Mac OS X Cypress MSC Driver Interrupt Pipe Support Notes

Purpose

The 1.2 Cypress MSC Driver for Mac OS X, version 1.2 and newer, provides Interrupt Pipe support to customers interested in implementing this feature in the ISD-300A1 USB 2.0 to ATA/ATAPI bridge solution. The driver setups up the USB Interrupt pipe and provides a mechanism to communicate with User Space applications when USB devices send interrupt pipe data.

This document describes the implementation of Interrupt pipe support in the Mac OS X driver and describes what the application developer needs to do in order to be receive the interrupt notification and the interrupt data.

Driver Implementation

All Mac OS X Kernel Extensions (KEXTs) contain a property list that is used not only for driver matching but also provides a location to enter other KEXT properties or characteristics. A property list consists of a dictionary (key/value pairs) containing other sub-dictionaries written in XML format. Inside the property list is a dictionary called IOKitPersonalities. The IOKitPersonalities dictionary contains sub-dictionaries that are used by the system during driver matching. And example of a property list is shown in Listing 1.

```
<!DOCTYPE plist SYSTEM \"file://localhost/System/Library/DTDs/PropertyList.dtd\">
<plist version=\"0.9\">
<dict>
  <key>CFBundleDevelopmentRegion</key>
  <string>English</string>
  <key>CFBundleExecutable</key>
  <string>com_cy_driver_USB_Device</string>
  <key>CFBundleIconFile</key>
  <string></string>
  <key>CFBundleIdentifier</key>
  <string>com.cy.iokit.Morpheus</string>
  <key>CFBundleInfoDictionaryVersion</key>
  <string>6.0</string>
  <key>CFBundlePackageType</key>
  <string>KEXT</string>
  <key>CFBundleSignature</key>
  <string>????</string>
  <key>CFBundleVersion</key>
  <string>1.2.0</string>
  <key>IOKitPersonalities</key>
  <dict>
          <key>ISD105-ADD</key>
          <dict>
                  <key>CFBundleIdentifier</key>
```

```
                <string>com.cy.iokit.Morpheus</string>
                <key>IOClass</key>
                <string>com_isd_driver_USS725_Device</string>
                <key>IOProviderClass</key>
                <string>IOUSBDevice</string>
                <key>idProduct</key>
                <integer>23041</integer>
                <key>idVendor</key>
                <integer>1451</integer>
        </dict>
        …
  </dict>
  <key>OSBundleLibraries</key>
  <dict>
        <key>com.apple.iokit.IOSCSIArchitectureModelFamily</key>
        <string>1.0</string>
        <key>com.apple.iokit.IOStorageFamily</key>
        <string>1.1</string>
        <key>com.apple.iokit.IOUSBFamily</key>
        <string>1.8</string>
        <key>com.apple.kernel</key>
        <string>1.1</string>
        <key>com.apple.kernel.iokit</key>
        <string>1.1</string>
  </dict>
  <key>OSBundleRequired</key>
  <string>Local-Root</string>
</dict>
</plist>
```

Listing 1.

Each USB device supported by the driver has it's own device dictionary entry in the IOKitPersonalities dictionary. To enable interrupt support for a device, an additional entry needs to be made in the device dictionary. The enable_interrupt key with a Boolean value of True will tell the driver to enable interrupt support. Listing 2 shows an example device dictionary with the enable_interrupt entry.

```
        <key>ISD300-Interrupt Test</key>
        <dict>
                <key>CFBundleIdentifier</key>
                <string>com.cy.iokit.Morpheus</string>
                <key>IOClass</key>
                <string>com_isd_driver_ISDMSC_Device</string>
                <key>IOProviderClass</key>
                <string>IOUSBDevice</string>
                <key>enable_interrupt</key>
                <true/>
                <key>idProduct</key>
                <integer>97</integer>
                <key>idVendor</key>
                <integer>1451</integer>
        </dict>
```

Listing 2.

When a KEXT is initialized it is passed a handle to the device dictionary that was selected during the matching process. If the device dictionary passed to the Cypress MSC Driver contains the enable_interrupt entry, the driver enables the USB Interrupt Pipe endpoint, sets up a completion routine and then begins IN requests on this endpoint.

When the Interrupt Pipe's IN request completes, the completion routine setup earlier gets called. The intent of the Interrupt pipe mechanism is to notify software on the host when an action has occurred on the device, such as when a button has been pressed by the user. It's the completion routine's job to package up the data returned from the interrupt pipe and send it in the form of a message to any waiting clients. The completion routine also restarts the IN requests on the interrupt pipe to repeat the cycle.

Client Software Implementation

In order that there it's perfectly clear, any client software that wishes to receive notification from a KEXT needs to be loaded and registered it's interest in receiving the notifications prior to the actual notification being sent. So, the client software that will receive the notification needs to be written as a Startup Item or a background task that gets loaded and waits for the interrupt pipe event.

Now, the real issue in implementing the notifications has to due with crossing the kernel boundary - The client software is in user space and the Cypress KEXT is in the kernel. The way this is done is to use an IOKit method called IOServiceAddInterestNotification to create the mach port need for the communications to occur. IOKit is a framework in the kernel in which all kernel extensions are based upon.

The first thing the client needs to do is create a master mach port to communicate with IOKit. This is done with the following call:

```
kern_return_t IOMasterPort( mach_port_t bootstrapPort,
                            mach_port_t * masterPort);
```

Pass a `MACH_PORT_NULL` or `bootstrap_port` for the first parameter to `IOMasterPort`. Both are predefined constants indicating we want the default master mach port. The second parameter passed in is a pointer to the mach port, which will contain the master mach port when the function returns.

With a mach port in hand a notification object can be created that can receive IOKit notifications by calling `IONotificationPortCreate()`.

```
IONotificationPortRef IONotificationPortCreate( mach_port_t masterPort);
```

IONotificationPortCreate() uses the master mach port in order to talk to IOKit then creates and returns a special communications port to handle notifications between this object and the kernel. The notification object will then be passed as one of the parameters to IOServiceAddInterestNotification() which finally does the act of registering to receive the notifications from the kernel.

```
kern_return_t IOServiceAddInterestNotification(
      IONotificationPortRef   notifyPort,
      io_service_t            service,
      const io_name_t         interestType,
      IOServiceInterestCallback callback,
      void *                  refCon,
      io_object_t *           notification );
```

The first parameter to IOServiceAddInterestNotification() is the notification object we created with IONotificationPortCreate ().

The second parameter to IOServiceAddInterestNotification() is a reference to an IOService object in the system to match against. IOService is the base class for all IOKit classes so a reference to an IOService object could be a generic reference to almost any object in the kernel. In this case we can use the most general IOService object possible - the root of the service plane. We get this by calling IORegistryEntryFromPath() which looks up an IOService object in the IORegistry. This call looks like:

```
obj = IORegistryEntryFromPath( masterPort, kIOServicePlane ":/");
```

The third parameter to IOServiceAddInterestNotification() identifies the type of notification used by the driver, and this is important. In order for this message to make it out of the kernel we use a type of general interest defined in IOKit.h as kIOGeneralInterest.

The forth parameter is a callback function called when the notification fires. The callback function passed in needs to have the signature of:

```
void IOServiceInterestCallback(
      void *                  refcon,
      io_service_t            service,
      natural_t               messageType,
      void *                  messageArgument );
```

The callback function will be passed the refcon, passed in as the fifth parameter to IOServiceAddInterestNotification (), as well as a reference to the IOService object that sent the message (hopefully this is the Cypress driver). The callback function is also passed messageType and messageArgument, these usually provide the state change information, in this case the messageType is a constant (0xBEEF) identifying the message as the Interrupt Message passed from the Cypress driver and messsageArgument is the interrupt data. The callback function can use this information to perform the task the Client was developed to perform.

The final setup work prior to receiving the callback is to set up for asynchronous notifications since the client application most likely won't want to block waiting for an interrupt to complete. To do this the client needs to add the notification object's run loop event source to the program's run loop. This done by getting the object's run loop event source from IONotificationPortGetRunLoopSource() and calling CFRunLoopAddSource() . This allows the notification object to deliver notifications to a CFRunLoop client.

And that's it. Sample code for a client is listed in Listing 3.

```
#include <ctype.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <limits.h>

#include <mach/mach_interface.h>
#include <IOKit/IOKitLib.h>
#include <CoreFoundation/CFRunLoop.h>

mach_port_t         masterPort;
mach_port_t         notifyPort;
CFRunLoopSourceRef  cfSource;

#define kCYInterruptInterest                    0xBEEF

void ServiceInterestCallback(void * refcon, io_service_t service,
                            natural_t messageType, void * messageArgument )
{
    io_name_t       name;

    assert( refcon == (void *) masterPort );

    if ( messageType == kCYInterruptInterest )
    {
        printf("** kCYInterruptInterest messageType recieved with arg %p\n", messageArgument);
        // Do something...
    }
    else
    {
        if( KERN_SUCCESS == IORegistryEntryGetName( service, name ))
            printf(name);
        else
            printf("???");

        printf(": messageType %x, arg %p\n",  messageType, messageArgument);
    }
}


// NotifyTest registers for notification of state changes in an IOService specified
// by calling IOServiceAddInterestNotification.
void NotifyTest( void )
{
    kern_return_t   kr;
    io_iterator_t   note;
    io_service_t    obj;
    const char *    type;
    IONotificationPortRef notify;


    // Using the IOMasterPort setup a notification object for receiving IOKit notifications
    assert( 0 != ( notify = IONotificationPortCreate( masterPort )));


    // They type of notification used by the driver is of the type "General Interest"
    type = kIOGeneralInterest;
```

```
    // The root of the service plane is the most general path possible.
    obj = IORegistryEntryFromPath( masterPort, kIOServicePlane ":/");
    assert( obj );
    assert( KERN_SUCCESS == (
    kr = IOServiceAddInterestNotification(
            notify,
            obj,
            type,
            &ServiceInterestCallback,
            (void *) masterPort,
            &note )
    ));


    // To set up asynchronous notifications, add the notification object's
    // run loop event source to the program's run loop
    CFRunLoopAddSource(CFRunLoopGetCurrent(),
                       IONotificationPortGetRunLoopSource(notify),
                       kCFRunLoopDefaultMode);

    printf("waiting...\n");


    //Start the run loop so notifications will be received
    CFRunLoopRun();

    // Deallocate notification object returned from IOServiceAddInterestNotification
    IOObjectRelease( note );
}



int
main(int argc, char **argv)
{
    kern_return_t           kr;

    /*
     * Get master device port
     */
    assert( KERN_SUCCESS == (
    kr = IOMasterPort(   bootstrap_port,
                  &masterPort)
    ));

      NotifyTest( );

    printf("Exit\n");
    return kr;
}
```

Listing 3.