USB VIDEO CLASS DRIVER

Step1: Download the source code from www.kernel.org

Step2: Compile the source code

make menuconfig

make -j128 or -j64 or -j32 or -j16 or -j4

Error:-`fatal error: openssl/opensslv.h: No such file or directory`

```
sudo apt-get install libssl-dev
```

make modules_install
make install
update-grub

Warning: Setting GRUB_TIMEOUT to a non-zero value when GRUB_HIDDEN_TIMEOUT is set is no longer supported.

`Goto this directry cat /etc/default/grub`

GRUB_HIDDEN_TIMEOUT=0 ===> comment this line S

Step3 : add the DEBUGG messages in each file in the UVC driver folder.

- uvc_video.c
- uvc_driver.c and some othre .c files

=> printk("\n%s: called from this LINE:%d\n", __func__,__LINE__);
=> printk("Caller is %pS\n", __builtin_return_address(0));
=> printk("Caller is %pS\n", __builtin_return_address(1));
=> printk("Caller is %pS\n", __builtin_return_address(2));

Step4: find the Entry point of the UVC driver from V4L2.

- uvc_init_video_isoc

URB ? It is buffer it contains the video data
the URB filled in the uvc_video_complete handlear .

=> We have limited number of URB, so that we need resubmit the URB for getting data.

Step5: get the data from uvc_video_complete , and store it into a file in the kenenl space using the vfs_write.

file: sample.yuv

OR

kernel_write is used for writing the video data into a file in the kernel space .

This sample.yuv file contains video data.
It is in MJPG formate :
- MJPG header file contains the 12 bytes of data , this 12 bytes of data we don't need skip and copy the remaining data into the sample.yuv
- Each Frame START with ff d8 and END with ff d9 .

If file contains No.of frames : the data like this ff d8 --- - - - - - ff d9 ff d8 ------ - - - - ff d9

[For file writing i use the kthreads , spinlocks and wait Queues]

**Function caller in linux kernel
play:-
[gst-launch-1.0 filesrc location=sample.yuv ! image/mjpg, width=1920,hight=1080, fromerate=30/1 ! autovideosink]

You can get the caller with __builtin_return_address(0).
The caller's caller is __builtin_return_address(1) and so on.
It's a GCC extension, documented in the gcc manual: http://gcc.gnu.org/onlinedocs/gcc/Return-Address.html

Edit: I should probably point out, that gets you the address of the caller. If you want the function name you can print it with %pS, eg: printk("Caller is %pS\n", __builtin_return_address(0));

If you don't want to print it, you can use kallsyms_lookup() etc.

=====>LERNING THING FOR DRIVER IMPLIMENTATION <=====
https://sysplay.in/blog/linux-kernel-internals/ i read this link and LLD3 book for Wait Queues

1.Wait Queues:-

Wait queue is a mechanism provided in kernel to implement the wait. wait queue is the list of processes waiting for an event.

```
#include <linux/wait.h>
// Data structure: wait_queue_head_t
// Created statically
DECLARE_WAIT_QUEUE_HEAD(wait_queue_name);
// Created dynamically
wait_queue_head_t my_queue;
init_waitqueue_head(&my_queue);
```

Once the wait queue is initialized, next step is to add our process to wait queue.

```
// APIs for Waiting
=> wait_event(queue, condition);
=> wait_event_interruptible(queue, condition);
=> wait_event_timeout(queue, condition, timeout);
=> wait_event_interruptible_timeout(queue, condition, timeout);
```

There are two variants –*wait_event()* and *wait_event_timeout()*. The former is used for waiting for an event as usual, but the latter can be used to wait for an event with timeout.
if the requirement is to wait for an event till 5 milliseconds, after which we need to timeout.

Wake up the waiting events :- *wake_up()* family of APIs

```
// Wakes up all the processes waiting on the queue
=> wake_up(wake_queue_head_t *);
// Wakes up only the processes performing the interruptible sleep
=> wake_up_interruptible(wait_queue_head_t *);
```

http://www.linuxjournal.com/node/8144/print For Kernel threads i gothrough this link.

File pointer:

Fixing the Address Space:-

To handle this address space mismatch, use the functions get_fs() and set_fs(). These functions modify the current process address limits to whatever the caller wants.

`set_fs(KERNEL_DS);`

The only two valid options for the set_fs() function are KERNEL_DS and USER_DS, roughly standing for kernel data segment and user data segment, respectively.

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/syscalls.h>
#include <linux/file.h>
#include <linux/fs.h>
#include <linux/fcntl.h>
#include <asm/uaccess.h>

static void write_file(char *filename, char *data)
{
  struct file *file;
  loff_t pos = 0;
  int fd;

  mm_segment_t old_fs = get_fs();
  set_fs(KERNEL_DS);

  fd = sys_open(filename, O_WRONLY|O_CREAT, 0644);
  if (fd >= 0) {
    sys_write(fd, data, strlen(data));
    file = fget(fd);
    if (file) {
      vfs_write(file, data, strlen(data), &pos);
      fput(file);
    }
    sys_close(fd);
  }
  set_fs(old_fs);
}

static int __init init(void)
{
  write_file("/tmp/test", "Evil file.\n");
  return 0;
}

static void __exit exit(void)
{ }

MODULE_LICENSE("GPL");
module_init(init);
module_exit(exit);
```

APIs for creating the Kernel thread

```
#include <kthread.h>
kthread_create(int (*function)(void *data), void *data, const char name[], ...)
```

Parameters:

function – The function that the thread has to execute

data – The 'data' to be passed to the function

name – The name by which the process will be recognized in the kernel

Retuns: Pointer to a structure of type task_struct

when we create the thread with *kthread_create()*, it creates the thread in sleep state and thus nothing is executed. So, how do we wake up the thread. We have a API *wake_up_process()*

```
// Module Initialization
static struct task_struct *thread_st;
{
    printk(KERN_INFO "Creating Thread\n");
    //Create the kernel thread with name 'mythread'
    thread_st = kthread_create(thread_fn, NULL, "mythread");
    if (thread_st)
    {
        printk("Thread Created successfully\n");
       wake_up_process(thread_st);
    }
    else
        printk(KERN_INFO "Thread creation failed\n");
    return 0;
}
```

As you might notice, *wake_up_process()* takes pointer to *task_struct* as an argument, which in turn is returned from *kthread_create()*.

As seen, running a thread is a two step process – First create a thread and wake it up using *wake_up_process()*. However, kernel provides an API, which **performs both these steps in one go** as

```
#include <kthread.h>
kthread_run(int (*function)(void *data), void *data, const char name[], ...)
```

Parameters:

function – The function that the thread has to execute

data – The 'data' to be passed to the function

name – The name by which the process will be recognized in the kernel

Returns: Pointer to a structure of type task_struct

So, just replace the *kthread_create()* and *wake_up_process()* calls in above code with kthread_run and you will notice that thread starts running immediately.

Stopping the Kernel Thread

```
#include <linux/kthread.h>
int kthread_stop(struct task_struct *k);
```

Parameters:

k – pointer to the task structure of the thread to be stopped

Returns: The result of the function executed by the thread, -EINTR, if *wake_up_process()* was never called.

Below is the code snippet which uses *kthread_stop()*:

From 12-Feb-2018 to 09-March-2018 I have try this many methods
- I am using struture pointer for getting the video data,
- I am using single pointer to capture the video data.