USB Device Trace

NuttX USB Device Trace

Last Updated: March 20, 2011

USB Device Tracing Controls. The NuttX USB device subsystem supports a fairly sophisticated tracing facility. The basic trace cabability is controlled by these NuttX configuration settings:

- `CONFIG_USBDEV_TRACE`: Enables USB tracing
- `CONFIG_USBDEV_TRACE_NRECORDS`: Number of trace entries to remember

Trace IDs. The trace facility works like this: When enabled, USB events that occur in either the USB device driver or in the USB class driver are logged. These events are described in `include/nuttx/usb/usbdev_trace.h`. The logged events are identified by a set of event IDs:

`TRACE_INIT_ID`	Initialization events
`TRACE_EP_ID`	Endpoint API calls
`TRACE_DEV_ID`	USB device API calls
`TRACE_CLASS_ID`	USB class driver API calls
`TRACE_CLASSAPI_ID`	Other class driver system API calls
`TRACE_CLASSSTATE_ID`	Track class driver state changes
`TRACE_INTENTRY_ID`	Interrupt handler entry
`TRACE_INTDECODE_ID`	Decoded interrupt event
`TRACE_INTEXIT_ID`	Interrupt handler exit
`TRACE_OUTREQQUEUED_ID`	Request queued for OUT endpoint
`TRACE_INREQQUEUED_ID`	Request queued for IN endpoint
`TRACE_READ_ID`	Read (OUT) action
`TRACE_WRITE_ID`	Write (IN) action
`TRACE_COMPLETE_ID`	Request completed
`TRACE_DEVERROR_ID`	USB controller driver error event
`TRACE_CLSERROR_ID`	USB class driver error event

Logged Events. Each logged event is 32-bits in size and includes

1. 8-bits of the trace ID (values associated with the above)
2. 8-bits of additional trace ID data, and
3. 16-bits of additional data.

8-bit Trace Data The 8-bit trace data depends on the specific event ID. As examples,

- For the USB serial and mass storage class, the 8-bit event data is provided in `include/nuttx/usb/usbdev_trace.h`.
- For the USB device driver, that 8-bit event data is provided within the USB device driver itself. So, for example, the 8-bit event data for the LPC1768 USB device driver is found in `arch/arm/src/lpc17xx_40xx/lpc17_40_usbdev.c`.

16-bit Trace Data. The 16-bit trace data provided additional context data relevant to the specific logged event.

Trace Control Interfaces. Logging of each of these kinds events can be enabled or disabled using the interfaces described in `include/nuttx/usb/usbdev_trace.h`.

Enabling USB Device Tracing. USB device tracing will be configured if `CONFIG_USBDEV` and either of the following are set in the NuttX configuration file:

- `CONFIG_USBDEV_TRACE`, or
- `CONFIG_DEBUG_FEATURES and CONFIG_DEBUG_USB`

Log Data Sink. The logged data itself may go to either (1) an internal circular buffer, or (2) may be provided on the console. If `CONFIG_USBDEV_TRACE` is defined, then the trace data will go to the circular buffer. The size of the circular buffer is determined by `CONFIG_USBDEV_TRACE_NRECORDS`. Otherwise, the trace data goes to console.

Example. Here is an example of USB trace output using `apps/examples/usbserial` for an LPC1768 platform with the following NuttX configuration settings:

- `CONFIG_DEBUG_FEATURES`, `CONFIG_DEBUG_INFO`, `CONFIG_USB`
- `CONFIG_EXAMPLES_USBSERIAL_TRACEINIT`, `CONFIG_EXAMPLES_USBSERIAL_TRACECLASS`, `CONFIG_EXAMPLES_USBSERIAL_TRACETRANSFERS`, `CONFIG_EXAMPLES_USBSERIAL_TRACECONTROLLER`, `CONFIG_EXAMPLES_USBSERIAL_TRACEINTERRUPTS`

Console Output:

```
ABDE
```

```
 usbserial_main: Registering USB serial driver
 uart_register: Registering /dev/ttyUSB0
 usbserial_main: Successfully registered the serial driver
1Class API call 1: 0000
2Class error: 19:0000
 usbserial_main: ERROR: Failed to open /dev/ttyUSB0 for reading: 107
 usbserial_main: Not connected. Wait and try again.
3Interrupt 1 entry: 0039
4Interrupt decode 7: 0019
5Interrupt decode 32: 0019
6Interrupt decode 6: 0019
7Class disconnect(): 0000
8Device pullup(): 0001
9Interrupt 1 exit: 0000
```

The numbered items are USB USB trace output. You can look in the file `drivers/usbdev/usbdev_trprintf.c` to see examctly how each output line is formatted. Here is how each line should be interpreted:

	USB EVENT ID	8-bit EVENT DATA	MEANING	16-bit EVENT DATA
1	`TRACE_CLASSAPI_ID`[1]	1	`USBSER_TRACECLASSAPI_SETUP`[1]	0000
2	`TRACE_CLSERROR_ID`[1]	19	`USBSER_TRACEERR_SETUPNOTCONNECTED`[1]	0000
3	`TRACE_INTENTRY_ID`[1]	1	`LPC17_40_TRACEINTID_USB`[2]	0039
4	`TRACE_INTDECODE_ID`[2]	7	`LPC17_40_TRACEINTID_DEVSTAT`[2]	0019
5	`TRACE_INTDECODE_ID`[2]	32	`LPC17_40_TRACEINTID_SUSPENDCHG`[2]	0019
6	`TRACE_INTDECODE_ID`[2]	6	`LPC17_40_TRACEINTID_DEVRESET`[2]	0019
7	`TRACE_CLASS_ID`[1]	3	`(See TRACE_CLASSDISCONNECT`[1]`)`	0000
8	`TRACE_DEV_ID`[1]	6	`(See TRACE_DEVPULLUP`[1]`)`	0001
9	`TRACE_INTEXIT_ID`[1]	1	`LPC17_40_TRACEINTID_USB`[2]	0000

NOTES:
[1]See `include/nuttx/usb/usbdev_trace.h`
[2]See `arch/arm/src/lpc17xx_40xx/lpc17_40_usbdev.c`

In the above example you can see that:

- **1**. The serial class USB setup method was called for the USB serial class. This is the corresponds to the following logic in `drivers/usbdev/pl2303.c`:

```
static int pl2303_setup(FAR struct uart_dev_s *dev)
{
  ...
  usbtrace(PL2303_CLASSAPI_SETUP, 0);
  ...
```

- **2**. An error occurred while processing the setup command because no configuration has yet been selected by the host. This corresponds to the following logic in `drivers/usbdev/pl2303.c`:

```
static int pl2303_setup(FAR struct uart_dev_s *dev)
{
  ...
  /* Check if we have been configured */

  if (priv->config == PL2303_CONFIGIDNONE)
    {
      usbtrace(TRACE_CLSERROR(USBSER_TRACEERR_SETUPNOTCONNECTED), 0);
      return -ENOTCONN;
    }
  ...
```

- **3-6**. Here is a USB interrupt that suspends and resets the device.
- **7-8**. During the interrupt processing the serial class is disconnected
- **9**. And the interrupt returns

USB Monitor. The *USB monitor* is an application in the `apps/system/usbmonitor` that provides a convenient way to get debug trace output. If tracing is enabled, the USB device will save encoded trace output in in-memory buffer; if the USB monitor is also enabled, that trace buffer will be periodically emptied and dumped to the system logging device (the serial console in most configurations). The following are some of the relevant configuration options:

Device Drivers -> USB Device Driver Support	
`CONFIG_USBDEV_TRACE=y`	Enable USB trace feature
`CONFIG_USBDEV_TRACE_NRECORDS=`*nnnn*	Buffer *nnnn* records in memory. If you lose trace data, then you will need to increase the size of this buffer (or increase the rate at which the trace buffer is emptied).

`CONFIG_USBDEV_TRACE_STRINGS=y`	Optionally, convert trace ID numbers to strings. This feature may not be supported by all drivers.
Application Configuration -> NSH LIbrary	
`CONFIG_NSH_USBDEV_TRACE=n`	Make sure that any built-in tracing from NSH is disabled.
`CONFIG_NSH_ARCHINIT=y`	Enable this option *only* if your board-specific logic has logic to automatically start the USB monitor. Otherwise the USB monitor can be started or stopped with the `usbmon_start` and `usbmon_stop` commands from the NSH console.
Application Configuration -> System NSH Add-Ons	
`CONFIG_USBMONITOR=y`	Enable the USB monitor daemon
`CONFIG_USBMONITOR_STACKSIZE=`*nnnn*	Sets the USB monitor daemon stack size to *nnnn*. The default is 2KiB.
`CONFIG_USBMONITOR_PRIORITY=50`	Sets the USB monitor daemon priority to *nnnn*. This priority should be low so that it does not interfere with other operations, but not so low that you cannot dump the buffered USB data sufficiently rapidly. The default is 50.
`CONFIG_USBMONITOR_INTERVAL=`*nnnn*	Dump the buffered USB data every *nnnn* seconds. If you lose buffered USB trace data, then dropping this value will help by increasing the rate at which the USB trace buffer is emptied.

`CONFIG_USBMONITOR_TRACEINIT=y`
`CONFIG_USBMONITOR_TRACECLASS=y`
`CONFIG_USBMONITOR_TRACETRANSFERS=y`
`CONFIG_USBMONITOR_TRACECONTROLLER=y`
`CONFIG_USBMONITOR_TRACEINTERRUPTS=y`
	Selects which USB event(s) that you want to be traced.

NOTE: If USB debug output is also enabled, both outputs will appear on the serial console. However, the debug output will be asynchronous with the trace output and, hence, difficult to interpret.