Access IS
Interfacing Solutions

Unit 18 Suttons Business Park
Reading RG6 1AZ, UK
Tel: +44 (0) 118 966 3333
Fax: +44 (0) 118 926 7281
e-mail: sales@access-is.com

Android HID Interface Overview

Document Revision

Version	*Date*	*Author*	*Revision*
1.0	06-09-2011	WDW	Initial Version
1.1	05-12-2012	AH	Added OCR Data Parsing with JNI/NDK
1.2	12-12-2013	AH	Updated Parser to New Version
1.3	11-03-2014	AH	Added support for OCR315 family and new method of connecting/writing to these devices

Overview

This document gives an overview as to how to interface with our devices using the HID interface.

Android 3.1 and newer supports USB Host at the operating system level.
This makes it easier to implement without having to provide a native interface.
But the device still has to provide a full powered USB Host port at the hardware level, with a generic HID driver in the operating system. We cannot guarantee that every tablet that is supplied with Android 3.1, implements the hardware and generic HID driver.

When connecting our devices to an Android device, Virtual Keyboard, CDC and HID modes are acceptable.

- **Virtual Keyboard mode** will depend on the device, but the vast majority of Android devices will support this. If you are unsure, please contact the manufacturer of the Android device to ensure it will support USB keyboards. When using the Virtual Keyboard/Keyboard Wedge mode, the MRZ data are outputted as a keyboard input.
- **CDC (virtual serial)** mode will allow data to be sent via a virtual serial port. The vast majority of Android devices will support this. If you are unsure, please contact the manufacturer of the Android device.
- **HID mode** is covered by this document. You can receive the MRZ data or parse them using our Android API.

If you have any other queries, please do not hesitate to contact your Access IS Sales Representative.

Source Code

We only supply sample source code, and not guaranteed fully tested working code. This document gives an overview on the interface and processes required when interfacing with Access HID devices.

Where precompiled libraries are supplied (without sources), these are fully supported by Access IS. Any problems with these libraries should be reported directly to Access IS.

Requirements

Android version 3.1 and newer (minSdkVersion=12) – This version supports the minimum implementation required.
Full Powered USB Host Port – This powers the device and provides the communication interface.
Warning: the Android device cannot be charged via the USB port while in this mode. The device must have enough battery capacity to run the application and power the USB device, or be powered from another external source.
Android Generic HID USB Host Support – The device needs to provide a generic HID driver as well as implement the Android 3.1 USB support.

Recommendations

If you need to use the device at all times, then you need to create the application as a service (http://developer.android.com/reference/android/app/Service.html).

Documentation for using USB Host can be found at http://developer.android.com/guide/topics/usb/host.html.

Data returned by the USB HID interface is contained in reports. These are byte arrays of a specific length.

Device IDs

This is a list of devices covered in this document, and will be updated on an as needed basis.

Device	OEM	Desktop	Vendor ID	Product ID	Read Interface	Write Interface	Write End Point Index
OCR310	☑		DB5 (hex) 3509 (dec)	137 (hex) 311 (dec)	0	1	2
OCR312		☑	DB5 (hex) 3509 (dec)	137 (hex) 311 (dec)	0	1	2
OCR313		☑	DB5 (hex) 3509 (dec)	137 (hex) 311 (dec)	0	1	2
OCR315	☑	☑	DB5 (hex) 3509 (dec)	137 (hex) 311 (dec)	0	1	2
OCR316	☑	☑	DB5 (hex) 3509 (dec)	137 (hex) 311 (dec)	0	1	2

The OCR312/313 and the OCR315/316 are paired with the same VID and PID. The only difference is the inclusion of a MSR head which does not affect the devices IDs.

Device	OEM	Desktop	Vendor ID	Product ID	Read Interface	Write Interface	Write End Point Index
OCR310 MKII*	☑		DB5 (hex) 3509 (dec)	13E (hex) 318 (dec)	0	0	0
OCR315 MKII*	☑	☑	DB5 (hex) 3509 (dec)	13E (hex) 318 (dec)	0	0	0
OCR316 MKII*	☑	☑	DB5 (hex) 3509 (dec)	13E (hex) 318 (dec)	0	0	0

*All MKII devices are due to be released in Q2 2014. They will be based on our Next Generation OCR head. Product names and further details are to be confirmed. Please do not hesitate to contact your Sales Representative to obtain additional information.

Prefix and Suffix Information

Data received from an OCR31x device will be prefixed and suffixed differently depending on whether the data is OCR or MSR data.

Data Type	Prefix	Suffix
OCR	0x1C 0x02	0x03 0x1D
MSR	0x0E 0x02	0x03 0x0F

AndroidManifest.xml

Recommended

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      ...
       >
    <uses-sdk android:minSdkVersion="12" />
    <uses-feature android:name="android.hardware.usb.host" />
```

AccessHIDInterface.java

Broadcast Receiver

```
/*
 * Receives a requested broadcast from the operating system.
 * In this case the following actions are handled:
 *             USB_PERMISSION
 *             UsbManager.ACTION_USB_DEVICE_DETACHED
 *             UsbManager.ACTION_USB_DEVICE_ATTACHED
 */
private final BroadcastReceiver usbReceiver = new BroadcastReceiver()
{
        @Override
        public void onReceive(Context context, Intent intent)
        {
                String action = intent.getAction();
                // Validate the action against the registered actions
                …
        }
};
```

Requesting Permission

In order to connect to the device, the user has to give their permission. This must be implemented in the Broadcast Receiver that will handle the response to the question that is displayed to the user.

The variable

private static final String ACTION_USB_PERMISSION = "com.access.device.USB_PERMISSION";
is a predefined variable for registering to receive device permissions.

The BroadcastReceiver code

```
if (ACTION_USB_PERMISSION.equals(action))
{
	// A permission response has been received, validate if the user has
	// GRANTED, or DENIED permission
	synchronized (this)
	{
		UsbDevice deviceConnected =
(UsbDevice)intent.getParcelableExtra(UsbManager.EXTRA_DEVICE);

		if (intent.getBooleanExtra(UsbManager.EXTRA_PERMISSION_GRANTED, false))
		{
			// Permission has been granted, so connect to the device
			// If this fails, then keep looking
			if (deviceConnected != null)
			{
				// call method to setup device communication
				currentDevice = deviceConnected;
				if (ShowDebuggingToasts)
				{
					Toast.makeText(hostDisplayActivity, "Device Permission
Acquired", Toast.LENGTH_SHORT).show();
				}
				if (!ConnectToDeviceInterface(currentDevice))
				{
					if (ShowDebuggingToasts)
					{
						Toast.makeText(hostDisplayActivity, "Unable to
connect to device", Toast.LENGTH_SHORT).show();
					}
				}
					sendEnabledMessage();
					// Create a listener to receive messages from the host
					...
				}
			}
			else
			{
				// Permission has not been granted, so keep looking for another
				// device to be attached....
				if (ShowDebuggingToasts)
				{
					Toast.makeText(hostDisplayActivity, "Device Permission
Denied", Toast.LENGTH_SHORT).show();
				}
				currentDevice = null;
			}
		}
	}
```

```
}
```

Registering the intent

```
usbManager = (UsbManager) hostDisplayActivity.getSystemService(Context.USB_SERVICE);
permissionIntent = PendingIntent.getBroadcast(hostDisplayActivity, 0, new
Intent(ACTION_USB_PERMISSION), 0);
IntentFilter permissionFilter = new IntentFilter(ACTION_USB_PERMISSION);
hostDisplayActivity.registerReceiver(usbReceiver, permissionFilter);
```

Requesting permission

Use this call to request the users permission to access the device. Before you connect to the device.

```
UsbDevice currentDevice = .....;
usbManager.requestPermission(currentDevice, permissionIntent);
```

Device Attached and Detached Intents

Handling a device attached or detached intent, requires that the user register them first.

Registering the intent

```
usbManager = (UsbManager) hostDisplayActivity.getSystemService(Context.USB_SERVICE);
IntentFilter deviceAttachedFilter = new IntentFilter();
deviceAttachedFilter.addAction(UsbManager.ACTION_USB_DEVICE_ATTACHED);
deviceAttachedFilter.addAction(UsbManager.ACTION_USB_DEVICE_DETACHED);
hostDisplayActivity.registerReceiver(usbReceiver, deviceAttachedFilter);
```

The BroadcastReceiver code

```
if (UsbManager.ACTION_USB_DEVICE_DETACHED.equals(action))
{
	// A device has been detached from the device, so close all the connections
	// and restart the search for a new device being attached
	UsbDevice device = (UsbDevice)intent.getParcelableExtra(UsbManager.EXTRA_DEVICE);
	if ((device != null) && (currentDevice != null))
	{
		if (device.equals(currentDevice))
		{
			// call your method that cleans up and closes communication with the
device
			CleanUpAndClose();
			if (hostDeviceStatusListener != null)
			{
				try
				{
					hostDeviceStatusListener.AccessDeviceDisconnected();
				}
				catch (Exception ex)
				{
					// An exception has occured
				}
			}
		}
	}
}
```

```
else if (UsbManager.ACTION_USB_DEVICE_ATTACHED.equals(action))
{
	// A device has been attached. If not already connected to a device,
	// validate if this device is supported
	UsbDevice searchDevice = (UsbDevice)intent.getParcelableExtra(UsbManager.EXTRA_DEVICE);
	if ((searchDevice != null) && (currentDevice == null))
	{
		// call your method that cleans up and closes communication with the device
		ValidateFoundDevice(searchDevice);
	}
}
```

Validating the Connected Device

The first step must be validating that this device is one that you support. The easiest way to do this is by validating the devices Vendor ID and Product ID's.

A vendor id is a global identifier for the manufacturer. For Access, the vendor id is 0xDB5. A product id refers to the product itself, and is unique to the manufacturer. The vendor id, product id combination refers to a particular product manufactured by a vendor.

The relevant Product IDs are at the beginning of this document.

```
searchDevice.getProductId()
searchDevice.getVendorId()
```

Connecting to the Device

If you are reading and writing, then the device can either have two end points on a single interface, or two interfaces each with a single end point. Either way, it is best if you know which interface you need to use and which end points. For example, the OCR312 uses two interfaces, one for reading and one for writing.

```
UsbInterface usbInterfaceRead = null;
UsbInterface usbInterfaceWrite = null;
UsbEndpoint ep1 = null;
UsbEndpoint ep2 = null;
boolean UsingSingleInterface = false;

if (UsingSingleInterface)
{
	// Using the same interface for reading and writing
	usbInterfaceRead = connectDevice.getInterface(0x00);
	usbInterfaceWrite = usbInterfaceRead;
	if (usbInterfaceRead.getEndpointCount() == 2)
	{
		ep1 = usbInterfaceRead.getEndpoint(0);
		ep2 = usbInterfaceRead.getEndpoint(1);
	}
}
else	// if (!UsingSingleInterface)
{
	usbInterfaceRead = connectDevice.getInterface(0x00);
```

```
        usbInterfaceWrite = connectDevice.getInterface(0x01);
        if ((usbInterfaceRead.getEndpointCount() == 1) && (usbInterfaceWrite.getEndpointCount()
== 1))
        {
                ep1 = usbInterfaceRead.getEndpoint(0);
                ep2 = usbInterfaceWrite.getEndpoint(0);
        }
}

if ((ep1 == null) || (ep2 == null))
{
        return false;
}

// Determine which endpoint is the read, and which is the write
if (ep1.getType() == UsbConstants.USB_ENDPOINT_XFER_INT)
{
        if (ep1.getDirection() == UsbConstants.USB_DIR_IN)
        {
                usbEndpointRead = ep1;
        }
        else if (ep1.getDirection() == UsbConstants.USB_DIR_OUT)
        {
                usbEndpointWrite = ep1;
        }
}
if (ep2.getType() == UsbConstants.USB_ENDPOINT_XFER_INT)
{
        if (ep2.getDirection() == UsbConstants.USB_DIR_IN)
        {
                usbEndpointRead = ep2;
        }
        else if (ep2.getDirection() == UsbConstants.USB_DIR_OUT)
        {
                usbEndpointWrite = ep2;
        }
}
if ((usbEndpointRead == null) || (usbEndpointWrite == null))
{
        return false;
}
connectionRead = usbManager.openDevice(connectDevice);
connectionRead.claimInterface(usbInterfaceRead, true);

if (UsingSingleInterface)
{
        connectionWrite = connectionRead;
}
else // if (!UsingSingleInterface)
{
        connectionWrite = usbManager.openDevice(connectDevice);
        connectionWrite.claimInterface(usbInterfaceWrite, true);
}

// Start the read thread
```

Writing to the Device

As USB devices work with reports of a specific length. It is best to format the data being sent into a ByteBuffer of the correct length. The length required can be retrieved from the endpoint.

```
int bufferDataLength = usbEndpointWrite.getMaxPacketSize();
```

The report must be formatted correctly for the device being connected to. For Access OCR/MSR devices, the format is as follows:

Position	Length	Description
1	1	Write End Point Index as defined in the Device IDs table at the beginning of the document
2	1	This byte consists of two values. Bit 1-7 – Data Length in report Bit 8 – Enable Device Flag
3+	Length specified in byte 2.	Data being sent to the device.

For MSR/OCR devices an empty report with the 8th bit of the length byte set on (enable reading), must be sent for the device to start reading. The same procedure, but with the 8th bit set to off (disable reading) must be followed when disconnecting from the device.

```
int bufferDataLength = usbEndpointWrite.getMaxPacketSize();
ByteBuffer buffer = ByteBuffer.allocate(bufferDataLength + 1);
UsbRequest request = new UsbRequest();

buffer.put(DataToSend);

request.initialize(connectionWrite, usbEndpointWrite);
request.queue(buffer, bufferDataLength);
try
{
	if (request.equals(connectionWrite.requestWait()))
	{
		return true;
	}
}
catch (Exception ex)
{
	// An exception has occured
}
```

Reading from the Device

As USB devices work with reports of a specific length. The data received from the device will always be the size specified by the report length, even if the report doesn't contain enough data to fill it.

If you are expecting unsolicited data from the device, then a read thread should be started so that the data can be processed as soon as it arrives.

```
int bufferDataLength = usbEndpointRead.getMaxPacketSize();
ByteBuffer buffer = ByteBuffer.allocate(bufferDataLength + 1);
UsbRequest requestQueued = null;
UsbRequest request = new UsbRequest();
request.initialize(connectionRead, usbEndpointRead);

try
{
	while (!getStopping())
	{
		request.queue(buffer, bufferDataLength);
		requestQueued = connectionRead.requestWait();
		if (request.equals(requestQueued))
		{
			byte[] byteBuffer = new byte[bufferDataLength + 1];
			buffer.get(byteBuffer, 0, bufferDataLength);

			// Handle data received
			...

			buffer.clear();
		}
		else
		{
			Thread.sleep(20);
		}
	}
}
catch (Exception ex)
{
	// An exception has occured
}
try
{
	request.cancel();
	request.close();
}
catch (Exception ex)
{
	// An exception has occured
}
```

Known Problems

"Jelly Bean" – Android v4.2
At the time of writing, there are certain issues with locking and unlocking an Android v4.2 device with an attached USB device. This does not affect earlier versions of Android. This bug has been logged with the Android team as Issue 38191 - http://code.google.com/p/android/issues/detail?id=38191 – (link correct at time of writing).

Screen Orientation Fix

When a USB device is attached and an application, such as described above, is running and the screen rotated, the OS may attempt to reconnect to the USB device. This can be prevented by adding the following line to the AndroidManifest.xml file in the activity where the USB connection is active.

```
Android:configChanges="orientation"
```

Parsing OCR Data

Data read from ICAO9303 compliant documents can be parsed and validated to provide individual fields of the decoded data from machine-readable identities. A list of fields is provided below. Access IS provides a pre-built JNI/Android NDK library for this purpose. The steps to implement this are provided below.

AccessParserNDKInterface.java

To provide an interface to the library file, this Java file must be created within the package *com.accessltd.device*. The file loads the library and provides method prototypes for available functions. At version 1.2, the library does not yet support parsing MSR data.

```
package com.accessltd.device;


public class AccessParserNDKInterface {

	static {
		System.loadLibrary("AccessHIDNDK");

	}

	public native String AccessHIDParseLastError();
	public native String AccessHIDParseOCR(String Line1, String Line2, String Line3,
boolean Validate);
	public native String AccessHIDParseMSR(String Track1, String Track2, String Track3,
}
```

```
AccessHIDParseLastError() – returns the last error string from AccessHIDParseOCR()
AccessHIDParseOCR() – performs the parsing of the OCR
AccessHIDParseMSR() – reserved – will allow the parsing of MSR data
```

libAccessHIDNDK.so

This file is the pre-built library file. It should be placed in *libs/armeabi/* within the main project workspace.

Main Activity

Within the main activity (or any suitable activity or service within the application), add the following import statement:

```
import com.accessltd.device.AccessParserNDKInterace;
```

The AccessParserNDKInterface class must then be instantiated:

```
private AccessParserNDKInterface accessParserNDKInterface = new AccessParserNDKInterface();
```

After this the methods of the class may be called to parse the lines of OCR data received. For example:

```
resultString = accessParserNDKInterface.AccessHIDParseOCR(Line[0], Line[1], Line[2],
validateOCR);
```

where validateOCR is Boolean; the document is automatically validated (this variable is included for compatibility with previous versions), Line is an array of Strings, each element is a line of decoded text and resultString is a String returned by the function containing the parsed data separated by CRLFs – (0x0D 0x0A).

Return Data

The OCR parsing method returns the parsed data as a string with fields separated by "\r\n" - (0x0D 0x0A).

These fields are:

Field	Data Type	Explanation	Example
DOB	String	Date of Birth, usually in the form YYMMDD	810213
Expiry	String	Expiry date of document (usually YYMMDD)	140810
Issuer	String	3 character country code of document issuer	GBR
Document Type	String	1 or 2 character document type	ID
Last Name	String	Last name	SMITH
First Name	String	First name(s) – may be truncated to fit field on document	JOHN
Nationality	String	3 character country code of document holder	GBR
Discretionary	String	Supplementary data on document	627006
Discretionary2	String	Supplementary data on document	8
Document Number	String	Unique document identifier	H123456789
Sex	char	Gender (M/F/X)	M

To split this string into an array of strings, use:

```
token = resultString.split("\n");
```

This is split on \n (0x0A) and not \r\n to ensure that any field with blank lines are still returned. This can be removed using the trim() method.