Table Of Contents

Document information

History

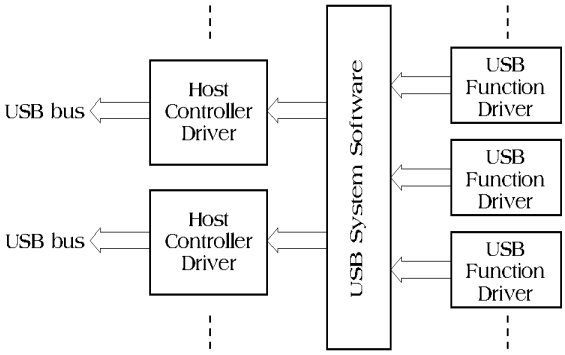
V0.1 - Initial draft. Based on USB System API V0.2 and USB HCD API V0.1.
V0.2 - Removed references to now obsoleted USBClaimFunction(fkt,NULL) call for locking USB Functions.
Added section on filesystem usage.

Disclaimer

Overview

Introduction

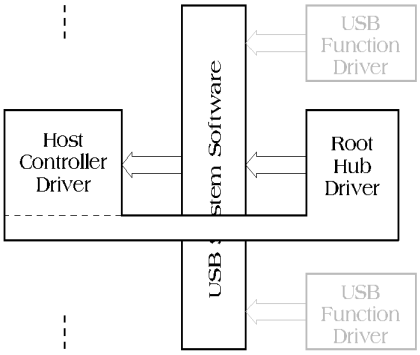
The USB bus is an expansion bus with only one bus master - the Host Controller. The Host Controller is responsible for all transactions on the USB bus. On writes it passes data packets to a Function on the USB bus. On reads it sends requests to a Function on the USB bus, which then returns the requested data. That is, all data transmissions are initiated by the Host Controller.
The same architecture has been transfered to the USB System Software (USS). Here the application (the software using USB - also referred to as a USB Function Driver) is responsible for initiating transactions, sending read and write requests to a USB Function through the USS. The USS then redirects the requests to the Host Controller expansion card to which the USB Function is attached, by forwarding requests to the Host Controller Driver (HCD) controlling the expansion card.
Using this architecture the USS has no need to communicate with the USB Function Drivers (except for return values in the requests). Also, the HCD has no need to communicate with the USS (with one exception for the root hub, covered later). Therefore, to get the different parts of the system talking together only downstream APIs are needed - that is, APIs for calls from a higher level software part to a lower level software part.



The Host Controller Driver

The Host Controller Driver (HCD) is responsible for controlling a USB expansion card inserted into the Amiga. It presents a standard API to the USS to allow Host Controllers to be appended to the USS without modifying the USS.
The HCD is an Exec device which is opened by the USS when the USS is first started. The USS places data send and retrieve requests using an extended device request. All other operations take place through library entry points in the device.
Although the HCD is responsible for controlling a USB Host Controller, it also has one other important role: It controls the root hub of that Host Controller. The root hub is the USB hub positioned at the end of the Host Controller, and which presents the top set of USB expansion connectors.
The HCD is responsible for this hub as it is often different in implementation than the rest of the USB hubs - it possibly being based on memory mapped registers - mak-

ing it impossible to control by a standard USB hub driver.
The requirement of a root hub driver makes the HCD both a driver for the Host Controller and a Function driver. For normal data transmission the HCD has no direct communication back to the USS, only receiving and replying to requests. But for operation as a root hub driver the HCD actually has to call functions in the USS to signal Function insertion and removal on the root hub ports.



The Function Driver

The Function Driver is responsible for handling a USB Function on the USB bus, using API functions in the USS.
There are two main types of Function Drivers defined for the USS: autostart drivers and selfstart drivers. An autostart driver is a driver which is linked to the USS and connected to a USB Function class and subclass. Upon insertion of a USB Function on the USB bus, the USS searches for an autostart driver capable of handling the USB Function. If one is found, the driver is started and connected to the USB Function. An autostart driver is an Exec library with the driver code as a library function.
The selfstart driver is, as the name suggests, a driver not connected with the USS. The driver may be part of an application program, and connects itself into the USS at will, by connecting to a USB Function through the USS API. As a selfstart driver is not directly coupled to the USS, no code format is enforced on selfstart drivers.

Other than the startup and method of linking into the USS, both driver types have the same responsibilities, and do not differ in operation.
Once running, the USB Function Driver is responsible for configuring its USB Function through the USS, after which the driver does its job - whatever it is - by sending and receiving data through the USS functions.

A Function Driver may be expunged with time. If the USB Function the driver is responsible for is removed from the USB bus the driver will receive notification. If this occurs the driver performs cleanup and terminates.
One special instance of a Function Driver is the USB hub driver. A hub driver is responsible for adding and removing USB Functions from the USS when they are attached/detached at the hub ports. This is done using specialized USS function calls.

Abstraction

One of the goals of the USS is to form an abstraction layer between the Function Drivers and the actual USB bus. A design goal of this USS implementation has also been to hide the data structures of each layer from the other layers. As can be seen from the USS APIs only the IoRequest structures used for data passing are shared. All other information are held in private structures referenced by a key of some sort (most likely pointers to private structures - but that is implementation dependant). This way of doing things keeps the high- and lowlevels of the USB implementation logically detached from eachother, making alterations to the implementation of the various layers easier.

Implementation

USB System

The USB System is to be implemented as an Exec device, as seen from the Function drivers point of view. However, as all autostarting Funtion drivers will be started by the USB System, someone or something must start the USB System first.
The actual implementation of this is left to the USB System implementor. It could be a stub program opening the USB System device to place it into memory. It could also be a USB System program being run, which manually sets up an Exec device and links it into the system (like e.g.Amiga TCP/IP stacks do it).

However, once running, the USB System acts like an Exec device. Function drivers will communicate to the USB System using (specialized) IoRequests and library entry points in the device.

Autostarting Function Drivers

Autostarting Function Drivers are to be implemented as Exec libraries. The library format forms an easily extendable and system standard implementation method for the Function Driver API. Also, the library system also supports reentrance, allowing the same Function Driver code to be used for several USB Functions, simply by launcing a new task and call the Function Driver entry point in the driver library.
By giving arguments to the driver entry point the USS can transfer a reference to the USB Function the Function Driver is to handle.
While running, the driver may get a detachment message from the USS (when the Function being handled by the driver has been detached from the USB bus). The driver is then supposed to perform cleanup and finally return out of the library function called to start the driver. This will, in turn, allow the USS to perform cleanup and, if necessary, close the Function Driver library.

Selfstarting Function Drivers

Selfstarting Function Drivers are, as mentioned earlier, not restrained to any specific implementation. The USS will never try to call functionality in a selfstarting driver - in fact the USS don’t have any link to the selfstarting driver at all.
A selfstarting driver simply opens the USS device and uses functions herein to connect to USB Functions.

Host Controller Drivers

A Host Controller Driver is implemented as an Exec device. A HCD is opened by the USS when the USS is initializing itself after being started.
After being opened the USS will use (specialized) IoRequests for data passing, and library entry points in the device for control operations.
The library entry points may be called from any task, as the USS may (or may not) be running in the task of a Function Driver. A Host Controller Driver is itself responsible for access arbitration if needed. This allow the HCD to optimize its code internally to allow multitasking where possible, instead of the USS blindly single-threading access to a HCD.

API usage

The following sections describe the usage of the APIs designed for the USB System Software. The usage is described seen from different points of view, depending on which API is being described.

System Software API

The System Software API is the API presented by the USB System device. The following describes the usage as seen from a Function Driver.

Autostarting Function Driver

When an autostarting Function Driver is launched (by calling its driver entry point), a reference to a USS USB Function is given as argument. The referred Function has already been locked by the USS to ensure that it is not removed from under the feet of the driver.
Using the Function reference the driver makes any necessary examination of the given USB Function (for other requirements than those already specified in the Function descriptor file). This can be done by reading descriptors from the USB Function using the USBGetDescriptor() function. If the given Function cannot be handled by the driver, the driver exits with an error code indicating the reason.
Otherwise the driver now claims ownership of the USB Function by calling USBClaimFunction(), linking the Function with a message port which will be used for sending messages and returning data requests to the driver from the USS.
Now the driver may read configuration descriptors using USBGetDescriptor() for determining a suitable configuration for the USB Function. Finally a configuration is chosen by calling USBSetConfiguration().
A onfiguration having been chosen, it is now possible to gain access to the various endpoints described by the configuration. A reference to an endpoint is obtained by using USBGetEndPoint().
Once a reference is held on an endpoint, data can be sent to, or received from the endpoint by sending (specialized) IO requests to the USB System device.

The autostarted Function driver keeps on running until it receives a detachment message, indicating that the Function it is handling has been detached from the USB bus. The Function driver must now abort any outstanding IO requests it has pending, and shut itself down. Shutting down includes calling USBDeclaimFunction(), unregistering as user of the detached USB Function.
Finally the Function driver exits out of the driver function, returning control of final termination to the USS.

Selfstarting Function Driver

A selfstarting Function Driver is not launched by the USS. The origin of the driver is not of imortance to the USS. Instead the Function Driver opens the USB System device to gain access to a USB Function.
Having opened the USB System the driver starts looking for the USB Function it wishes to handle. This is done by successive calls to USBFindFunction(), each call returning a reference to a USB Function matching the search criteria.

USBFindFunction() will return a locked USB Function reference, ensuring that the referenced Function is not expunged while in use. If doing successive USBFind-Function() calls the first returned reference must be unlocked after having obtained the second reference, by calling USBDeclaimFunction(fkt, NULL). Unlocking must not be performed until after obtaining the next Function reference, as USBFindFunc-tion() uses the first Function reference to return the next. Unlocking before calling USBFindFunction() could potentially result in the reference given to USBFindFunc-tion() being expunged by the time the reference is used inside the function call.
Having found the USB Function to use, the driver claims it by calling USBClaimFunc-tion(), binding an (already initialized) message port to the Function. After having claimed the Function the temporary lock set on the Function reference by USBFind-Function() must be cleared. This is, again, done by calling USBDeclaimFunction(fkt, NULL).
The proper way of iterating through possible USB Functions with USBFindFunction() is therefore:

```
prevfkt = NULL;
usable = False;
do {
      fkt = USBFindFunction(class, subclass, prevfkt); // Get Func-
                                                          tion reference
      if ( prevfkt ) {
            USBUnlockFunction( prevfkt ); // Unlock previously exam-
                                                    ined Function
      }

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

      prevfkt = fkt;
} while ( fkt );

if ( fkt ) {
      DoYourStuffWithTheFunction(); // Configure and handle the Func-
                                       tion (driver body)
      USBDeclaimFunction( fkt, fktkey ); // 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 variables class and subclass are set to values of the needed USB Function type.

Having claimed a USB Function the driver may now read configuration descriptors using USBGetDescriptor() for determining a suitable configuration for the USB Function. Finally a configuration is chosen by calling USBSetConfiguration().
When a configuration has been chosen it is possible to gain access to the various endpoints described by the configuration. A reference to an endpoint is obtained by using USBGetEndPoint().
Once a reference is held on an endpoint, data can be sent to, or received from the endpoint by sending (specialized) IO requests to the USB System device.

The selfstarting Function driver now utilizes the claimed Function until it is either done with its job, or it receives a message informing that the USB Function has been detached from the USB bus. If the Function has been detached the driver must now abort any outstanding IO requests it has pending, and release the USB Function reference it is holding. If the driver is done with its job it must simply release the USB Function as part of its clean up.
For both termination causes releasing the USB Function is done by calling USBDeclaimFunction(). This is also outlined in the example code shown above.

Special case: Hub driver

As mentioned earlier in this document a Hub on the USB bus is the place where USB devices are attached and detached. The Hub therefore plays a special role in detacting attachment and detachment of Functions on the USB bus.
For this reason hub drivers (both HCD root hub and other hubs) must be able to add and remove Functions from the USS.

Function attachment and detachment

When a hub driver detects that a new device has been attached at its hub, the driver must add the Function to the USS. This is done by calling USBAddFunction(). USBAddFunction() returns a reference to the newly added USB Function. The hub driver should take note of this reference.
If, or when, a Function is detached from the driver’s hub, the driver must remove the detached Function from the USS. This is done by calling USBRemFunction(), giving the reference returned by USBAddFunction() at attachment time as argument.

Hub attachment

When a hub Function is attached to the USB the hub into which it is inserted will add the hub to the USS. The USS will then launch a hub driver for that hub Function.
When a new hub driver is launched it is, apart from normal Function Driver setup of its Function, also responsible for attaching any devices which may be attached to the hub ports at hub attachment time. This is done in the normal way, calling USBAddFunction(), for all attached Functions.
In this way an entire tree of USB devices can be attached in one go by launching the hub driver for the root of the tree.

Hub detachment

When a hub Function is detached from the USB bus the hub driver must be shut down. Together with the hub driver shutting down, all USB Functions which was attached to the hub at detachment time must be removed from the USS, as they have also been detached with the hub.
The hub driver responsible for the detached hub must therefore follow normal Function detachment procedure for all Functions which was attached to it. That is, call USBRemFunction() for all Functions attached to its ports.
This ensures that, by signalling Function removal to a hub driver, all USB Functions beneath the hub in the USB topology will also be removed from the USS.

Host Controller Driver API

The Host Controller Driver API is the API presented by a Host Controller Driver to the USB System Software. The following describes the API usage as seen from the USS.

HCD Startup

When the USS is booting it opens all HCD devices it can find. If the HCD is unable to locate the Host Controller (HC) hardware it is a driver for, it should refuse to be opened.
When opened, the USS will want to get a reference to the root hub of the HC to get a top entry point for adding Functions. This is done by the USS calling USBHCAddFunction() with a NULL hub argument. The HCD sees the NULL reference and returns its root hub reference. The usfunction argument to USBHCAddFunction() is a reference to the USS representation of the hub. This must be used by the HCD root hub driver when adding USB Functions to the USS.

HCD Shutdown

The USS should be able to shut down. Generally this will not be used, but it should be possible to safely shut down USB operations. When the USS shuts down it will do clean up, and then close the HCD devices it has open.
Part of the cleanup is to remove all Functions from the USB bus (logically - not physically). This is done by calling USBHCRemFunction() for all attached USB Functions. Calling will be done starting from the bottom of the USB Function tree and up, ensuring that the USB topology is always valid.
As the last cleanup step the USS will call USBHCRemFunction() on the HCD root hub.

Function attachment

When a USB Function is attached to the Function tree of a HC, the HCD for the HC is informed. This is done to allow the HCD to setup internal structures to be able to handle the new Function. The HCD is, however, also responsible for enumerating the new USB Function on the USB bus. Enumeration may take place either at Function add time, or when the USS adds the Default Control Pipe EndPoint.
When a Function is being added to the USS, the USS will call the HCD responsible

for the USB bus the Function has been attached to. The HCD will then do its internal setup and return a reference to its private Function data. The USS does this by calling USBHCAddFunction() in the HCD. Together with this call the HCD is given its private hub reference to the hub in which the Function is attached, and a reference to the new Function in the USS.
The newly added Function has, at this time, no EndPoints attached to it what-so-ever. The USS will add EndPoints as found necessary at a later time. Even the Default Control Pipe is added later by the USS.

Function detachment

When a USB Function is removed from the Function tree of a HC, the HCD for the HC is informed. This is done to allow the HCD to clean up internal structures after Function removal.
A Function is removed from the USS by a call to USBRemFunction(). This call will be forwarded by the USS to the HCD responsible for the USB bus on which the Function being removed is placed. The HCD can then do its internal cleanup. The USS forwards the removal by calling USBHCRemFunction() in the HCD.
Before removing the Function from the HCD the USS will close all open EndPoints by calling USBHCRemEndPoint() for each open EndPoint in the Function.

EndPoint creation

To be able to communicate with a USB Function EndPoints must be created. When a Function has been added to a HCD the Function has no EndPoints.
After adding a Function the USS will open the Default Control Pipe EndPoint to the Function. Using this the Function driver can select a configuration, which will make the USS open the EndPoints needed for the selected configuration.
To the HCD the USS is opening a new EndPoint by a call to USBHCAddEndPoint(). The HCD must then examine the requirements of the EndPoint being created and either support it, or deny it. If the EndPoint is supported the HCD must setup its internal structures to be able to handle the EndPoint, and return a reference to it. The reference will be used by the USS on subsequent calls related to the EndPoint.
An EndPoint cannot be supported if its bandwidth requirements exceed the available USB bus bandwidth. In such a case the HCD refuses the EndPoint by returning a NULL EndPoint reference.

EndPoint removal

When the USS is certain that an EndPoint is not going to be used anymore, it will close the EndPoint. This is done by the USS calling USBHCRemEndPoint(), giving the EndPoint reference returned by USBHCAddEndPoint() when the EndPoint was initially added to the HCD.

Filesystem usage

In this section the layout of filesystem usage in the USB System Software is explained.

Filetree

All files related to the USS is positioned what has been found the most suitable place for USB drivers (HCD, FD and USS alike). The file position is a drawer named Usb under the DEVS: assign. That is, all USB related driver software (except self-starting drivers) are placed in "DEVS:Usb/".
In this drawer a filesystem tree exist, holding all the USB related files in an ordered manner. This is done by the following drawers:

DEVS:Usb/fdclasses	All descriptors for AutoStarting Function Drivers are to be placed here
DEVS:Usb/fd	All AutoStarting Function Driver libraries are to be placed here
DEVS:Usb/hcd	All Host Controller Driver devices are to be placed here

The position of the USS executable is not covered by this specification.

Filenaming

To allow easy detection of filetypes from directory contents it has been chosen to set a filenaming standard for USS driver files. This also keeps the driver devices and libraries separate from other system and 3rd party libraries by name.
The following naming conventions must be followed:

Function Driver Descriptors	*.usbfdcls
Function Drivers	*.usbfd
Host Controller Drivers	*.usbhcd

Function Driver descriptors

Each Function Driver must have a descriptor file, informing the USS which USB devices the driver can handle.
A Function Driver descriptor file is a text file holding keyword-and-value pairs describing the capabilities of the Function Driver. These keywords are outlined below.

FUNCTIONDRIVER

A Function Driver descriptor file must start with this keyword on the first line of the file. If this keyword is not present the USS will not interpret the file as a Function Driver descriptor.

CLASS

The value of this keyword is a hex value specifying the USB class code supported by the Function Driver. If this is not specified the USS interprets the Function Drivers as capable of handling all USB classes.

SUBCLASS

The value of this keyword is a hex value specifying the USB subclass code sup-

ported by the Function Driver. If this is not specified the USS interprets the Function Drivers as capable of handling all USB subclasses for the USB class specified by the CLASS keyword.

PRI

The value of this keyword specifies the driver priority. If more drivers are able to handle a USB Function the driver with the highest priority will get the Function.
The priority is a signed decimal number between +127 and -127.

DRIVER

The value of this keyword is the filename of the Function Driver library the descriptor file describes. Generally this should be supplied without path, making the USS look in the "DEVS:Usb/fd" drawer for the driver. If a path is specified in the filename the Function Driver library will be opened from that place.
The full filename must be given for the Function Driver. That is, including the standard ".usbfd" file extension.