed for moving larger blocks of data in irregular intervals. This is used for things such as transfering data to/from USB harddisks and reading image data from a scanner.

Isochronous xfer is a timely transfer type. It guarantees timely delivery of data at the expense of data integrity. This is used for things like streaming of video or audio where the importance is placed on realtime constraints rather than the correctness of data - e.g. it doesn't matter as much if a snip of audio is malformed, as if the audio looses sync with its videostream.

The transfer type is hardcoded into the EndPoint to which a transfer is made. The EndPoint type is thus generally already defined in the communication specification for the type of USB device you are developing a driver for.

Introducing the USB stack

Introductary

USB concepts

The AmigaOS USB stack is, luckily enough, built upon the information held in the USB specification. As such the topological and structural design of the USB stack is much alike what is laid out in the USB specification.

The USB stack generally operates with bus entities such as

- UsbFunction representing a physical USB Function attached on the USB bus.
- UsbInterface representing an Interface of a USB Function.
- UsbEndPoint representing an EndPoint in a USB Function or Interface.

These entities forms the basis for all operations regarding the USB bus in the USB stack. To simplify the rest of this guide, a USB Function or Interface targeted by a Function Driver is termed a USB Target, rather than mentioning both Interfaces as Functions, where a section applies to both.

To actually be able to communicate with EndPoints, the USB stack defines an IO request structure, known as the USBIOReq. How this is created and used will be covered in a later chapter on transfers.

The USB spec also defines a concept known as descriptors, which is a data structure with a fixed header, followed by free form data. Descriptors are what binds the generic nature of USB with the individual drivers. By reading and understading descriptor contents a driver can find out information about Functions, Interfaces and EndPoints, as well as other information depending on the type of device in question.

The above three conceptual main categories forms the foundation of the USB stack. There is much more, but at this early overview stage, there is now need to complicate things any further. We'll get to that later on.

Function Drivers

In context of the AmigaOS USB stack, a driver for a USB Function, or an Interface in a USB Function, is called a Function Driver, or just an FD.
The USB stack supports two main types of Function Drivers: SelfStarting FDs and AutoStarting FDs. The difference between them lies in the way in which they are started.

SelfStarting FDs are just ordinary programs which opens the USB stack device, locates the USB Function to use and start using it. There is no surrounding fuss - it's just a normal program you can run from Workbench or a CLI.

AutoStarting FDs, on the other hand, are Exec libraries which the USB stack itself will open and use when a USB device which the FD can handle is attached.
If your FD is going to control USB Functions which offers immediate functionality to the user (like e.g. hubs, keyboards, mice, harddisks etc.) it makes life easier for the

user if the driver is automatically started when such a Function is attached. In such a case the FD should be made as an AutoStarting FD - once the Function is attached it immediately starts offering its services to the user.
If your FD is going to control USB Functions which does not offer immediate functionality to the user, or which depends on some other software to be usefull (e.g. printers, scanners, ethernet adapters etc.) there is no need for the driver to be around at all times. In such cases a SelfStarting FD might be a more suitable choice. In the end, however, it is up to you to make a clever decision.

For reasons of simplicity this guide will concentrate on SelfStarting FDs thru most of the chapters, due to the fact that It is simpler to get to grasps with SelfStarting FDs. Also, debugging of normal programs are much easier than debugging Exec libraries, making it easier to experiment while working on a SelfStarting FD.
The methods laid out for SelfStarting FDs also hold for AutoStarting FDs. SelfStarting FDs can always be changed into an AutoStarting FD at a later time.

Choosing a driver style

A USB Function is always bound to a USB device class - e.g. Hub, HID, MassStorage etc. - indicating the type of functionality contained within the Function. This class can be used by an FD to determine if it can handle that specific USB Function.
Not all USB Functions can, however, be fitted into a single class. Such Functions are multiclass, and their content class cannot be derived from the Function alone.
In this case one has to look at the Interfaces in the Function. Each Interface also has a class indicator. By looking at the Interface class it is thus possible to identify the type of service delivered by the Interface. As opposed to a Function, an Interface can’t be multiclass. One Interface always belongs to one class only.

As a functionality classes can be assigned to both Functions as well as Interfaces there are two different approaches to creating a Function Driver: It can either focus on a Function or an Interface.

Focusing on a Function (termed a Function based FD) allows the FD to take full control of that Function. That is, the FD is free to reconfigure the entire Function as it sees fit, and use as many of the Interfaces in the Function as it can handle. This is usefull for vendor specific drivers for devices with more than one Interface where you need to know that all the Interfaces you have belongs to the same Function. Take for instance a USB phone device where the microphone and speaker are different Interfaces. Here you’d like your driver to handle both Interfaces, and be sure that they are both from the same phone device.
The price you pay for using a Funtion based FD is that you have to do everything yourself. If you own the entire Function no other FDs can step in and handle specific Interfaces within that Function. Thus, you have to handle all required aspects of the Function without any outside help.

Focusing on an Interface (termed an Interface based FD) only gives your FD access to one Interface, although many Interfaces may exist within an attached Function. While this may seem limited, most USB devices are logically split into multiple Interfaces each covering a selfcontained part of the Function. Thus an Interface based

FD will not need to know about any other Interfaces.
The important aspect of Interface based FDs is, that they allow the USB stack to assign different FDs to different Interfaces in the same Function. It thus allows FDs to cooperate in handling all the various functionalities presented by the Interfaces of a Function.

When developing FDs it is adviced that you develop Interface based FDs. Only if you need close linkage between various Interface types, or if you are doing vendor or product specific FDs should you create Function based FDs.

USB stack datatypes

Before going into descriptions of how to use the USB stack, a short introduction to the datatypes found in the stack might be a good idea. Use it as an introduction now, and as a reference later, when reading the rest of this manual.

UsbRawFunction

The UsbRawFunction acts as a searching and reference key in the USB stack Function lists. It represents a USB Function on the USB bus, but it does not grant you access to communicate with the USB Function. For communication you will need a UsbFunction, which is obtained on the basis of a UsbRawFunction.

UsbFunction

The UsbFunction is the basis for operation for a Function based FD. It represents a USB Function on the USB bus, but, unlike the UsbRawFunction, the UsbFunction grants you access to communicate with the Function. Or rather, it grants you access to find UsbInterfaces and UsbEndPoints in the Function with which you then communicate.
A UsbFunction is created from a UsbRawFunction when you claim ownership of the Function. Once you have a UsbFunction you are guaranteed to be the only user of that Function.

UsbRawInterface

The UsbRawInterface acts as a searching and reference key in the USB stack Interface lists. It represents a USB Interface on the USB bus, but it does not grant you access to communicate with it. For communication you will need a UsbInterface, which is obtained on the basis of a UsbRawInterface.

UsbInterface

The UsbInterface is the basis for operation for an Interface based FD. It represents a USB Interface in a USB Function on the USB bus, but, unlike a UsbRawInterface, a UsbInterface grants you access to communicate with the Interface. Or rather, it grants you access to find the UsbEndPoints which make up the Interface so you can communicate thru them.

UsbEndPoint

A UsbEndPoint is the basis for all communication with a USB Function or Interface.

All communication takes place thru a UsbEndPoint.
A UsbEndPoint belongs to either a UsbFunction or a UsbInterface. You get UsbEndPoints from their parent UsbFunction or UsbInterface, depending on which they belong to. One notable exception to this is the Default Control Pipe of a USB Function. Due to its general usage the Default Control Pipe (aka EndPoint 0) is part of both a UsbFunction and all of the UsbInterfaces in the USB Function.

USBIOReq

This datatype is an extension of an Exec IORequest structure. It is used for reading and writing data from/to a UsbEndPoint by performing Exec device-style IO operations (DoIO(), SendIO() etc.).

USBNotifyMsg

The USB stack includes a simple notification system used for sending notifications to FDs if the USB Function or Interface they control has been detached from the USB bus. The USBNotifyMsg is the message used for sending such notifications.

USBFDStartupMsg

When the USB stack starts an AutoStarting FD the FD is given a USBFDStartupMsg as argument. This structure holds information on which USB Target the FD should take control of.

USB datatypes

Apart form the above USB stack datatypes there are a set of datatypes defined by the USB specification. Although not directly defined by the USB stack you will come across these structures when working with the USB stack.
One thing which is very important to notice is that these structures are USB bus structures. This means that they are all in LittleEndian data format, as opposed to the Amiga which operates in BigEndian data format. Whenever a 16 bit integer is present in a USB datatype you must be sure to perform endianness conversion (swaping MSB and LSB of the word), or you will end up reading or writing wrong values.
A macro has been made in the include file "usb/usb.h" which performs this for 16 bit integers, named LE_WORD(). Use it for both reading and writing to 16 bit fields in the USB datatypes.
The include file "usb/usb.h" also holds the structure definitions described in this section. Here you will also find constant definitions used in the structures.

USBBusSetupData

The USBBusSetupData structure is a structurization of the USB Setup Data packet used for initiating a Control transfer to a Control EndPoint.
When using a Setup stage in a USBIOReq this is the data structure which io_SetupData must point to, and its size which must be stored in io_SetupLength.

USBBusDscHead

This structure is at the head of all USB descriptors. You'll propably never encounter

this on its own, rather embedded into a USB descriptor of some sorts. It describes the type and size of a descriptor.

USBBusDevDsc

This is a structurization of a USB Device Descriptor. It holds basic information on a USB Function attached to the USB bus.

USBBusCfgDsc

This is a structurization of a USB Configuration Descriptor. It describes the properties of one possible configuration of a USB Function. Each USB Function have one or more possible configurations, but only one can be active at any time.

USBBusIntDsc

This is a structurization of a USB Interface Descriptor. It describes the properties of a single Interface within a USB Function. A USB Function may have zero or more Interfaces, depending on the chosen configuration of the USB Function.

USBBusEPDsc

This is a structurization of a USB EndPoint Descriptor. It describes the properties of a specific EndPoint in a USB Function. A USB Function will have one or more EndPoints depending on the configuration chosen for the USB Function.

Locking of datatypes

The USB stack is a complex piece of work. Bear in mind that the USB bus is a bus architecture allowing the user to plug and unplug devices while the computer is powered on. The USB stack must therefore perform cleanup when a USB Function is unplugged (detached).
When the Function is detached there may, however, still be software using the Function and referencing structures in memory related to the just detached USB Function. The USB stack must therefore keep track of who is using which Functions (or the Interfaces within it). For this reason a locking system has been created.
As long as a piece of software has a lock on either a Function or an Interface the USB stack will not remove the in-memory structures used for referring to that Function/Interface even though the physical USB Function is detached. In order for external programs to work reliably with the USB stack, they must therefore adhere to the locking rules of the USB stack.
The locking rules of the USB stack will come to show later in this guide. They are also explained in the usbsys.device documentation for the individual API calls.

Finding a USB Target

Introduction

The first thing a FD needs to do, is to find the USB Interface, or Function (the Target), that it wants to control. This is a multi-step operation, due to the nature of the USB bus design.
This section will go thru the method of finding a suitable USB Target for a FD for both Interface and Function based FDs.

Finding a USB Interface

An Interface based FD will be developed with a type or make of USB Interface to control in mind. This is typically defined by either a specific USB vendor and/or product number, or a USB Interface class.

When run, an FD starts looking for a USB Interface to handle. This is done by successively calling to USBFindInterfaceA() to find out which Interfaces are available. Each call returns a reference to a USB Interface matching the search criteria. USBFindInterfaceA() will return one UsbRawFunction reference every time it is called. This UsbRawInterface has been locked once to ense that the referenced Interface is not expunged while we use it. If doing successive USBFindInterfaceA() calls the first returned reference must be unlocked *after* having obtained the next reference, by calling USBUnlockInterface(). Unlocking must not be performed until after obtaining the next Interface reference, as USBFindInterfaceA() uses the first Interface reference to return the next. Unlocking before calling USBFindInterfaceA() could potentially result in the UsbRawInterface given to USBFindInterfaceA() being expunged by the time the reference is used inside the function call.
Having found the UsbRawInterface to use, the driver claims it by calling USBClaimInterface(). By claiming the UsbRawInterface the FD ensures unique access to the Interface, and is able to communicate with the Interface. Furthermore the FD bind an (already initialized) message port to the Interface for receiving notifications. After having claimed the Interface the temporary lock set on the UsbRawInterface by USBFindInterfaceA() can be cleared - the claiming of the UsbRawFunction (resulting in a UsbFunction) is in itself an indirect lock on the Function. The unlocking is, as mentioned above, done by calling USBUnlockInterface().
The proper way of iterating through possible USB Interfaces with USBFindInterfaceA() is therefore:

```
prevrawifc = NULL;
ifc = NULL;
do {
      rawifc = USBFindInterfaceA( prevrawifc, ptags ); // Get Inter-
                                                          face reference
      if ( prevrawifc ) {
            USBUnlockInterface( prevrawifc ); // Unlock previously
                                                 examined Interface
      }

      if ( rawifc ) {
            ifc = USBClaimInterface( rawifc, ourifcref, msgport ); //
```

```
                                    Claim Interface and bind to msg port
            if ( ifc ) {
                  USBUnlockInterface( rawifc ); // Unlock Interface
                                                   (undo lock made by USBFind-
                                                   InterfaceA() now that we
                                                   own the Interface)
                  break; // Exit loop - Interface has been claimed
            } // Interface already claimed by other. Keep searching
                        for another Interface
      }

      prevrawifc = rawifc;
} while ( rawifc );

if ( ifc ) {
      DoYourStuffWithTheInterface(); // Configure and handle the In-
                                        terface (driver body)
      USBDeclaimInterface( ifc ); // Declaim Interface after use
} // else do nothing - fall through as we didn't find a suitable In-
     terface
```

The above code assumes a message port has already been initialised and is pointed to by the variable *msgport*, and that *ptags* points to a TagItem array holding properties of the needed USB Interface type. The *ourifcref* variable is an anything-but-NULL-goes value which the USB stack will pass to the FD as a reference in notification messages sent to the message port given in USBClaimInterface(). It could be the UsbRawInterface reference, it could be a pointer to an FD-private structure holding USB Interface related data, it could be a constant - it's all up to you.

Once claimed the FD has ownership of the Interface it wants to control, and a UsbInterface reference for use in further calls to the USB stack.

Finding a USB Function

A Function based FD will be made with a specific type or make of USB Function in mind, just like an Interface based FD. This can be defined by either a specific vendor and/or product number, a USB Function class and USB Function subclass.
As you will discover, Function based FDs locate their target Function much like an Interface based FD locates its target Interface.

When run, the FD starts looking for a USB Function to handle. This is done by successively calling USBFindFunctionA(), each call returning a reference to a USB Function matching the search criteria.
USBFindFunctionA() will return a UsbRawFunction reference. The UsbRawFunction has been locked by USBFindFunctionA() to ense that the referenced Function is not expunged while in use. If doing successive USBFindFunctionA() calls the first returned reference must be unlocked after having obtained the second reference, by calling USBUnlockFunction(). Unlocking must not be performed until after obtaining the next Function reference, as USBFindFunctionA() uses the first Function reference to return the next. Unlocking before calling USBFindFunctionA() could potentially result in the reference given to USBFindFunctionA() being expunged by the time the reference is used inside the function call.

Having found the UsbRawFunction to use, the driver claims it by calling USBClaimFunction(). By claiming the UsbRawFunction the FD ensures unique access to the Function, and is able to bind an (already initialized) message port to the Function for receiving notifications.
After having claimed the Function the temporary lock set on the Function reference by USBFindFunctionA() can be cleared - the claiming of the UsbRawFunction (resulting in a UsbFunction) is in itself an indirect lock on the Function. The unlocking is, as mentioned above, done by calling USBUnlockFunction().
The proper way of iterating through possible USB Functions with USBFindFunctionA() is therefore:

```
prevrawfkt = NULL;
fkt = NULL;
do {
      rawfkt = USBFindFunctionA( prevrawfkt, ptags ); // Get Function
                                                          reference
      if ( prevrawfkt ) {
            USBUnlockFunction( prevrawfkt ); // Unlock previously ex-
                                                    amined Function
      }

      if ( rawfkt ) {
            fkt = USBClaimFunction( rawfkt, ourfktref, msgport ); //
                                       Claim function and bind to
                                       msg port
            if ( fkt ) {
                  USBUnlockFunction( rawfkt ); // Unlock Function
                                          (undo lock made by USBFind-
                                          FunctionA() now that we own
                                          the Function)
                  break; // Exit loop - Function has been claimed
            } // Function already claimed by other. Keep searching
                  for another Function
      }

      prevrawfkt = rawfkt;
} while ( rawfkt );

if ( fkt ) {
      DoYourStuffWithTheFunction(); // Configure and handle the Func-
                                       tion (driver body)
      USBDeclaimFunction( fkt ); // Declaim Function after use
} // else do nothing - fall through as we didn’t find a suitable
     Function
```

The above code assumes a message port has already been initialised and is pointed to by the variable *msgport*, and that *ptags* points to a TagItem array holding properties of the required USB Function type. The *ourfktref* variable is an anything-but-NULL-goes value which the USB stack will give the FD as a reference in notification messages sent to the message port given in USBClaimFunction(). It could be the UsbFunction reference, it could a pointer to an FD-private structure holding USB Function related data, it could be a constant - it’s your choice.

Once claimed the FD has ownership of the Function it wants to control, and a Usb-Function reference for use in further calls to the USB stack.

Getting to know your USB Target

Introduction

When an FD has found its USB Target, it knows who to communicate with. But it does not know about the properties of its Target, other than the fact that the FD believes it can handle the Target based on vendor/product/class information.
It is now time to find out more about the just aquired USB Target.

Getting to know your USB Interface

With USBFindInterfaceA() the FD has found an Interface which suits its basic needs. Now it’s time to get to know that Interface better.

Each USB Interface contains a descriptor which describes the properties of that Interface, as well as the EndPoints it consists of. This descriptor, or rather list of descriptors, is obtained by calling USBIntGetAltSettingA().
The list of descriptors you now have, contains descriptors for the Interface and all EndPoints belonging to it. There may also be other descriptors in the list, depending on the class specification for the Interface class. So don’t be alarmed if you stuble upon something you don’t recognize in the descriptor list.
What you now have to do is traverse the descriptor list using USBNextDescriptor() and collect the information you need. Normally this will be extracting information on the Interface, when you come across the Interface Descriptor, and determine which EndPoints you will be using, when passing EndPoint Descriptors in the list.

The following code snippet will traverse the descriptor list of an Interface, and identify the Interface descriptor, as well as EndPoints.

```
// ...
struct USBBusDscHead *dsclist; // Configuration descriptor list ptr
struct USBBusDscHead *dsc; // A descriptor in the cfg list
// ...
dsclist = USBIntGetAltSettingA( openreq, ifc, NULL ); // Get Inter-
                                                        face info.
dsc = dsclist; // Init traversal
while ( dsc ) {
      puts( " Descriptor\n" );
      printf( "  • Type: %ld\n", (ULONG) dsc->dh_Type );
      printf( "  • Length: %ld\n", (ULONG) dsc->dh_Length );

      if ( dsc->dh_Type == USBDESC_INTERFACE ) {
            puts( "It’s the Interface descriptor!" );
      }
      if ( dsc->dh_Type == USBDESC_ENDPOINT ) {
            printf( "It’s the descriptor for EndPoint %ld\n", (ULONG)
                  ((struct USBBusEPDsc *) dsc)->ed_Address );
      }

      dsc = USBNextDescriptor( dsc ); // Get next descriptor
```

```
}
// ...
USBFreeDescriptors( dsclist ); // Free Interface descriptor list af-
                                  ter use
```

Note that the descriptor list returned by USBIntGetAltSettingA() is a copy of the current configuration of the Interface. Once you've inspected the descriptors and don't need them anymore, they must be freed by calling USBFreeDescriptors().

Getting to know your USB Function

Read the Device Descriptor

At this point you have a UsbFunction which you know fits the search criteria set in USBFindFunctionA(). But you may want to know more about the USB Function you're about to take control of.

Each USB Function hold in it a descriptor which describe its basic properties. This is the Device Descriptor. The Device Descriptor is obtained from a UsbFunction by calling USBEPGetDescriptorA(). USBEPGetDescriptorA() is a USB stack API call for reading single USB descriptors from a USB Function. Due to its general use some control arguments must be set correctly to read a Device Descriptor. Further more you will have to specify which UsbEndPoint to read the descriptor from. For a Device Descriptor this will always be EndPoint 0 (the Default Control Pipe).
We've not dealt with finding EndPoints yet, so just accept that the code snippet below uses EndPoint 0.

```
// ...
struct USBBusDevDsc     *devdsc;

// Get the device descriptor
devdsc = (struct USBBusDevDsc *) USBEPGetDescriptorA( ioreq,
                        USBGetEndPoint( fkt, NULL, 0 ),
                        USBSDT_TYP_STANDARD | USBSDT_REC_DEVICE,
                        USBDESC_DEVICE, 0, NULL );
if ( devdsc ) {
      // .. do your examination / information extraction

      // Cleanup after use
      USBFreeDescriptors( (struct USBBusDscHead *) devdsc );
}
// ...
```

Here it is assumed that the variable *fkt* is the claimed UsbFunction the FD is going to use, and that *ioreq* is a USBIOReq request previously allocated with USBAllocReqeustA() - something we'll go into more detail with later on.

Configure the Function

Now the FD knows about the Function it has obtained. However, it still knows nothing about the Interfaces contained within the Function. For this information to be accessible, the FD needs to configure the Function.
At this point in time the UsbFunction may or may not be configured by the USB

stack, so from a FD perspective a newly obtained UsbFunction is always unconfigured. Never expect the Function to be configured, and never trust an already selected configuration. It is as simple as that.

Selecting a configuration implies reading the available configurations from the USB Function, and then selecting a suitable one. This is generally a tricky job, and often you will find yourself selecting configuration 0, or a pre-specified configuration from the specification of the device you are going to handle.

A single configuration consists of a list of USB descriptors read from the USB Function. This list holds a configuration descriptor, followed by any number of Interface descriptors and EndPoint descriptors, as well as some device specific descriptors. Each configuration has an ID number which sets it apart from other possible configurations in the USB Function. These IDs range from zero and up to the maximum number of possible configurations in the USB Function.
The list of descriptors is obtained from the UsbFunction by calling USBFktGetCfgDescriptorsA(). You specify the UsbFunction as well as the ID of the configuration you whish to inspect, and receive the first USB descriptor in the list of descriptors which make up the configuration.
Using USBNextDescriptor()/USBPrevDescriptor() calls you can now traverse the list and inspect which Interfaces and EndPoints are available in that specific configuration.
One EndPoint you will never find, however, is EndPoint 0. This EndPoint is always available and is not part of any configuration.

The descriptor pointers returned by USBFktGetCfgDescriptorsA(), USBNextDescriptor() and USBPrevDescriptor() all starts with a USBBusDscHead structure. By evaluating the dh_Type field you can find the type of descriptor, and dh_Length specifies the byte size of the descriptor, including the USBBusDscHead structure.
The following example program will dump the dh_Type value of all descriptors in configuration zero.

```
// ...
struct USBBusDscHead *dsclist; // Configuration descriptor list ptr
struct USBBusDscHead *dsc; // A descriptor in the cfg list
// ...
dsclist = USBFktGetCfgDescriptorsA( openreq, fkt, 0, NULL ); // Get
                                                        configuration.
dsc = dsclist; // Init traversal
while ( dsc ) {
      puts( " Descriptor\n" );
      printf( "  • Type: %ld\n", (ULONG) dsc->dh_Type );
      printf( "  • Length: %ld\n", (ULONG) dsc->dh_Length );

      dsc = USBNextDescriptor( dsc ); // Get next descriptor
}
// ...
USBFreeDescriptors( dsclist ); // Free configuration descriptor list
                                  after use
```

Note that the above example finishes off by freeing the descriptor list returned by

USBFktGetCfgDescriptorsA().
The standard dh_Type values are defined in the usb/usb.h include file.

When you have found (or chosen) the configuration to use, you are ready to configure the USB Function. This is done using the USBSetFktConfigurationA() API call:

```
// ...
if ( USBSetFktConfigurationA( ioreq, fkt, cfgdsc, NULL ) ==
     USBSETCONFIG_OK ) {
      // Fkt is configured - Interfaces and EndPoints are available.
}
else {
      // Something went wrong - can't use Fkt as it is not
      // properly configured
}
// ...
```

In the above code ioreq is a USBIOReq request allocated using USBAllocRequestA(), fkt is the Function the FD controls, and cfgdsc is a pointer to the Configuration Descriptor of the configuration to use in the USB Function.

Get the Interfaces

The Function has now been configured, and the USB stack has prepared itself for handling the Function using that specific configuration.
What you do not yet have, is access to the Interfaces contained in the configuration, or access to the EndPoints of those Interfaces.
When selecting a suitable Function configuration the FD has traversed the descriptors in the configuration. The FD thus also knows which Interfaces and EndPoints are available. So all it has to do is to gain access to the Interfaces, and thru them gain access to the EndPoints required.

The Interfaces of the newly configured Function is found thru the Function based FD specific API call USBGetInterface().
USBGetInterface() will return a UsbInterface for the requested Interface in a Function. It is important to note that it is a claimed Interface which is returned. Therefore the Interface must be declaimed using USBDeclaimInterface() once the FD is done using it.

Let us assume the FD has identified a specific Interface that it wants to use from the Function held in the variable *fkt*. And let us assume that a pointer to the Interface descriptor is held in the *ifcdsc* variable. Then the following code snippet will get that specfic Interface, use it for something, and clean up:

```
// ...
struct UsbInterface     *ifc;

if ( ifc = USBGetInterface( fkt, ifcdsc->id_InterfaceID ) ) {
      // .. let's do something here. This would propably be
      // .. the driver body or something similar.

      puts( "Got the Interface" );
```

```
        // Now we're done, and declaim the Interface
        USBDeclaimInterface( ifc );
}
else {
        puts( "Darn! The Interface does not exist in the Function af-
              terall!" );
}
// ...
```

Get the EndPoints

Regardless of wether the FD is Interface or Function based it will need EndPoints to be able to do any communication over the USB bus. Functions and Interfaces themselves are just means by which to group and classify the EndPoints.

An Interface based FD finds out which EndPoints it needs while examining the Interface descriptor list, as explained earlier in this guide. A Function based FD finds out which EndPoints it needs while examining the entire Configuration descriptor list of the configuration it has chosen for the Function.
In either case, the FD already knows which EndPoints it will be needing, as well as which Interfaces are holding those EndPoints. What the FD needs to do now, is gain acces to those EndPoints. This is done using the API call USBGetEndPoint().

Let us assume the variable *epdsc* points to an EndPoint descriptor for an EndPoint in the Interface *ifc*. Then the following code will get that EndPoint for use:

```
// ...
ep = USBGetEndPoint( NULL, ifc, epdsc->ed_Address );
if ( ep ) {
        puts( "We got the EndPoint!" );
}
else {
        puts( "The EndPoint does not exist!" );
}
// ...
```

One thing which is important to note here is, that there is no cleanup after using the EndPoint. EndPoints are not lockable resources - they live in the belly of their parent USB Interface or Function, and follow the locking of that: If you own the Interface, you also have acces to the EndPoints of the Interface. The same goes for EndPoints in Functions.
One thing to be very carefull about, though, is that changing the configuration of a Function or an Interface will change the use of EndPoints in that Function or Interface. The result of this is that all EndPoints previously obtained by the FD from USBGetEndPoint() are obsoleted at the moment the FD either changes the configuration of a Function, or changes the Alternate Setting of an Interface.
However, if your FD does not change the configuration of its Function or Interface, you need not worry about this.

A common source of problems when trying to find an EndPoint is to belive that you

can just request a predefined EndPoint number. However, normally you'll have to search thru the EndPoint descriptors to find a suitable EndPoint based on EndPoint properties. Once found you get the EndPoint number to use with USBGetEndPoint() from ed_Address of the EndPoint descriptor.
Should you try to get your EndPoints by hardcoded number anyhow, e.g. because a device specification actually calls for this, you must remember that the EndPoint number you supply, is in fact the value which would be found in an ed_Address field of a descriptor for the EndPoint. That is, the EndPoint number and any extended flags which may be set for the EndPoint in ed_Address. For example, if you're in need of the OUT (write) version of EndPoint 1 you would ask for EndPoint 0x01. Should you need the IN (read) version of EndPoint 1 you would instead ask for End-Point 0x81. The difference between the two lies in the direction bit of the ed_Address field.

Just because the USB system sees IN and OUT dataflows as separate EndPoints doesn't mean that EndPoints come in pairs. You may very well have only the IN part of a given EndPoint, or the OUT part.
This is, however, not true for Control EndPoints due to their bidirectional nature. Control EndPoints are always bidirectional, and consist of a single EndPoint capable of both reading and writing.

Talking with your USB Target

Introduction

The previous sections have described how to find, examine and configure your USB Target. All that is just groundwork building up to the real task of an FD, namely talking with the USB Target and get it to do something.
This section will go thru the various ways to communicate with your USB Target.

Basics of transfering data

Allocating IO

The USB stack is designed as an Exec device. What this basically means is, that you use Exec-style IO request passing for USB communication.

Due to the nature of the USB stack it is not sufficient to use *struct IORequest* or even *struct IOStdReq* requests with the USB stack. More information than can be given in these IO request types is needed for IO to find its destination on the USB bus. The USB stack therefore introduces its own *struct USBIOReq* IO request. This request type must be used for any interaction with the USB stack which involves IO. That is, when doing Exec IO operations directly with usbsys.device, or when using USB API calls which maps to IO operations. The usbsys.device API documentation clearly states which API functions performs IO.
One very important aspect to be aware of with *USBIOReq* requests is the fact that

you must never build your own USBIOReq structures in memory.

It is a simple rule, and it must be followed. All *USBIOReq* structures must instead be allocated using the USB stack API call USBAllocRequestA(). This is the only supported way of creating a *USBIOReq* structure.
"Why", you ask? Because your driver will fail miserably otherwise. The USB stack needs internal resource tracking to be able to correctly handle untimely detachment of USB devices. Some of this tracking includes pending IO, so the USB stack needs a private data area to be allocated along with a *USBIOReq*.
When your FD is done with a *USBIOReq* it must be disposed of using USBFreeRequest().

```
struct IORequest  *usbopenreq;
struct USBIOReq   *usbio;

// Allocate IORequest for <usbopenreq>, and open usbsys.device with
   it here
// ...

if ( usbio = USBAllocRequestA( usbopenreq, NULL ) ) {
      // .. good to go ..
      // ...

      USBFreeRequest( usbio );
}
```

```
// ...
// Close usbsys.device and free <usbopenreq>
```

The USBIOReq structure

The USBIOReq structure itself is defined in the include file "usb/system.h". It is based on a standard Exec IOStdReq, but has been extended with USB specific fields. We'll go thru the individual structure fields now, and then get into using them afterwards.

```
struct USBIOReq {
    // *** IOStdReq structure
    struct  Message     io_Message;
    struct  Device      *io_Device;
    struct  Unit        *io_Unit;
    UWORD               io_Command;
    UBYTE               io_Flags;
    BYTE                io_Error;
    ULONG               io_Actual;
    ULONG               io_Length;
    APTR                io_Data;
    ULONG               io_Offset;
    // *** USB Extension
    struct UsbEndPoint  *io_EndPoint;
    APTR                io_SetupData;
    ULONG               io_SetupLength;
};
```

The **io_Message**, **io_Device** and **io_Unit** fields are off limits, and used by Exec and the USB stack for general AmigaOS IO passing.

io_Command is the command to send. At the time of writing the USB stack will respond positively to three commands:

CMD_READ	-	Read data from the target
CMD_WRITE	-	Write data to the target
NSCMD_DEVICEQUERY	-	Query usbsys.device for its capabilities. This you generally won't use as an FD developer. See the AmigaOS NewStyleDevice documentation for further information on this request.

All other commands will be failed with a IOERR_NOCMD error.

io_Flags are standard Exec IO flags. See Exec documentation for these.
One thing to note is that the USB stack will not go into Quick IO. All IO will be enqueued and handled in a synchronous manner.

io_Error holds the error code of the IO request when it is returned. The error code set is the merged set of Exec IOERR_ and the USB stack USBERR_ error codes.

io_Actual returns, as always with Exec IO, the actual number of bytes written to, or read from, the target.

io_Length is the byte size of the data bufferpointed to by io_Data. If this field is non-zero, then the USB stack can use the io_Data buffer for transfers. If zero, then the io_Data buffer pointer is never followed.

io_Data is a pointer to the data buffer for the actual transfer. Only if io_Length is non-zero will this pointer be followed. I.e. data read from, or written to, the buffer.

io_Offset is private to the USB stack and must be left alone.

io_EndPoint is the first USB specific IO field. Here you store the UsbEndPoint reference of the EndPoint you whish to communicate with.

io_SetupData is a field used sololy when communicating with Control type EndPoints. It points to a data block to use for a Control transfer Setup stage. Keep this to NULL if you're not talking with a Control EndPoint.

io_SetupLength defines the byte size of the buffer pointed to by io_SetupData. If there is no io_SetupData buffer for any reason, this field must be zero.

Talking with Control EndPoints

Background information on Control EndPoints

Before going into details on communicating with Control EndPoints, some background information on this special kind of EndPoint may be in order.
Control EndPoints plays a special role in the USB framework. They handle all (or at least most of) the sideband signalling and housekeeping of the USB bus.
Therefore Control EndPoints has a more complex nature when it comes to the actual transfers performed on the USB bus for a single IO operation.
A transaction with a Control EndPoint is broken into three steps:

1. Setup stage
2. Data stage
3. Acknowledge stage

Of these three steps, only the first two are of interest for FD developers, but the Acknowledge stage is also mentioned here for reasons of completeness.

In the **Setup stage** a Setup packet is sent to the Control EndPoint. This packet, represented by the struct USBBusSetupData defined in the include file "usb/usb.h", contains information about the transfer. This include information such as the command to perform at the target, the data direction of the Data stage (read or write data), command arguments, number of bytes to transfer in the Data stage etc.

In the **Data stage** data is transfered to or from the Control EndPoint depending on the content of the Setup data. The Data stage will transfer as much data as was defined in the Setup data.

As the last stage the **Acknowledge stage** verifies that the transaction was success-

fully carried out. You, as a Function Driver developer will not need to think about this stage, however, as it is hidden by the USB stack. But now you know that it exist.

Setting up your USBIOReq

The special transfer style used on the USB bus for a Control EndPoint, as outlined above, means that IO concerning a Control EndPoint needs two separate data buffers to be passed to the USB stack: One for the Setup stage data, and one for the Data stage data. As you might have guessed by now, the io_SetupData and io_SetupLength fields of the USBIOReq structure are there for exactly this reason: To define a Setup data buffer for IO concerning Control EndPoints.

You therefore need to take three steps to talk with a Control EndPoint:

1. Allocate a struct USBBusSetupData buffer in a publicly accessible memory area and fill in the fields according to the command you're sending.
2. Set up the USBIOReq fields.
3. Pass on the request to the USB stack.

The actual values to place in the USBBusSetupData fields will not be explained in this document. The USBBusSetupData structure has a field-to-field mapping with the Setup Data structure defined in the USB specification, where you'll also find a definition of all the standard commands.
One thing which should be mentioned about the USBBusSetupData here, is that

the USBBusSetupData is a USB bus structure - it is LittleEndian by nature.

When you read or write values other than byte-size, you must remember to perform endianness conversion of the values. The include file "usb/usb.h" defines a macro, LE_WORD(), which will perform endianness convertion to/from LittleEndian of a 16 bit data word.

Once you've filled the USBBusSetupData it's time to roll the IO request. Here you'll set the io_Command to CMD_READ or CMD_WRITE, depending on the intended direction of the Data stage of the Control transfer.
You then set the io_SetupData and io_SetupLength to that of your USBBusSetupData structure.
Now is a good time to define the data buffer for the Data stage of the transfer, which is done with io_Data and io_Length. If there is no Data stage involved with the Control transfer, then you're free to leave these at NULL and zero respectively.
The only thing missing now, is the EndPoint to talk to. That you specify in the io_EndPoint field of the USBIOReq structure.

Doing the IO

With the steps explained above performed, everything is now ready for the USB stack to start handling the transfer. The IO is initiated by passing the USBIOReq on to the USB stack using Exec's SendIO() or DoIO() functions.

Taking the shortcut

As you might have discovered, Control transfers are not the simplest form of data transfer on the USB bus. It involves some setting up, and a preallocated USBBusSetupData structure.
To make things a bit easier, the USB stack has an API call for handling Control transfers in a more friendly way. That API call is USBEPControlXferA().
The upside of this call is that you can specify the USBBusSetupData as function arguments, rather than rolling your own USBBusSetupData. USBEPControlXferA() will internally build the USBBusSetupData and handle endianness conversion for you where it is required.
The downside of USBEPControlXferA() is that it is a synchronous call. It will not return until the Control transfer has finished. If you need asynchronous Control transfers, then you're on your own.

An example

Two methods for Control transfers have been described above. To give a better view of these methods, and their differences, an example is in place.
What we want to achieve in the example code, is to read the status word of a USB Function. The status word is a 16 bit bitmask holding the Function status flags.
First we'll read the status word by rolling the IO request by hand. Then we'll do the same thing once more, but this time make a shortcut thru the USB stack API USBEPControlXferA() call.
Both example code snippets below assumes that usbsys.device has already been opened using an IORequest named "usbopenreq", and that USBSysBase has been set to the value of usbopenreq->io_Device. Furthermore the variable "fkt" holds the claimed USB Function to get the status word from.

First the manualy build Control transfer version:

```
// ...
struct UsbEndPoint *dcp; // Default Control Pipe
struct USBIOReq    *usbio;
UBYTE              *buffer; // Buffer for the Function status

// ** Get Control EP to talk to
dcp = USBGetEndPoint( fkt, NULL, 0 );

// ** Allocate IO and buffer
usbio = USBAllocRequestA( usbopenreq, NULL );
buffer = AllocMem( 2, MEMF_PUBLIC );

if ( dcp && usbio && buffer ) {
      struct USBBusSetupData  *setupdata;

      if ( setupdata = AllocMem( sizeof( struct USBBusSetupData ),
                                 MEMF_PUBLIC ) ) {
            // ** Set up the Setup stage data
            setupdata->sd_RequestType = USBSDT_DIR_DEVTOHOST |
                                        USBSDT_TYP_STANDARD |
                                        USBSDT_REC_DEVICE;
            setupdata->sd_Request = USBREQC_GET_STATUS;
```

```
            setupdata->sd_Value = LE_WORD(0); // (LE_WORD is absurd
                            for zero value, but you get the idea...)
            setupdata->sd_Index = LE_WORD(0);
            setupdata->sd_Length = LE_WORD(2); // This must match us-
                                                  bio->io_Length

            // ** Setup the IO request
            usbio->io_Command = CMD_READ;
            usbio->io_SetupData = setupdata;
            usbio->io_SetupLength = sizeof( struct USBBusSetupData );
            usbio->io_Data = buffer;
            usbio->io_Length = 2;
            usbio->io_EndPoint = dcp;

            // ** Perform the Control transfer
            DoIO( (struct IORequest *) usbio );

            if ( usbio->io_Error == USBERR_NOERROR ) {
                  // ... use the result for something
            }

            FreeMem( setupdate, sizeof( struct USBBusSetupData ) );
      }
}

if ( buffer ) FreeMem( buffer, 2 );
if ( usbio ) USBFreeRequest( usbio );
// ...
```

And now for the short version of the exact same functionality:

```
// ...
struct UsbEndPoint *dcp; // Default Control Pipe
struct USBIOReq    *usbio;
UBYTE              *buffer; // Buffer for the Function status

// ** Get Control EP to talk to
dcp = USBGetEndPoint( fkt, NULL, 0 );

// ** Allocate IO and buffer
usbio = USBAllocRequestA( usbopenreq, NULL );
buffer = AllocMem( 2, MEMF_PUBLIC );

if ( dcp && usbio && buffer ) {
      LONG err;

      err = USBEPControlXferA( usbio, dcp, USBREQC_GET_STATUS,
                               USBSDT_DIR_DEVTOHOST |
                               USBSDT_TYP_STANDARD |
                               USBSDT_REC_DEVICE, 0, 0, 2, NULL );
      if ( err == USBERR_NOERROR ) {
            // ... use the result for something
      }
}

if ( buffer ) FreeMem( buffer, 2 );
if ( usbio ) USBFreeRequest( usbio );
```

```
// ...
```

It is quite easy to spot the difference in code complexity between the two methods. Most of the time it suffices to use USBEPControlXferA(), but on rare occasions, such as asynchronous Control xfers, you will need to do it all by hand.

Some things you're not allowed to do

As explained earlier, Control EndPoints handle the housekeeping of USB entities attached to the USB bus. If you read up on the standard Control commands, you'll find that configuration of USB Functions and selection of Alternate Settings for Interfaces are also done by Control transfers.
Due to the structural nature of USB Function and Interface configuration, the USB stack needs to be in full control of when and why a configuration is changed. Only in knowing that, can the USB stack create the required internal structures that keeps it all working. For this specific reason some of the standard Control transfer commands are disallowed by the USB stack:

- SET_CONFIGURATION
- SET_INTERFACE

Don't use them - it's as simple as that. At best you'll have the USB stack complaining about your FD, at worst somethings goes terribly wrong.
Instead of using the forbidden lowlevel Control commands, the USB stack has API calls which maps to these commands. The replacement API calls are USBFktSetConfigurationA() for SET_CONFIGURATION and USBIntSetAltSettingA() for SET_INTERFACE.

Talking with non-Control EndPoints

In the previous section we've been going thru communicating with Control EndPoints. In this section we'll take a look at communicating with non-Control EndPoints. On the IO request level all non-Control EndPoints are used alike. All these EndPoint types are therefore covered as one.

Setting up your USBIOReq

As non-Control EndPoints do not involve any Setup stage in their transactions, these EndPoints are much easier to communicate with on an IO request level. In other words, there is no USBBusSetupData structure to prepare, and as a result io_SetupData and io_SetupLength of the USBIOReq structure are simply kept at NULL and zero respecetively. The rest of the USBIOReq structure is used just as with Control EndPoints:

1. Set io_Command to CMD_READ to read data in from the EndPoint, or CMD_WRITE to write data to the EndPoint.
2. Set io_SetupData to NULL and io_SetupLength to zero as there is no Setup stage for non-Control EndPoint0s.
3. Set io_Data to point to your data buffer. Remember the buffer must be in publicly available memory.

4. Set io_Length to the number of data bytes to read/write.
5. Set io_EndPoint to the EndPoint with which to communicate.

Sometimes it is required to do a zero length write to an EndPoint. Such a transaction is created by having a zero io_Length and a non-NULL io_Data. In this case io_Data does not have to point to any specific memory area - it will never be referenced. Any value is safe, as long as io_Length is zero.

Doing the IO

With the USBIOReq prepared as described above, everything is now ready for the USB stack to start handling the transfer. The IO is initiated by passing the USBIOReq to the USB stack using Exec's SendIO() or DoIO() functions.

An example

The following example will write a buffer to non-Control EndPoint 2 of a USB Interface, followed by reading some data from non-Control EndPoint 3 of the same Interface.
It is assumed that usbsys.device has already been opened using an IORequest named "usbopenreq", and that USBSysBase has been set to the value of usbopenreq->io_Device. Furthermore the variable "ifc" holds the claimed USB Interface hosting the EndPoint we're going to communicate with.

```
struct UsbEndPoint *ep;
struct USBIOReq *usbio;
UBYTE *buffer;

// ...

// ** Allocate IO and buffer
usbio = USBAllocRequestA( usbopenreq, NULL );
buffer = AllocMem( 10, MEMF_PUBLIC );

if ( usbio && buffer ) {

      // ** Get EndPoint to write to
      if ( ep = USBGetEndPoint( NULL, ifc, 2 ) ) {

            // ** Setup and perform data write
            usbio->io_Command = CMD_WRITE;
            usbio->io_SetupData = NULL;
            usbio->io_SetupLength = 0;
            usbio->io_Data = buffer;
            usbio->io_Length = 10;
            usbio->io_EndPoint = ep;

            SendIO( usbio );
            // .. perhaps you'd like to do something until IO is done
            WaitIO( usbio );

            if ( usbio->io_Error != USBERR_NOERROR ) {
                  // Uh, oh, something went wrong!
            }
```

```
        }

        // ** Get EndPoint to read from
        if ( ep = USBGetEndPoint( NULL, ifc, 3 ) ) {

                // ** Setup and perform data read
                usbio->io_Command = CMD_READ;
                usbio->io_SetupData = NULL;
                usbio->io_SetupLength = 0;
                usbio->io_Data = buffer;
                usbio->io_Length = 2;
                usbio->io_EndPoint = ep;

                SendIO( usbio );
                // .. perhaps you'd like to do something until IO is done
                WaitIO( usbio );

                if ( usbio->io_Error == USBERR_NOERROR ) {
                        // We got the data - now use it for something...
                }
        }
}

if ( buffer ) FreeMem( buffer, 10 );
if ( usbio ) USBFreeRequest( usbio );
// ...
```

IO errors

Every IO request that is returned from the USB stack it carries with it an error code indicating the status of the request. This chapter is devoted to these error codes, and will try to explain the meaning of them.

Apart from the standard Exec IO error codes the USB stack at the time of writing defines the following set of error codes:

Structural errors
USBERR_DETACHED
USBERR_NOBANDWIDTH
USBERR_NOMEM
USBERR_NOSIGBIT
USBERR_BADARGS
USBERR_NOENDPOINT
USBERR_UNSUPPORTED
USBERR_ISPRESENT
USBERR_ATTACHFAIL

Transaction errors
USBERR_STALL
USBERR_XFERFAIL
USBERR_NAK
USBERR_TIMEOUT
USBERR_BUFFEROVERFLOW

The error codes are defined in the include file "usb/system.h". Apart from the above error codes, the USBERR_NOERROR error code also exist, which is just a naming of the non-error value zero.
In the following sections each error code will be explained in turn.

USBERR_DETACHED

You will see this error code if the target of the IO request has been detached from the USB bus before or during the processing of the request.
This is most likely to happen around the time when a USB Function has been detached from the USB bus. Any pending IO will then be returned with USBERR_DETACHED, and any subsequently initiated IO will be sent back with this error code as well.

Due to the multitasking nature of the USB stack and its Host Controller Drivers, which handles the lowlevel USB bus drivers, USBERR_TIMEOUT or USBERR_XFERFAIL errors may be returned in place of USBERR_DETACHED for IO requests until the USB system is fully aware that the target has actually been detached.

USBERR_NOBANDWIDTH

This error indicates that there is not sufficient bandwidth available on the USB bus to support the request.
This error is not directly tied to IO operations, but rather to situations where the overall USB configuration changes in a way that influences on bandwidth usage. The mostprominent example of this is when a new USB Function is being attached. If the USB stack determines that there is not enough bandwidth available to sustain this new Function, it will deny its addition. The same will happen when reconfiguring a USB Function or USB Interface if the new configuration will claim more bandwidth than is available on that particular USB bus.

USBERR_NOMEM

If at any time the USB stack fails to allocate memory for the handling of a request, the request will be failed with USBERR_NOMEM.

USBERR_NOSIGBIT

Not directly related to IO this error is used if the USB stack is unable to allocate a signal bit for internal signalling.
Generally you should make sure at least one free signal bit is available when operating with the USB stack APIs to be on the safe side.

USBERR_BADARGS

If you pass invalid arguments to the USB stack, be it in IO requests or in API calls returning a USB error code, you will get this error - if the USB stack is able to detect the error, that is.
As an FD developer this error typically indicates something very bad is happening

within your FD. You should check and recheck your code until you find the source of the problem. A properly implemented FD will *never* see this error.
Typically a log entry with further information is added to the USB system log in "T:usb.log".

USBERR_NOENDPOINT

This error is returned if the USB stack for some reason is unable to find a referenced EndPoint during IO or an API call. Either the EndPoint has been detached obsoleting your EndPoint reference, or something is wrong further down the foodchain of the USB stack. In any case you won't in the touch with the referenced EndPoint anymore.

USBERR_UNSUPPORTED

This error is used for indicating that a requested operation is not supported. The reasons for not supporting a specific request can be many:

- The hardware does not support it.
- The hardware driver does not support it.
- The USB stack does not support it.
- .. add your own reasons here ..

What should be mentioned here is that only lowlevel lack of support for a request will result in this error. Any unsupported feature on a USB Function/Interface/EndPoint or USB class level is identified and reported within the contents of IO transactions with an EndPoint in the unsupporting entity.

USBERR_ISPRESENT

This is not an error as such, rather a special return value used when registering an AutoStarting FD with usbresource.library.
Seeing this return value simply indicates that the FD is already registered with the USB stack.

The error code may come to use in other places where it can come in handy to know if something already exist. Currently, though, it is only used with usbresource.library.

USBERR_ATTACHFAIL

Only hub driver FDs should ever come to experience this error.
The error indicates that it was impossible to handle attachment of a newly added USB entity on the USB bus. The error relates to the lowlevel adding and address assignment going on in the Host Controller Driver handling the USB bus at a hardware level before the USB stack takes over and makes the USB Function publicly available.
This error does not relate to Function Driver IO operations whatsoever.

USBERR_STALL

This error indicates that the target EndPoint of the IO request stalled while processing the IO. Stalling is a way for EndPoints to indicate being busy or being in failure.

The actual meaning of the stall is dependent on the specification of the device or device class the FD is dealing with.

USBERR_XFERFAIL

A transfer failure on the USB bus will result in USBERR_XFERFAIL being returned for the IO request being processed at the time of failure.
The reason for a transfer failure is hard to predict, but may very well come from temporary conditions such babble or noise on the USB bus.

USBERR_NAK

If the target EndPoint of an IO operation gives a negative acknowledge on the USB bus as response to a transaction, the IO request is retired with a USBERR_NAK error. Depending on the device or class specification this might not be an error, rather a signal to the FD that the EndPoint is busy processing an asyncronous request, has no data to deliver, can’t process any more data at the moment, or something entirely different.

USBERR_TIMEOUT

This error occurs if there was no response on the USB bus from the target EndPoint, so the bus controller had to retire the request.
This will typically happen just when a USB Function is physically detached from the USB bus. The USB stack is not yet aware that the Function is detached, so it keeps on sending IO for EndPoints in the Function. Those IO requests will time out.
If timeouts occur for a prolonged duration it could point to a dead USB Function.

USBERR_BUFFEROVERFLOW

The buffer overflow error indicates that the io_Data buffer of an IO request has been filled completely, and that excess data bytes have been discarded to avoid overwriting innocent memory.
This error will occur if a request for data at the target EndPoint results in more data than expected. Whether this is an error condition depends upon the specification for the target device or the device class.