

User-Space Driver Development Kit

Reference Manual

User-Space Driver Development Kit Reference Manual

Table of Contents

- 1. User-Space Driver Development Kit 1
 - USDDK 1
 - Tutorial 13

Chapter 1. User-Space Driver Development Kit

USDDK

Name

USDDK — User-Space Driver Development Kit

Synopsis

```
struct      Usddk;
struct      UsddkDevice;
Usddk*      usddk_new                                (void);
int         usddk_init                              (Usddk *usddk);
void        usddk_free                              (Usddk *usddk);
int         usddk_fileno                            (Usddk *usddk);
int         usddk_get_version                      (Usddk *usddk);
int         usddk_get_n_pci_devices               (Usddk *usddk);
UsddkDevice* usddk_get_pci_device                 (Usddk *usddk,
int index);
int         usddk_pci_select_by_id                 (Usddk *usddk,
int vendor,
int device);
void*       usddk_contiguous_map                   (Usddk *usddk);
unsigned long usddk_contiguous_get_addr            (Usddk *usddk);
int         usddk_contiguous_alloc                 (Usddk *usddk,
unsigned long size);
char*       usddk_debug_get_last_message           (Usddk *usddk);
int         usddk_debug_set_level                  (Usddk *usddk,
int level);
int         usddk_device_attach                   (Usddk *usddk,
int index);
void*       usddk_device_map                       (Usddk *usddk);
int         usddk_dma_alloc                        (Usddk *usddk,
unsigned long size);
void*       usddk_dma_map                         (Usddk *usddk);
unsigned long* usddk_dma_get_page_addrs           (Usddk *usddk);
int         usddk_irq_request                      (Usddk *usddk,
int irq);
int         usddk_irq_free                        (Usddk *usddk);
int         usddk_irq_wait                        (Usddk *usddk);
```

Description

The USDDK is a library of functions allowing an application to directly access PCI devices in Linux as well as making it easy to use DMA to transfer to or from the device.

The USDDK library uses the `pkg-config` utility to describe the correct compiler flags to compile and link with it. The following commands output the compiler flags and linking flags:

```
pkg-config --cflags usddk-0.1
pkg-config --libs usddk-0.1
```

The `usddk_irq_*` functions are related to interrupts.

The `usddk_dma_*` functions are related to an area that can be allocated using the kernel function `vmalloc()`. The pages making up this area are not physically contiguous, but when mapped to user space, are contiguous in the process address space.

The `usddk_contiguous_*` functions are related to an area of memory that can be allocated using the kernel function `kmalloc()`. This memory area is physically contiguous, and when mapped to user space, is contiguous in the process address space.

The `usddk_debug_*` functions are related to the USDDK kernel module's internal debugging system. When enabled, debug messages are outputted using `printk()`, thus appear in the kernel log and/or to the console, depending on how `syslog` is configured on your system. Retrieving the last message is independent of the log level.

Details

struct Usddk

```
struct Usddk {
};
```

This structure is opaque to the application.

struct UsddkDevice

```
struct UsddkDevice {
    int index;
```

```

int vendor;
int device;
int subsystem_vendor;
int subsystem_device;
int pci_class;
int rev;

int bus;
int slot;
int function;

int irq;
struct {
    int start;
    int size;
    int flags;
} resources[USDDK_RESOURCE_COUNT];
};

```

This structure represents several identifying properties of a device.

int index

The index of the device.

int vendor

PCI vendor ID

int device

PCI device ID

int subsystem_vendor

PCI subsystem vendor ID

int subsystem_device

PCI subsystem device ID

int pci_class

PCI device class

int rev

PCI revision

int bus

bus number

int *slot*

slot number

int *function*

function number

int *irq*

IRQ used by the device

usddk_new ()

```
Usddk*      usddk_new                (void);
```

Creates a new Usddk object. After creation, the object is uninitialized; one must call `usddk_init()` to initialize the object.

Implementation: allocates and clears space for the object.

Returns :

a pointer to a new Usddk object

usddk_init ()

```
int          usddk_init              (Usddk *usddk);
```

Initializes a Usddk object.

Implementation: opens the device `/dev/usddk`.

usddk :

the driver object to initialize

Returns :

-1 on failure, ≥ 0 on success

usddk_free ()

```
void         usddk_free              (Usddk *usddk);
```

Frees a Usddk object and any associated resources.

Implementation: frees allocated memory and closes device.

usddk :

the driver object to free

usddk_fileno ()

```
int          usddk_fileno          (Usddk *usddk);
```

Returns the file descriptor used by the object to talk to the kernel driver. This function generally does not need to be used by applications, since most functionality of the driver is handled by the library.

Implementation: returns a field in *usddk*.

usddk :

the driver object

Returns :

the file descriptor used to talk to the kernel driver, or -1 if there is an error.

usddk_get_version ()

```
int          usddk_get_version     (Usddk *usddk);
```

Returns the kernel interface version used by the kernel driver. Currently this is under-implemented. FIXME.

Implementation: Calls the USDDK_GET_VERSION ioctl and returns the result.

usddk :

the driver object

Returns :

the kernel interface version used by the kernel driver, or -1 if there is an error.

usddk_get_n_pci_devices ()

```
int          usddk_get_n_pci_devices          (Usddk *usddk);
```

Gets the number of PCI devices.

Implementation: calls the USDDK_GET_N_PCI_DEVICES ioctl and returns the result.

usddk :

the driver object

Returns :

the number of PCI devices or -1 if there is an error.

usddk_get_pci_device ()

```
UsddkDevice* usddk_get_pci_device          (Usddk *usddk,
                                           int index);
```

Gets a structure containing information about a particular PCI device. The PCI devices are numbered 0 .. DEVS-1, where DEVS is the return value of `usddk_get_n_pci_devices()`. The returned pointer should be freed with `free()` when it is no longer needed.

Implementation: calls the USDDK_GET_PCI_DEVICE ioctl to retrieve information, and copies it into a newly allocated structure.

usddk :

the driver object

index :

the index of the PCI device

Returns :

a pointer to a UsddkDevice structure, or NULL if there is an error.

usddk_pci_select_by_id ()

```
int          usddk_pci_select_by_id       (Usddk *usddk,
                                           int vendor,
                                           int device);
```

Searches the list of PCI devices for a device that matches the vendor ID and device ID. Note that this only returns the first matching device.

Implementation: calls `usddk_get_pci_device()` for each index, and returns the first index that matches the vendor and device ID.

usddk :

the driver object

vendor :

the PCI vendor ID

device :

the PCI device ID

Returns :

index of matching device, or -1 if there is no match.

usddk_contiguous_map ()

```
void*      usddk_contiguous_map      (Usddk *usddk);
```

Maps the region of memory allocated by `usddk_contiguous_alloc()` into the address space of the process. If the region is already mapped, the old pointer is returned. The region will be automatically unmapped when the *usddk* object is freed. Do not unmap the region directly using `munmap()`.

Implementation: calls `mmap()` with a special offset to indicate that the contiguous area is being mapped and returns the result.

usddk :

the driver object

usddk_contiguous_get_addr ()

```
unsigned long usddk_contiguous_get_addr      (Usddk *usddk);
```

Gets the physical address for the memory region allocated by `usddk_contiguous_alloc()`.

Implementation: calls the `USDDK_GET_CONSISTENT_ADDR ioctl()` and returns the result.

usddk :

the driver object

Returns :

a physical address (expressed as an unsigned long), or 0 if there is an error.

usddk_contiguous_alloc ()

```
int          usddk_contiguous_alloc          (Usddk *usddk,
                                             unsigned long size);
```

Allocates a region of memory for the purpose of DMA to or from the PCI device. This memory is physically contiguous, and because of the way that the kernel allocates memory, only a limited amount of contiguous memory can be allocated. The hard limit is 128 kB for Linux-2.4 and Linux-2.6, although memory fragmentation can cause the effective limit to be much lower.

size must be a power of 2 and multiple of the page size. The page size can be determined using `getpagesize()`, and is 4096 on i386 systems.

Warning: This function can commonly fail for large sizes (64 kB or 128 kB).

Implementation: calls the `USDDK_ALLOC_CONSISTENT` ioctl and returns the result.

usddk :

the driver object

Param2 :

Returns :

≥ 0 if successful, -1 if there is an error.

usddk_debug_get_last_message ()

```
char*       usddk_debug_get_last_message   (Usddk *usddk);
```

Gets the last debug message generated by the kernel driver and returns it. The returned string should be freed using `free()` when it is no longer needed.

Implementation: calls the `USDDK_GET_LAST_DEBUG` ioctl and returns the result.

usddk :

the driver object

Returns :

a pointer to a string of characters, or NULL if there was an error.

usddk_debug_set_level ()

```
int          usddk_debug_set_level          (Usddk *usddk,  
                                             int level);
```

Sets the kernel driver debug level.

Implementation: calls the USDDK_SET_DEBUG_LEVEL ioctl and returns the result.

usddk :

the driver object

level :

the new debug level

Returns :

the previous debug level, or -1 if there is an error.

usddk_device_attach ()

```
int          usddk_device_attach          (Usddk *usddk,  
                                             int index);
```

Allocates kernel resources associated with the PCI device with the given index and links *usddk* to that device. This is a prerequisite to using the PCI device.

Implementation: calls the USDDK_ATTACH ioctl and returns the result.

usddk :

the driver object

index :

the index of the device to attach

Returns :

≥ 0 on success, -1 if there is an error.

usddk_device_map ()

```
void*          usddk_device_map          (Usddk *usddk);
```

Maps the PCI device's IO memory region into the memory address space of the current process, so that it can be accessed directly. This map will be automatically deleted when *usddk* is destroyed. Do not unmap the region directly using `munmap()`.

Implementation: calls `mmap()` with a special offset to map the correct region.

usddk :

the driver object

usddk_dma_alloc ()

```
int          usddk_dma_alloc          (Usddk *usddk,
                                     unsigned long size);
```

Allocates a region of memory for the purpose of DMA to or from the PCI device. This memory is NOT physically contiguous, so each page has a different physical address. DMA uses the physical address, so this must be taken into account when programming the DMA controller.

size must be a multiple of the page size. The page size can be determined using `getpagesize()`, and is 4096 on i386 systems. The maximum amount that can be allocated is limited mainly on the size of the page list. The page list can be a maximum of 128 kB, which can store 32768 pointers to pages, giving a maximum size of 128 MB.

Implementation: calls the `USDDK_ALLOC` ioctl and returns the result.

usddk :

the driver object

Param2 :

Returns :

≥ 0 if successful, -1 if there is an error.

usddk_dma_map ()

```
void*      usddk_dma_map          (Usddk *usddk);
```

Maps the region of memory allocated by `usddk_dma_alloc()` into the address space of the process. If the region is already mapped, the old pointer is returned. The region will be automatically unmapped when the `usddk` object is freed. Do not unmap the region directly using `munmap()`.

Implementation: calls `mmap()` with a special offset to indicate that the DMA area is being mapped and returns the result.

usddk :

the driver object

usddk_dma_get_page_addrs ()

```
unsigned long* usddk_dma_get_page_addrs    (Usddk *usddk);
```

Gets an array of physical addresses for the pages in the DMA region allocated by `usddk_dma_alloc()`. The length of the array is the number of pages allocated. The list should be freed using `free()` when it is no longer needed.

Implementation: calls the `USDDK_GET_PAGE_ADDRS ioctl()` and returns the result.

usddk :

the driver object

Returns :

a pointer to an array of physical addresses (expressed as unsigned longs), or NULL if there is an error.

usddk_irq_request ()

```
int      usddk_irq_request          (Usddk *usddk,
                                     int irq);
```

Allocates the given IRQ and attaches a handler that puts a dummy 0 byte into a buffer for each interrupt handled. This buffer is read by `read()` function call, or more easily, by `usddk_irq_wait()`. The IRQ is automatically freed when `usddk` is freed.

The interrupt handler disables interrupts on the given IRQ when an interrupt arrives. The `read()` handler reenables interrupts on the IRQ if there is no interrupt pending.

Implementation: calls the `USDDK_REQUEST_IRQ` ioctl and returns the result.

usddk :

the driver object

irq :

the IRQ to request

Returns :

>=0 if successful, -1 if there is an error.

usddk_irq_free ()

```
int          usddk_irq_free          (Usddk *usddk);
```

Frees the IRQ previously allocated for *usddk*.

Implementation: calls the `USDDK_FREE_IRQ` ioctl and returns the result.

usddk :

the driver object

Returns :

>=0 if successful, -1 if there is an error

usddk_irq_wait ()

```
int          usddk_irq_wait         (Usddk *usddk);
```

If there is pending interrupt on the IRQ associated with *usddk*, `usddk_irq_wait()` will return immediately. Otherwise, `usddk_irq_wait()` waits for the next interrupt.

Implementation: calls `read()` to read one byte from the device.

usddk :

the driver object

Returns :

>=0 if successful, -1 if there is an error.

Tutorial

Name

Tutorial — User-space driver tutorial.

A simple example

This example demonstrates a few things that every application that uses `usddk` will need. This example finds a particular PCI device and connects to it.

```
#include <usddk/usddk.h>
#include <stdio.h>
#include <stdlib.h>

#define VENDOR_ID 0x8086
#define DEVICE_ID 0x1010

int
main (int argc, char *argv[])
{
    Usddk *usddk;
    int ret;
    int i;

    usddk = usddk_new ();
    if (usddk == NULL) exit (1);

    ret = usddk_init (usddk);
    if (ret == -1) exit (1);

    i = usddk_pci_select_by_id (usddk, VENDOR_ID, DEVICE_ID);
    if (i == -1) {
```

```

    printf("could not find %04x:%04x\n", VENDOR_ID, DEVICE_ID);
    exit (1);
}

ret = usddk_device_attach (usddk, i);
if (ret == -1) exit(1);

/* do something interesting here */

return 0;
}

```

The steps performed by this example are: creating an object, initializing the object, selecting a device based on its vendor and device ids, and then attaching to the device. Attaching to the device tells the library and the kernel that the application intends to use that device, and no other. The steps of creation and initialization of the object are separate, since the latter can fail, as the process of opening the device may fail. The object contains error messages, so in this way, it is possible to get error messages for failure to open a device.

A driver object can connect to one PCI device, one IRQ, and one region of allocated kernel memory (the DMA region), or any combination of the three. Since PCI devices can have multiple I/O memory regions, any of the regions can be mapped. I/O ports cannot be accessed through the usddk library, but one can obtain information about the ports, and access them directly after calling `iopl()` to allow permission.

Note that you can create any number of Usddk objects in order to access any number of PCI devices. These objects will all operate independently.

Another simple example

This example demonstrates how to allocate and use an interrupt.

see `testsute/rtc/rtclock.c`

The real-time clock chip in all PCs is capable of producing periodic interrupts on IRQ 8. This isn't a PCI device, so the program doesn't need to go through the steps of attaching to a device. However, the driver object needs to be created and initialized. There is also a number of things that need to be done to configure the real-time clock to produce periodic interrupts. Note that since the real-time clock is accessed through port I/O, `ioperm()` is called to allow access to the ports -- this requires root privileges.

The interesting part of the program is the call to `usddk_request_irq()`, which requests and enables the interrupt. Subsequent calls to `usddk_irq_wait()` will cause the process to wait until an interrupt arrives on the requested IRQ. The IRQ is automatically freed when the program exits, when the driver object is freed, or when `usddk_free()` is called.