USB Generic Driver Programmer's Guide

Date: December 14, 2006

Abstract

This document provides a functional description of a USB generic device driver.

Contents

Preface

This document describes the USB generic driver, SYS$UGDRIVER.EXE, which allows users to support USB devices such as scanners and smart card readers without having to write a USB device driver. This is analogous to GKDRIVER for SCSI, which enables programmers to interface to SCSI devices without having to write a full OpenVMS device driver.

Associated Documents

- *Universal Serial Bus Specification Revision 2.0*

Change History

Date	Issue #	Description/Summary of Changes
January 6, 2005	x0.1	Initial version.
January 18, 2005	x0.2	Add ability to send control pipe request fix some typing mistakes.
March 30, 2005	x0.3	Make changes corresponding to new code.
April 13, 2005	x0.4	Add some more corrections found while writing an example program.
August 17, 2005	x0.5	Convert functional specification to draft programmers guide.
October 17, 2005	1.0	Add section on device configuration
November 30, 2005	1.1	More editing clean up and section on USB$_STALL errors
December 20, 2005	1.2	More comments added in.
December 14, 2006	1.3	Crank in some more minor corrections

Introduction

This document describes the USB generic driver SYS$UGDRIVER whose device name is UGAx. The generic driver allows users to support USB devices that are not part of the USB Human Interface Device (HID). This document does not tell you how to support a specific device because devices vary greatly. Rather, this document describes various capabilities of the generic driver and provides a simple example of how to use it.

USB Device Structure

A USB device is usually made up of one or more interfaces, with each interface having one or more possible configurations. Each interface consists of one or more communications paths called **pipes**. You can think of a pipe as behaving like a virtual circuit in a network.

One pipe, called the **control pipe**, is opened by default as part of identifying a device and matching up a driver for the device. The control pipe is a bi-directional pipe; you send commands out over the pipe and optionally receive data back.

Three other types of pipes are the **interrupt, bulk**, and **isochronous pipes**. OpenVMS currently does not support isochronous pipes. The interrupt pipe is used to report an insertion and removal of a card. Bulk input and bulk output pipes are used to move data on and off the card.

As part of configuring a device, the driver opens all the necessary pipes and sets the desired configuration.

Driver Model

This section describes a simple fictional device and lists the steps an application takes to use the generic USB driver to control the device. This fictional device is a smart card reader that does not conform to the smart card device class. This reader has one interface that uses the vendor-specific class “sub class” and protocol types of 0xff. It has a bulk-in pipe, a bulk-out pipe, an interrupt pipe, and the required control pipe. For now, assume that the steps necessary for the USB configuration to load a driver are complete. (How device configuration works and how to obtain the information necessary for configuration are discussed later.) So, with these assumptions, simply plug the device into the system.

Driver Actions

At this point, the generic driver has opened all the pipes for the chosen interface and is waiting for an application to assign a channel to it. The first channel assigned must be associated with the control pipe before it can be used for anything else.

1. The application now associates a channel to the interrupt pipe, the bulk-in pipe, and the bulk-out pipe.

2. The application next determines the type of pipe it has and other data about the device that it needs by using the IO$_SETCHAR and IO$_SENSECHAR functions.

3. The application then issues a “read” to the interrupt to determine if a card is present in the reader. If a card is present, the application uses the control pipe and the bulk in and bulk out pipes to exchange data with the smart card.

User Interface

$QIO functions supported

This section describes the $QIO function codes that this driver supports.

IO$_READxBLK

The driver treats read virtual, logical, and physical in the same way. Note that normal $QIO processing rules for logical and physical block I/O still apply and are enforced by the $QIO dispatching code. When a read is queued to a pipe, the driver checks to see if there is an outstanding I/O for that pipe. If one is found, the request is placed in the I/O queue of the pipe. If no I/O is outstanding, the driver starts the I/O queue for that pipe.

The driver treats parameters from the $QIO P1-P6 as follows:

P1	Address of buffer in which to store results
P2	Size of buffer in bytes
P3	Flag USB$_SHORT_XFER_OK allows fewer bytes than requested to complete the I/O
P4	Pipe handle

Status return codes are the usual OpenVMS ones for I/O devices. Because USB device status codes are a longword in length, after first checking the status word of the I/O status block, the application must check the second longword of the I/O status block. The second longword contains the USB status code for the request. The status word in the IOSB can indicate success but have a USB error in the second longword.

Xfer size bytes	VMS Status
USB Status	

IO$_WRITExBLK

The driver treats write virtual, logical, and physical in the same way. Note that normal $QIO processing rules for logical and physical block I/O still apply and are enforced by the $QIO dispatching code.

When a write is queued to a pipe, the driver checks to see if there is an outstanding I/O for that pipe. If one is found, the request is placed in the I/O queue of the pipe. If no I/O is outstanding, the driver starts an I/O queue for that pipe.

The driver treats parameters from the $QIO P1-P6 as follows:

P1	Address of buffer from which to read date

P2	Size of buffer in bytes
P3	Flag USB$_SHORT_XFER_OK allows fewer bytes than requested to complete the I/O
P4	Pipe handle

Status return codes are the usual OpenVMS ones for I/O devices. Because USB device status codes are a longword in length, after first checking the status word of the I/O status block, the application must check the second longword of the I/O status block. The second longword contains the USB status code for the request. (The status word in the IOSB can indicate success but have a USB error in the second longword.)

IO$_SETMODE/CHAR

Enable Unplug notification AST

This item allows an application to associate an AST that is delivered if a device is unplugged. You can use any channel to enable this AST. HP recommends that you use the control channel for this AST. To cancel the AST, do not supply an AST routine address and parameter. (Do you use the same command that you used before to enable it and omit the AST routine address and parameter?)

The driver treats parameters from the $QIO P1-P6 as follows:

P1	AST routine address
P2	AST parameter
P3	UG$_ENABLE_AST
P4	Access mode

Associate channel

Use this command to associate a VMS channel to a pipe and to break the association of a channel to a pipe.

The driver treats parameters from the $QIO P1-P6 as follows:

P1	Unused
P2	Unused
P3	UG$_ASSOCIATE associates a channel to a pipe; UG$_DISASSOCIATE breaks an association
P4	Pipe handle

Set pipe state

Use this command to set the state of a pipe.

The driver treats parameters from the $QIO P1-P6 as follows:

P1	Unused
P2	Pipe state values are UG$_PIPE_STATE_ACTIVE, UG$_PIPE_STATE_STALED, and UG$_PIPE_STATE_IDLE.
P3	UG$_SET_PIPE_STATE
P4	Pipe handle

Send a control request

Use this command to send a device request to the device control pipe. For more details about device requests, see section 9.3 USB 1.1 or 2.0 specifications.. (Where does one get this document?)

The driver treats parameters from the $QIO P1-P6 as follows:

P1	Address of setup data; see table below.
P2	Must be 8.
P3	UG$_DEVICE_REQUEST.
P4	Pipe handle.
P5	Address of buffer to receive data if there is a data phase.
P6	Flag USB$_SHORT_XFER_OK allows fewer bytes than requested to complete the I/O.

The P1 buffer layout is shown below.

Offset	Field	Size	Description
0	bmRequestType	1	Characteristics of the request: B7: 0 - Host to device 1 - Device to host B6..5 TYPE 0 - Standard 1 - Class 2 - Vendor 3 - Reserved B 4..0 Recipient 0 - Device 1 - Interface 2 - Endpoint 3 - Other

			4…31 - Reserved
1	bRequest	1	See Table 9-3 in USB specification.
2	wValue	2	Word sized field varies according to request.
4	wIndex	2	Word sized field varies according to request.
6	wlength	2	Number of bytes to transfer if there is a data phase.

IO$_SENSEMODE/CHAR

Get number of pipes

Use this command to obtain the number of pipes. Make this the first operation that an application performs using the driver. Use the channel for the control connection for this operation.

The driver treats parameters from the $QIO P1-P6 as follows:

P1	Address of longword to store the number of pipes.
P2	Size of buffer in bytes must be 4.
P3	UG$_GET_PIPE_COUNT

Get pipe handles

Use this command to obtain all the pipe handles. The buffer must have one quadword for each pipe of the device.

The driver treats parameters from the $QIO P1-P6 as follows:

P1	Address of buffer to hold pipe handles
P2	Size of buffer in bytes
P3	UG$_GET_PIPE_HANDLES

Get Pipe direction

Use this command to obtain the direction of a pipe associated with its handle.

The driver treats parameters from the $QIO P1-P6 as follows:

P1	Address of buffer to store pipe direction. Legal returns are USB$_XFER_OUT, USB$XFER_IN, and USB$XFER_SETUP.
P2	Must be 4.
P3	UG$_GET_PIPE_TYPE

P4	Pipe handle

Get Pipe type

Use this command to obtain the type of pipe associated with its handle.

The driver treats parameters from the $QIO P1-P6 as follows:

P1	Address of buffer to store pipe type. Types are UG$_PIPE_TYPE_CONTROL, UG$_PIPE_TYPE_BULK, UG$_PIPE_TYPE_INTERRUPT, UG$_PIPE_TYPE_ISOCHRONOUS (The last type is currently not supported.)
P2	Must be 4.
P3	UG$_GET_PIPE_TYPE
P4	Pipe handle

Get pipe state

Use this command to obtain the state of the pipe.

The driver treats parameters from the $QIO P1-P6 as follows:

P1	Address of buffer to hold the pipe state. Values of the pipe state are UG$_PIPE_STATE_ACTIVE, UG$_PIPE_STATE_STALLED, UG$_PIPE_STATE_IDLE
P2	Must be 4
P3	UG$_GET_PIPE_STATE
P4	Pipe handle

Get pipe size

Use this command to obtain the size of the largest transfer on the pipe. (This is really the largest size that is sent on the bus in one transfer.) Actual requests can be larger. The driver takes care of splitting the transfer up into appropriately sized bus transfers.

The driver treats parameters from the $QIO P1-P6 as follows:

P1	Address of buffer to hold pipe size
P2	Must be 4
P3	UG$_GET_PIPE_SIZE
P4	Pipe handle

Get device descriptor

Use this routine is used to obtain the device descriptor from the device.

The driver treats parameters from the $QIO P1-P6 as follows:

P1	Address of buffer to receive the device descriptor. The format of the buffer is shown below.
P2	Size of buffer in bytes.
P3	UG$_GET_DEVICE_DESCRIPTOR

Format of the Device Descriptor

unsigned char	ug$b_blength	Descriptor length in bytes
unsigned char	ug$b_bdescriptortype	Descriptor type constant 0X01
unsigned short int	ug$w_bcdusb	BCD encoded specification release number
unsigned char	ug$b_bdeviceclass	Device class code
unsigned char	ug$b_bdevicesubclass	Device sub class code
unsigned char	ug$b_bdeviceprotocol	Device protocol
unsigned char	ug$b_bmaxpacket	Maximum packet size for control pipe; 8, 16, 32, 64 are valid.
unsigned short int	ug$w_idvendor	Vendor ID
unsigned short int	ug$w_idproduct	Product ID
unsigned short int	ug$w_bcddevice	BCD encoded device release number.
unsigned char	ug$b_imanufacturer	Index of string descriptor that describes the manufacturer.
unsigned char	ug$b_iproduct	Index of string descriptor that describes the product.
unsigned char	ug$b_iserailnumber	Index of string descriptor of device serial number.
unsigned char	ug$b_bnumconfigurations	Number of possible device configurations.

Get Interface descriptor

Use this command to obtain the interface descriptor from the device.

The driver treats parameters from the $QIO P1-P6 as follows:

P1	Address of buffer to receive the INTERFACE descriptor. The format of the buffer is shown below.
P2	Size of buffer in bytes
P3	UG$_GET_INTERFACE_DESCRIPTOR

Format of Interface Desciptor

unsigned char	ug$b_blength	Descriptor length in bytes.
unsigned char	ug$b_bdescriptortype	Descriptor type constant 0X04.
unsigned char	ug$b_binterfacenumber	Zero-based count of this interface.
unsigned char	ug$b_balternatesetting	Used to select alternate setting for the interface.
unsigned char	ug$b_bnumendpoints	Number of endpoints for the interface.
unsigned char	ug$b_binterfaceclass	Interface class code.
unsigned char	ug$b_binterfacesubclass	Interface sub class code.
unsigned char	ug$b_binterfaceprotocol	Interface protocol.
unsigned char	ug$b_iinterface	Index of string descriptor that describes this interface.

Cancel I/O

When you issue a cancel on a channel, the driver checks the I/O queue of the channel, flushes any queued requests, and returns them with a status of SS$_CANCEL. Any pending I/O to the pipe is aborted using the USB abort pipe code. In that situation, the status in the I/O status block is SS$_ABORT, and the second longword has the status that is returned from the aborted I/O.

If you deassign a channel, the association between the channel number and the pipe is broken. Deassigning the channel does not close the pipe. The pipes are closed only when the device is unplugged. Therefore, you can reuse a device without unplugging it from the system and plugging it back in.

Error Handling

It is impossible to tell you how to deal with most errors that you will encounter while developing code to support your device. There is one common error that you are likely to encounter when getting started that is USB$_STALL. That is the common way for USB devices to indicate, that the command they just received is invalid. Unfortunately, is also possible to receive this in normal operation if the device is simply too busy to acknowledge the request.

Example

An example program is in SYS$COMMON:[SYSHLP.EXAMPLES.USB],: ug_example.c. This program is intended as a simple example of how to use the UG driver to control a USB device. In this case, it loops two PL2303 USB to RS232 controllers and exchanges data. Note that this example does not exercise all the capabilities of the UG driver nor does it work on all PL2303-based controllers. Some PL230- based controllers require additional setup, which is not shown in this example.

To compile the example, copy the programs ug_example.c and ugdef.h from sys$common:[syshlp.examples.usb] to a local directory where you have write access. Then simply compile and link them; no special switches are needed. To run the program, you must add both PL2303 devices into the system. To do this, follow the steps in the "USB Device Configuration" section.

The example program follows steps that are the usual ones for any device you want to control.

1. Assign a channel to the device or devices.
2. Find out how many pipes the device has.
3. Make sure you are communicating with the correct device. The program does this by reading the device descriptor and checking it against what it expects to find.
4. Associate a VMS channel to a pipe and obtain the pipe type and direction.
5. Perform any device-specific setup that is required.
6. Exchange data with the device.

USB Device Configuration

USB device configuration is as simple as adding some text lines to SYS$USER_CONFIG.DAT; it is also simple to do wrong.

You perform USB configuration with the same files that you use to configure device controllers for OpenVMS: SYS$CONFIG.DAT and SYS$USER_CONFIG.DAT. Both files are located in the SYS$SYSTEM: directory. As you might expect, user-written drivers add their configuration records to SYS$USER_CONFIG.DAT; OpenVMS does not modify the contents – even across O/S upgrades.

The contents of the files are evaluated: SYS$USER_CONFIG.DAT is evaluated first, and SYS$CONFIG.DAT second, allowing a user-written configuration record to supersede a system-supplied record.

USB is different from normal OpenVMS device configuration in several respects:

- The devices are not classic bus-based controllers, but, rather, devices connected to a peripheral bus.
- Simple vendor/device identification matching, which is performed for other buses, is not sufficient to determine which driver to load for a USB device.
- USB device drivers are part of a larger "stack" of drivers; the controller port driver, the HUB driver, or the HID driver are involved in aspects of configuration and operation of the device. A USB device driver is a pseudo-driver in the sense that it does not directly talk to the device, but passes messages to other drivers that can talk to the USB bus and send messages to and receive messages from USB devices.
- The USB protocol was developed to allow device-to-driver matching to be done on multiple levels, depending on the type of device and the needs of the driver.
- Device discovery is asynchronous on the USB bus, and it is not feasible to poll the bus to find devices. Instead, devices are configured in response to an event from a HUB device indicating that it has a new device to report. HUBs are both external devices that provide additional slots, and a Root HUB is built onto the controller to which the initial UCB connections are attached.
- You can attach and remove devices at will, even at runtime. which requires USB drivers to be loaded on the fly as well as made offline on the fly.

The UGDRIVER is the basis of a "generic" driver. It is the functional equivalent of the SCSI GKDRIVER for USB devices; it implements simple logic that takes care of USB housekeeping and allows a user to read and write raw data packets to the USB device.

The next section describes how to (configure UGDRIVER to a specific device or a specific class of devices), and how to make sure that UGDRIVER does not interfere with the configuration of *other* devices and their drivers.

The Basics of Configuration

USB devices include the device itself and one or more *Interfaces.* Most devices present a single interface. An interface can be serviced by a single driver, or by multiple drivers. A single driver can also service multiple interfaces. This might sound complex, and it is. However, for the typical USB device, there is only one interface.

When a new device is discovered by a HUB, the HUB driver collects information about the device and sends a message to the USB Configuration Manager (UCM), which is a background process that hibernates, waiting to service configuration events. UCM is the code that knows how to perform device-to - driver matching and how to load device drivers. UCM also maintains an on-disk database of device-to-driver mappings that it previously performed and made permanent (persistent). This database allows a device always to obtain the same OpenVMS device name each time it is plugged in.

Plugging in a new device

The HUB driver collects information about the device and its interfaces, and then requests UCM to attempt to configure and load a device driver for it. The HUB driver does this in multiple steps:

- First it tries to configure the device as a "DEVICE," the simplest type of configuration; it ignores the interface information. Devices can be identified by ***vendor_id, product_id, revision, device_class, device_subclass, and device_protocol***.
- If a driver is not successfully configured, then the HUB driver asks UCM to try to configure the device as an "INTERFACE" -- for each interface the device presents (which is usually only one). Interfaces are identified by ***vendor_id, product_id, revision, interface_class, interface_subclass, and interface_protocol***. The vendor and product ID codes and revision value are inherited from the device.

Note: This discussion leaves out Human Interface Devices. These devices involve interaction with a Human,-- such as a mouse, keyboard, joystick, simulator, tablet, or game pad – and are handled by a special HID driver. HID devices are identified by a two-byte value of Usage Page and Usage Type; these values are combined into a 16-bit value "TAG," and device- driver matching is performed by searching for a matching TAG value. The UG driver can be used to talk to a HID device, but it cannot be loaded using the HID Usage Page/Type values. A second generic HID driver is needed for that purpose.

The Generic List

UCM now has the device information it needs to match to a device driver. To do this matching, it examines the Generic list. It has created this list by reading the SYS$USER_CONFIG.DAT file and the SYS$CONFIG.DAT file, searching for records that contain a private section with a USB_CONFIG_TYPE record.

The records in the file are simple; each record starts with a DEVICE keyword and ends with an END_DEVICE keyword. USB records are pseudo-devices in the sense that they provide no ADAPTER type and do not have a conventional device ID. Instead, using the BEGIN_PRIVATE and END_PRIVATE construct, they provide USB-specific information. Within this private data area, each line starts with a USB keyword.

USB keywords are in the following table.

Keyword	Description
USB_CONFIG_TYPE	Tells UCM how the driver is to be configured – as a DEVICE, INTERFACE or TAG method.
USB_CLASS_DRIVER	Used for specialized drivers that are class drivers for other USB drivers such as the HID driver. You do not need to use it. The values are SINGLE_INSTANCE and MULTIPLE_INSTANCE
VENDOR_ID	Vendor ID
PRODUCT_ID	Product ID

RELEASE_NUMBER	Revision number
DEVICE_CLASS	The device class code
DEVICE_SUB_CLASS	Device subclass
DEVICE_PROTOCOL	Device Protocol
NUMBER_OF_INTERFACES	The number of interfaces the device presents
BEGIN_INTERFACE	Starts an interface definition. (There can be multiple interface definitions.)
INTERFACE_CLASS	Interface class
INTERFACE_SUB_CLASS	Interface subclass
INTERFACE_PROTOCOL	Interface protocol
END_INTERFACE	Ends an interface definition
HID_USAGE_DATA	The Usage Page/Type TAG for HID devices
USAGE_TAG	An alternate TAG type used by HID-like drivers for performing TAG lookups -- for example, the EDGEPORT Serial Multiplexer uses this.
USB_LOGGING	Used to enable some extra logging (not available to normal drivers – used by CLASS drivers)

In addition, the standard DEVICE and DRIVER keywords must be included outside the BEGIN_PRIVATE and END_PRIVATE section, telling UCM the device name and driver name to use for the device.

UCM parses this data into a data structure and creates an in-memory Generic list of all the USB devices that are in the files. The queue is in the same order as the devices appear in the file, and the SYS$USER_CONFIG.DAT records come before the SYS$CONFIG.DAT records.

The data in this list is used to match against the configuration request that the HUB driver makes. The matching process is quite complex.

DEVICE Configuration

In DEVICE configuration, the HUB driver asks UCM to configure the device by DEVICE, not by INTERFACE or TAG.

In general, drivers do not use DEVICE configuration; rather, they use INTERFACE configuration. The most common use of DEVICE configuration is to load special device classes such as HUB devices. For a general driver, the only practical use of DEVICE configuration is to force the loading of a specific device driver, regardless of any other configuration records that might otherwise match.

The match logic for a device that has not been connected to the system before is not a simple comparison of all the fields in search of a match. The reason is that you a driver (and its configuration record) can match a variety of devices; this is a generic driver. Alternatively, you might have a vendor-specific driver.

The driver class code can be 0-255, and 255 can have special meanings: if the device code is zero, the device presents has no device class, no subclass, and no protocol; all of these fields are zero. If the class is 255 (0xFF), the protocol is vendor-specific and *must* match the vendor ID.

A set of tests determines whether a generic record matches the configuration request. The tests are not all equal: a “priority” is assigned to each test. All the generic records are scanned. A record that matches is compared against the previous match; if the new match has a greater priority, it is used. If no records have matched, a zero is used. This matching means that:

- Higher priority matches win over lower ones.
- Duplicate matches of the same priority ignore subsequent matches.

In this manner, records are created so that drivers are selected from *more* specific to *less* specific. The following tests are in order of priority -- from best match to worst match. When only a field is included, both the configuration request and the generic list entry field must match. When a generic field must be 0 (because omitting the field in the device record in the file sets it to zero), the request field is ignored.

Match 1:

- Vendor ID
- Product ID
- Release Number
- Device Class
- Device Subclass
- Device Protocol

Match 2:

- Vendor ID
- Product ID
- Release Number
- Device Class
- Device Subclass
- Generic Device Protocol must be 0

Match 3:

- Vendor ID
- Product ID
- Release Number
- Generic Device Class must be zero

Match 4:

- Vendor ID
- Product ID
- Generic Record Release Number must be 0
- Generic Device Class must be zero

Match 5:

- Generic Vendor ID must be 0
- Generic Product ID must be 0
- Generic Release Number must be 0
- Device Class (not 255)
- Device Subclass
- Device Protocol

Match 6:

- Generic Vendor ID must be 0
- Generic Product ID must be 0
- Generic Release Number must be 0
- Device Class (not 255)
- Device Subclass

The matching tests show that an entry that is fully qualified always matches before a more generic one.

Note that there is no explicit testing for a Device Class of 0 because the standard requires that devices with a class field of zero have the subclass and protocol set to zero. The preceding tests handle classes of zero correctly.

All tests in which the device class cannot be 255 require that the generic record contain no vendor ID (and, by implication, no product ID and no Release Number). This allows the HUB record, for example, which has no vendor or product IDs, to match against all devices with a class code of 9. However, a user record that provides only the vendor and product IDs claims a device with a class code of 9 over the generic HUB record.

The tests might be tuned to provide a finer granularity, but, in general, the current tests provide all the control a user might need for configuring a device.

INTERFACE Configuration

An interface configuration means that the HUB driver asks UCM to configure the device by INTERFACE -- not by DEVICE or TAG.

The match logic for a device interface that has not been connected to the system before is not simply a comparison of all the fields looking for a match. This is because you can have an interface driver (and a configuration record for it) that can match a variety of devices; this is a generic driver??????. On the other hand, you might have a very vendor-specific driver.

The interface class code can be 0 through 255.. The value 255 has a special meaning: if the class is 255 (0xFF), the interface is vendor-specific and *must* match the vendor ID.

A set of tests determines if a generic record matches the configuration request. The tests are not all equal – a “priority” is assigned to each test, and all the generic records are scanned. A record that matches is compared against the previous match (or against zero if no matches are found). If the new match has a greater priority, it is used. This matching means that:

- Higher priority matches win over lower ones.
- Duplicate matches of the same priority ignore subsequent matches.

In this manner of matching, records can be created so that drivers are selected from *more* specific to *less* specific. The following tests are in order of priority -- from best match to worst match. When only one field is given, both the configuration request and the generic list entry field must match. When a generic field must be 0 (because omitting the field in the device record in the file sets it to zero), the request field is ignored.

Match 1:

- Vendor ID
- Product ID
- Interface Class
- Interface Subclass
- Interface Protocol

Match 2:

- Vendor ID
- Product ID
- Interface Class
- Interface Subclass
- Generic Interface Protocol must be 0

Match 3:

- Vendor ID
- Interface Class must be 255
- Interface Subclass
- Interface Protocol

Match 4:

- Vendor ID
- Interface Class must be 255
- Interface Subclass
- Generic Interface Protocol must be 0

Match 5:

- Generic Vendor ID must be 0
- Interface Class must not be 255
- Interface Subclass
- Interface Protocol

Match 6:

- Generic Vendor ID must be 0
- Interface Class must not be 255
- Interface Subclass

Just as in device matching, the order is from strongest match to weakest match, from more specific to less specific, from vendor-specific to generic.

To use an example…

You might find an inexpensive tablet on the internet and want to write a driver for it. The first thing you need to do to configure the device is to obtain its device information. Therefore, you must plug it in.

Using the UCM command SHOW EVENT, you can look at events on the USB bus.

```
UCM> show event/since=today
Date        Time        Type         Priority Component
--------------------------------------------------------------------------------
15-OCT-2005 13:23:14.54 DRIVER       NORMAL   HUBDRIVER
        Message: Configured device UCM0 using driver SYS$HUBDRIVER:

15-OCT-2005 13:23:16.83 DRIVER       NORMAL   HUBDRIVER
        Message: Configured device UCM0 using driver SYS$HUBDRIVER:

15-OCT-2005 13:25:05.27 DRIVER       NORMAL   HUBDRIVER
        Message: Configured device HID0 using driver SYS$MOUDRIVER:

UCM>
```

This example shows the events from today. The first two are HUB devices; the last event, however, is your device. To obtain more information, ask for INFORMATIONAL events:

```
UCM> sho event/since=today/priority=informational
Date        Time        Type         Priority Component
--------------------------------------------------------------------------------
15-OCT-2005 13:23:14.52 DRIVER       INFORMATIONAL HUBDRIVER
        Message: Find a driver for DeviceClass/DeviceSubClass = 0x9/0x0

15-OCT-2005 13:23:14.52 DRIVER       INFORMATIONAL HUBDRIVER
        Message: Find a driver for DeviceClass/DeviceSubClass = 0x9/0x0

15-OCT-2005 13:23:14.54 UNKNOWN      INFORMATIONAL UCM DEVICE UCM0
        Message: VENDOR_ID = 4113
                 PRODUCT_ID = 0
                 RELEASE_NUMBER = 0
                 BUS_NUMBER = 0
                 PATH = 0.0.0.0.0.0
                 DEVICE_CLASS = 9
                 DEVICE_SUB_CLASS = 0
                 DEVICE_PROTOCOL = 0
                 NUMBER_OF_INTERFACES = 1
                 NUMBER_OF_CONFIGURATIONS = 1
                 CONFIGURATION_NUMBER = 0.

15-OCT-2005 13:23:14.54 UCM          INFORMATIONAL SYS$HUBDRIVER.EXE
        Message: Loaded single instance class driver for UCM0.

15-OCT-2005 13:23:14.77 DRIVER       INFORMATIONAL HUBDRIVER
        Message: Find a driver for DeviceClass/DeviceSubClass = 0x9/0x0

15-OCT-2005 13:23:16.83 UNKNOWN      INFORMATIONAL UCM DEVICE UCM0
        Message: VENDOR_ID = 1033
                 PRODUCT_ID = 89
                 RELEASE_NUMBER = 256
                 BUS_NUMBER = 1
                 PATH = 1.0.0.0.0.0
                 DEVICE_CLASS = 9
                 DEVICE_SUB_CLASS = 0
                 DEVICE_PROTOCOL = 0
                 NUMBER_OF_INTERFACES = 1
                 NUMBER_OF_CONFIGURATIONS = 1
                 CONFIGURATION_NUMBER = 0.

15-OCT-2005 13:23:16.83 UCM          INFORMATIONAL SYS$HUBDRIVER.EXE
        Message: Loaded single instance class driver for UCM0.

15-OCT-2005 13:25:04.94 DRIVER       INFORMATIONAL HUBDRIVER
        Message: Find a driver for DeviceClass/DeviceSubClass = 0x0/0x0

15-OCT-2005 13:25:04.94 DRIVER       INFORMATIONAL HUBDRIVER
        Message: Find a driver for InterfaceClass/InterfaceSubClass/Protocol = 0
x3/0x0/0x0

15-OCT-2005 13:25:04.99 UNKNOWN      INFORMATIONAL UCM DEVICE HID0
        Message: VENDOR_ID = 2250
                 PRODUCT_ID = 16
                 RELEASE_NUMBER = 261
                 BUS_NUMBER = 1
                 PATH = 1.2.0.0.0.0
                 DEVICE_CLASS = 0
                 DEVICE_SUB_CLASS = 0
                 DEVICE_PROTOCOL = 0
                 NUMBER_OF_INTERFACES = 1
                 CONFIGURATION_VALUE = 1
                 INTERFACE_NUMBER = 0
                 INTERFACE_PROTOCOL = 0
                 INTERFACE_CLASS = 3
                 INTERFACE_SUB_CLASS = 0
                 NUMBER_OF_CONFIGURATIONS = 1
                 MANUFACTURER_STRING = AIPTEK International Inc.
                 PRODUCT_STRING = USB Tablet Series Version 1.05
                 CONFIGURATION_NUMBER = 0
                 CURRENT_INTERFACE = 0.

15-OCT-2005 13:25:04.99 UCM          INFORMATIONAL SYS$HIDDRIVER.EXE
        Message: Loaded single instance class driver for HID0.

15-OCT-2005 13:25:05.00 DRIVER       INFORMATIONAL HIDDRIVER
        Message: Find a driver for usage page 0001 usage type 0002

15-OCT-2005 13:25:05.27 UNKNOWN      INFORMATIONAL UCM DEVICE MOU
        Message: BUS_NUMBER = 1
                 PATH = 1.2.0.0.0.0.HID_USAGE_DATA = 65538.

UCM>
```

This display provides more information. The last section is the one that shows the device, which uses an Interface Class of 3, the class that causes the Human Interface Driver (HID) to claim it.

You might wonder how you should configure your driver (UGDRIVER). Assume that you want to handle only this device (because the generic Interface driver for this class is HID) and currently no way exists to provide user-written HID drivers.

Edit SYS$USER_CONFIG.DAT to add the following record:

```
device = "CyberTablet 12000"
name   = UG
```

```
driver = sys$ugdriver
begin_private
usb_config_type = interface
vendor_id = 2250
product_id = 16
begin_interface
interface_class = 3
interface_sub_class = 0
interface_protocol = 0
end_interface
end_private
end_device
```

This new record indicates that if a device has the vendor code of 2250, and product ID of 16, and Interface Class of 3, and Protocol and Subclass of 0, load the UGDRIVER and call the device UG.

All of these numbers came from the event information. You need to include a vendor and product code because you do ***not*** want other devices, such as a generic mouse or some other vendor's tablet, to use your driver.

You then need to reload the database for UCM. You can do this by using the RELOAD or RESTART command. The difference between the two commands is that a RESTART besides reading in new configuration data also removes any in-memory structures that might have been built by earlier device events.

In this case, you create a MOU0 (USB MOUSE) device, which means that you do not have to do anything special because MOU0, by default, is never saved as a permanent device (see the description of permanent devices). To reduce the amount of information in the event file, you also need to reset it. Then you unplug the device and plug it back in. For example:

```
$ UCM
Universal Serial Bus Configuration Manager, Version V1.0
UCM> restart
Restart UCM Server? [N]: y
Waiting for UCM Server image to exit....
Waiting for UCM Server image to restart....
%USB-S-SRVRRESTART, Identification of new UCM Server is 0000021E
UCM> set log/new
UCM> show event
Date        Time        Type         Priority Component
--------------------------------------------------------------------------------
15-OCT-2005 13:47:13.58 DECONFIGURED NORMAL   HUBDRIVER
        Message: Deconfiguring device on bus 1 at port 2 bus tier 2 usb address 3

15-OCT-2005 13:47:14.76 UCM          NORMAL   SYS$UGDRIVER.EXE
        Message: Tentative device UGA0 proposed... auto-loading driver.

15-OCT-2005 13:47:14.78 UCM          NORMAL   UGA
        Message: Auto-perm converting tentative device UGA0 into permanent device.

15-OCT-2005 13:47:14.88 DRIVER       NORMAL   HUBDRIVER
        Message: Configured device UGA0 using driver SYS$UGDRIVER:

UCM>
```

The messages indicate that the device was loaded.

If you display INFORMATIONAL data, you see the following additional information:

```
UCM> show event/priority=all
Date        Time        Type         Priority Component
--------------------------------------------------------------------------------
15-OCT-2005 13:47:13.58 DECONFIGURED NORMAL   HUBDRIVER
        Message: Deconfiguring device on bus 1 at port 2 bus tier 2 usb address 3

15-OCT-2005 13:47:14.71 DRIVER       INFORMATIONAL HUBDRIVER
        Message: Find a driver for DeviceClass/DeviceSubClass = 0x0/0x0

15-OCT-2005 13:47:14.71 DRIVER       INFORMATIONAL HUBDRIVER
        Message: Find a driver for InterfaceClass/InterfaceSubClass/Protocol =
0x3/0x0/0x0

15-OCT-2005 13:47:14.76 UNKNOWN      INFORMATIONAL UCM DEVICE UGA
        Message: VENDOR_ID = 2250
                 PRODUCT_ID = 16
                 RELEASE_NUMBER = 261
                 BUS_NUMBER = 1
                 PATH = 1.2.0.0.0.0
                 DEVICE_CLASS = 0
                 DEVICE_SUB_CLASS = 0
                 DEVICE_PROTOCOL = 0
                 NUMBER_OF_INTERFACES = 1
                 CONFIGURATION_VALUE = 1
                 INTERFACE_NUMBER = 0
                 INTERFACE_PROTOCOL = 0
                 INTERFACE_CLASS = 3
                 INTERFACE_SUB_CLASS = 0
                 NUMBER_OF_CONFIGURATIONS = 1
                 MANUFACTURER_STRING = AIPTEK International Inc.
                 PRODUCT_STRING = USB Tablet Series Version 1.05
                 CONFIGURATION_NUMBER = 0
                 CURRENT_INTERFACE = 0.

15-OCT-2005 13:47:14.76 UCM          NORMAL   SYS$UGDRIVER.EXE
        Message: Tentative device UGA0 proposed... auto-loading driver.

15-OCT-2005 13:47:14.78 UCM          NORMAL   UGA
        Message: Auto-perm converting tentative device UGA0 into permanent device.

15-OCT-2005 13:47:14.88 DRIVER       NORMAL   HUBDRIVER
        Message: Configured device UGA0 using driver SYS$UGDRIVER:

UCM>
```

The important part of the device configuration is that it does not interfere with other devices with the same interface class: for example, the joystick also uses class 3, subclass 0, and protocol 0. However, if you plug in a joystick, it correctly uses the HID driver, which uses the generic match for Interface Class 3 to load the joystick driver (AGDRIVER), as shown in the following example:

```
UCM> show event/priority=all
Date        Time        Type         Priority Component
```

```
--------------------------------------------------------------------------------
15-OCT-2005 13:47:13.58 DECONFIGURED NORMAL   HUBDRIVER
        Message: Deconfiguring device on bus 1 at port 2 bus tier 2 usb address 3

15-OCT-2005 13:47:14.71 DRIVER       INFORMATIONAL HUBDRIVER
        Message: Find a driver for DeviceClass/DeviceSubClass = 0x0/0x0

15-OCT-2005 13:47:14.71 DRIVER       INFORMATIONAL HUBDRIVER
        Message: Find a driver for InterfaceClass/InterfaceSubClass/Protocol =
0x3/0x0/0x0

15-OCT-2005 13:47:14.76 UNKNOWN      INFORMATIONAL UCM DEVICE UGA
        Message: VENDOR_ID = 2250
                 PRODUCT_ID = 16
                 RELEASE_NUMBER = 261
                 BUS_NUMBER = 1
                 PATH = 1.2.0.0.0.0
                 DEVICE_CLASS = 0
                 DEVICE_SUB_CLASS = 0
                 DEVICE_PROTOCOL = 0
                 NUMBER_OF_INTERFACES = 1
                 CONFIGURATION_VALUE = 1
                 INTERFACE_NUMBER = 0
                 INTERFACE_PROTOCOL = 0
                 INTERFACE_CLASS = 3
                 INTERFACE_SUB_CLASS = 0
                 NUMBER_OF_CONFIGURATIONS = 1
                 MANUFACTURER_STRING = AIPTEK International Inc.
                 PRODUCT_STRING = USB Tablet Series Version 1.05
                 CONFIGURATION_NUMBER = 0
                 CURRENT_INTERFACE = 0.

15-OCT-2005 13:47:14.76 UCM          NORMAL   SYS$UGDRIVER.EXE
        Message: Tentative device UGA0 proposed... auto-loading driver.

15-OCT-2005 13:47:14.78 UCM          NORMAL   UGA
        Message: Auto-perm converting tentative device UGA0 into permanent device.

15-OCT-2005 13:47:14.88 DRIVER       NORMAL   HUBDRIVER
        Message: Configured device UGA0 using driver SYS$UGDRIVER:

15-OCT-2005 14:16:46.55 DECONFIGURED NORMAL   HUBDRIVER
        Message: Deconfiguring device on bus 1 at port 2 bus tier 2 usb address 3

15-OCT-2005 14:16:49.46 DRIVER       INFORMATIONAL HUBDRIVER
        Message: Find a driver for DeviceClass/DeviceSubClass = 0x0/0x0

15-OCT-2005 14:16:49.46 DRIVER       INFORMATIONAL HUBDRIVER
        Message: Find a driver for InterfaceClass/InterfaceSubClass/Protocol =
0x3/0x0/0x0

15-OCT-2005 14:16:49.49 UNKNOWN      INFORMATIONAL UCM DEVICE HID0
        Message: VENDOR_ID = 1699
                 PRODUCT_ID = 13630
                 RELEASE_NUMBER = 256
                 BUS_NUMBER = 1
                 PATH = 1.2.0.0.0.0
                 DEVICE_CLASS = 0
                 DEVICE_SUB_CLASS = 0
                 DEVICE_PROTOCOL = 0
                 NUMBER_OF_INTERFACES = 1
                 CONFIGURATION_VALUE = 1
                 INTERFACE_NUMBER = 0
                 INTERFACE_PROTOCOL = 0
                 INTERFACE_CLASS = 3
                 INTERFACE_SUB_CLASS = 0
                 NUMBER_OF_CONFIGURATIONS = 1
                 MANUFACTURER_STRING = Saitek
                 PRODUCT_STRING = Cyborg evo Wireless
                 CONFIGURATION_NUMBER = 0
                 CURRENT_INTERFACE = 0.

15-OCT-2005 14:16:49.49 UCM          INFORMATIONAL SYS$HIDDRIVER.EXE
        Message: Loaded single instance class driver for HID0.

15-OCT-2005 14:16:49.50 DRIVER       INFORMATIONAL HIDDRIVER
        Message: Find a driver for usage page 0001 usage type 0005

15-OCT-2005 14:16:49.63 UNKNOWN      INFORMATIONAL UCM DEVICE AGA
        Message: BUS_NUMBER = 1
                 PATH = 1.2.0.0.0.0.HID_USAGE_DATA = 65541.

15-OCT-2005 14:16:49.63 UCM          NORMAL   SYS$AGDRIVER.EXE
        Message: Tentative device AGA0 proposed... auto-loading driver.

15-OCT-2005 14:16:49.65 UCM          NORMAL   AGA
        Message: Auto-perm converting tentative device AGA0 into permanent device.

15-OCT-2005 14:16:49.78 DRIVER       NORMAL   HUBDRIVER
        Message: Configured device HID0 using driver SYS$AGDRIVERR:

UCM>
```

You might be puzzled about the message saying the device is tentative and will be converted into a permanent device. The following section explains this.

Permanent Devices and Tentative Devices

USB devices have OpenVMS device names assigned to them when they are configured. However, if you plug in multiple devices of the same type, in a different order or in different places, they all might have different names. Worse still, the USB bus discovery is asynchronous, and between each boot, the order of device discovery might be different.

It is not advisable for two printers, for example, to change names randomly when the system is booted.

The UCM tries to ensure that names are persistent (permanent) across boots and across hot-plugs. UCM uses two strategies to do this:

- Serial Number. If a device has a serial number, it must be unique -- at least the vendor/product code part must be.
- Path. The USB bus is a hierarchical topology. Each device can be described by the level (HUB level) and port within the HUB. A path is a six- digit value that is similar to 1.2.0.0.0.0.

When a device is configured, UCM looks in a database of PERMANENT devices to determine if this device has been seen before. If it has not, the device is configured (as described above), and the complete information about the device is stored in the permanent database, including the OpenVMS name that was used for it.

In general, the matching of devices in the permanent database is not a heuristic; it is, rather, an exact match.

The exception to this is TEMPLATE devices. Currently, there are only two – the Mouse and Keyboard. These devices have pre-allocated entries in the permanent database. A flag tells UCM that if a Mouse or Keyboard is plugged in always to create MOU0 and KBD0, no matter where they are plugged in. Mice and Keyboards do not have serial numbers, and it would not be user-friendly to create MOU1 instead of MOU0 just because someone plugged the connectors into a different USB slot. In reality, however, this dates from a time when making devices permanent and configuring and loading the OpenVMS device was a manual process.

Controlling Device Permanence and Loading

You can use the UCM commands SET AUTO and SHOW AUTO to restrict the automatic recognition of new devices. This can be useful when debugging your USB device or debugging its configuration. For example:

```
$ UCM SET AUTO/ENABLE=(LOAD)/DISABLE=(PERM)
```

This command allows the device to be loaded but does not save it in the permanent (on disk) database.

```
$UCM SET AUTO/DISABLE
```

This command disables automatic loading of the device. Instead, the device is made “Tentative” – that is, UCM knows that the device is there and what driver to load but requires the UCM command ADD DEVICE to cause it to be made permanent. In addition, the device must then be hot-swapped (unplugged and plugged back in again).

The default is `SET AUTO/ENABLE,` which enables auto-load and auto-perm. The SHOW AUTO command displays the current settings.

In addition, you can set EXCLUDE and INCLUDE lists. See the UCM section of the *HP OpenVMS System Management Utilities Manual* for more information.