Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ *Информатика и системы управления*

КАФЕДРА *Программное обеспечение ЭВМ и информационные технологии*

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:

Драйвер для использования графического планшета Wacom Bamboo CTH-670 в качестве клавиатуры

Студент ИУ7-71Б _____ А.Д. Сушина
(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель курсовой работы _____ **М**.В. Филиппов
(Подпись, дата) (И.О.Фамилия)

Консультант _____ _____
(Подпись, дата) ***(И.О.Фамилия)***

2021 г

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

УТВЕРЖДАЮ
Заведующий кафедрой ИУ7
(Индекс)
_____ И.В.Рудаков
(И.О.Фамилия)
« _____ » _____ 20 г.

З А Д А Н И Е
на выполнение курсового проекта

по дисциплине Операционные системы

Студент группы ИУ7-61б

Сушина Анастасия Дмитриевна
(Фамилия, имя, отчество)

Тема курсового проекта Драйвер для использования графического планшета Wacom Bamboo CTH-670 в качестве клавиатуры.
Направленность КП (учебный, исследовательский, практический, производственный, др.)
Учебный
Источник тематики (кафедра, предприятие, НИР) кафедра

График выполнения проекта: 25% к 4 нед., 50% к 7 нед., 75% к 11 нед., 100% к 14 нед.

Задание
Разработать драйвер для устройства графический планшет Wacom Bamboo CTH-670, который позволит использовать устройство в качестве клавиатуры. Драйвер разработать для операционный системы Linux.
Оформление курсового проекта:

Расчетно-пояснительная записка на 25-30 листах формата А4.
Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.)
На защиту проекта должна быть представлена презентация, состоящая из 10-15 слайдов. На слайдах должны быть отражены: постановка задачи, использованные методы и алгоритмы, расчетные соотношения, структура комплекса программ, интерфейс, характеристики разработанного ПО.

Дата выдачи задания « ___ » _____ 20__ г.

Руководитель курсового проекта _____ Филиппов М.В.
(Подпись, дата) (И.О.Фамилия)
Студент _____ Сушина А.Д.
(Подпись, дата) (И.О.Фамилия)

Оглавление

Введение

В современном мире люди каждый день используют компьютеры и телефоны для обмена информацией. Чтобы была возможность производить поиск информации и набирать ее необходима клавиатура.

Клавиатура — это устройство, позволяющее пользователю вводить информацию, т. е. устройство ввода. Устройство представляет собой набор клавиш, расположенных в определенном порядке.

Кроме физических устройств существуют экранные клавиатуры, где клавиши — это определенные части экрана, на которые можно нажать с помощью мыши или пальца. Однако зачастую этот способ набора информации неудобен: мышь не позволяет нажимать на клавиши достаточно быстро, а на устройствах с сенсорным экраном могут быть слишком маленькие кнопки, по которым не всегда можно попасть пальцем.

В качестве клавиатуры может работать, например, графический планшет, если разделить его на зоны, каждая из которых будет отвечать за определенную кнопку. Однако, устройство не может работать так само по себе. Необходимо написать драйвер, который позволит операционной системе воспринимать сигналы устройства так, как этого хочет пользователь.

Использование устройства таким образом может быть полезно, если у пользователя нет компьютерной клавиатуры или ему неудобно использовать встроенную экранную клавиатуру планшетов. Такое использование планшета может повысить скорость набора информации: можно нажимать на клавиши сразу несколькими пальцами, а кнопки достаточно большие, чтобы не было возможности промахнуться.

Целью данной работы является разработка драйвера графического планшета, который позволит использовать планшет в качестве клавиатуры. Драйвер должен быть реализован для ОС Linux.

Для достижения этой цели были поставлены следующие задачи:

1. Изучить, каким образом реализуются драйвера ОС Linux.
2. Опередить, к какому типу устройств относится графический планшет, каким образом драйвер сможет получать информацию о нажатии на графическим планшет.
3. Изучить способы имитации работы клавиатуры.
4. Разработать драйвер для графического планшета.

1 Аналитический раздел

В аналитическом разделе будет выполнена постановка задачи, а также приведены методы решения задачи.

1.1 Постановка задачи

Необходимо разработать драйвер для графического планшета Wacom Bamboo CTH-670, который позволит использовать планшет в качестве клавиатуры. Драйвер должен быть разработан для операционной системы Linux.

Экран графического планшета должен быть разделен на зоны, каждая из которых будет отвечать за свою кнопку. При нажатии на планшет, драйвер должен определить, на какую зону было совершено нажатие и сообщить операционной системе, какая кнопка была нажата.

1.2 Графический планшет

Чтобы понимать, каким образом должен быть реализован драйвер, необходимо сначала определить, для какого устройства он будет разработан и как это устройство работает.

Графический планшет - это устройство для ввода информации, созданной от руки, непосредственно в компьютер.

Планшет Wacom Bamboo CTH-670 состоит из стилуса и плоского планшета с сенсорным экраном и несколькими кнопками в левой части. Данный планшет реагирует на нажатие на сенсорный экран пальцем или пером, а также на близость пера. К компьютеру планшет подключается с помощью usb. Устройство представлено на рисунке 1.1.

Рис. 1.1 Графический планшет Wacom Bamboo CTH-670

1.3 Драйвера в ОС Linux

Драйвер — это программное обеспечение, с помощью которого операционная система получает доступ к аппаратному обеспечению некоторого устройства. В Linux драйвера — это «черные ящики», которые заставляют специфичную часть оборудования соответствовать строго заданному программному интерфейсу; они полностью скрывают детали того, как работает устройство [1]. Этот программный интерфейс таков, что драйверы могут быть собраны отдельно от остальной части ядра и подключены в процессе работы, когда это необходимо.

Драйвера устройств необходимы в операционной системе, так как каждое отдельное устройство воспринимает только свой строго фиксированный набор специализированных команд, с помощью которых им можно управлять. Приложения обычно используют команды высокого уровня, предоставляемого ОС, а преобразовывает эти команды в управляющие последовательности для конкретного устройства драйвер этого устройства. Поэтому каждое отдельное устройство должно иметь свой программный драйвер, который выполняет роль связующего звена между аппаратной частью устройства и программными приложениями, использующими это устройство.

В Linux драйверы устройств бывают трех типов:

- Драйверы первого типа являются частью программного кода ядра. Соответствующие устройства автоматически обнаруживаются системой и становятся доступны доя приложений.
- Драйверы второго типа представлены загружаемыми модулями ядра. Они оформлены в виде отдельных файлов. Для их подключения необходимо выполнить команду подключения модуля. Если необходимость в использовании отпала, модуль можно выгрузить из памяти. Использование модулей обеспечивает большую гибкость, так как каждый драйвер может быть переконфигурирован без остановки системы.
- В драйверах третьего типа программный код драйвера поделен между ядром и специальной утилитой, предназначенной для управления данным устройством.

Во всех драйверах взаимодействие с устройством осуществляет ядро или модуль ядра, а пользовательские программы взаимодействуют через специальные файлы, расположенные в каталоге /dev и его подкаталогах.

Драйвер графического планшета может быть реализован как драйвер второго типа. Так, у пользователя будет возможность подключить и отключить драйвер в любой момент.

1.4 Загружаемые модули ядра

Загружаемые модули ядра Linux позволяют добавлять функциональность в ядро, когда система запущена и работает [1].

Часть кода, которая может быть добавлена в ядро во время работы, называется модулем. Ядро Linux предлагает поддержку довольно большого числа типов модулей, включая драйвера устройств. Каждый модуль является подготовленным объектным кодом, который может быть динамически подключен в работающее ядро программой insmod и отключен программой rmmod.

Каждый модуль обычно классифицируется как символьный модуль, блочный модуль или сетевой модуль. Однако существуют и другие пути классификации

модулей-драйверов, которые по-другому подразделяют устройства. Так, можно говорить о модулях универсальной последовательной шины (USB), последовательных модулях, модулях SCSI и других.

Каждое USB-устройство управляется модулем USB, который работает с подсистемой USB, но само устройство представлено в системе или как символьное устройство, или как блочное устройство, или как сетевой интерфейс.

1.5 Драйвера USB-устройств

Universal Serial Bus (USB, Универсальная Последовательная Шина) является соединением между компьютером и несколькими периферийными устройствами.

Ядро Linux поддерживает два основных типа драйверов USB:

- Драйверы на хост-системе. Они управляют USB-устройствами, которые к ней подключены, с точки зрения хоста (обычно хостом USB является персональный компьютер.)
- Драйверы на устройстве. Они контролируют, как одно устройство видит хост-компьютер в качестве устройства USB.

В ОС Linux USB устройства описываются структурой usb_device.

1.5.1 Основы USB устройства

Ядро Linux предоставляет подсистему, называемую ядром USB. Ядро USB предоставляет интерфейс для драйверов USB, используемый для доступа и управления USB оборудованием, без необходимости беспокоится о различных типах аппаратных контроллеров USB, которые присутствуют в системе.

USB - устройства всегда отвечают на запросы host-компьютера, но они никогда не могут посылать информацию самостоятельно. Передача данных выполняется между буфером в памяти хост компьютера и конечной точкой универсальной последовательной шины USB устройства. Перед передачей данные организуются в пакеты.

В составе USB-функции, то есть в устройстве с интерфейсом USB, имеется периферийный контроллер USB. Этот контроллер имеет две основные функции: он взаимодействует с USB-системой и содержит в себе буферы в количестве от одного до шестнадцати, называемые конечными точками. Конечная точка USB может переносить данные только в одном направлении, либо со стороны хост-компьютера на устройство (OUT) или от устройства на хост-компьютер (IN).

Конечная точка USB может быть одной из четырёх типов, которые описывают, каким образом передаются данные:

1. Control. Передача типа control является двунаправленной и предназначена для обмена с устройством короткими пакетами типа «вопрос-ответ». Обеспечивает гарантированную доставку данных.
2. Isochronous. Изохронный канал имеет гарантированную пропускную способность (N пакетов за один период шины) и обеспечивает непрерывную передачу данных. Передача осуществляется без подтверждения приема.
3. Interrupt. Канал прерывания позволяет доставлять короткие пакеты без гарантии доставки и без подтверждений приема, но с гарантией времени доставки – пакет будет доставлен не позже, чем через N миллисекунд.
4. Bulk. Поточные оконечные точки передают большие объёмы данных. Этим передачам протокол USB не гарантирует выполнения в определённые сроки. Если на шине нет достаточного места, чтобы отправить целый пакет BULK, он распадается на несколько передач в или из устройства.

Конечные точки USB описаны в ядре структурой struct usb_host_endpoint. Эта структура содержит реальную информацию конечно точки в другой структуре struct usb_endpoint_descriptor. Из нее можно получить информацию об адресе данной конечной точки, ее типе, максимальном размере информации, которую эта конечная точка может обработать за раз.

Конечные точки USB завернуты в интерфейсы [1]. USB-интерфейсы обрабатывают только один тип логического соединения, такого как мышь, клавиатура или аудио-поток. Некоторые usb-устройства имеют несколько интерфейсов. Каждый драйвер USB управляет интерфейсом. USB-интерфейсы описаны в ядре структурой struct usb_interface.

Драйверу USB устройства обычно требуется преобразовать данные из заданной структуры struct usb_interface в структуру struct usb_device, которая необходима ядру USB для широкого круга функциональных вызовов. Для этого предоставляется функция interface_to_usbdev.

1.5.2 Блоки запроса USB

Код USB в ядре Linux взаимодействует со всеми устройствами USB помощью так называемых urb (USB request block, блок запроса USB). Этот блок запроса описывается структурой struct urb.

URB используется для передачи или в или из заданной конечной точки USB на заданное USB устройство в асинхронном режиме. В зависимости от потребностей, драйвер USB устройства может выделить для одной конечной точки много блоков или может повторно использовать один urb для множества разных конечных точек.

Типичный жизненный цикл содержит следующие шаги:

1. Создание urb драйвером USB.
2. Назначение urb в определенную точку конечную точку.
3. Передача драйвером USB устройства в USB ядро.
4. Передача urb драйвером USB в заданный драйвер контроллера USB узла для указанного устройства
5. Обработка драйвером контроллера USB узла, который выполняет передачу в USB устройство.

6. После завершения работы с urb драйвер контроллера USB узла уведомляет драйвер USB устройства.

URB может быть отменен драйвером, который передал urb или драйвером USB. URB создается динамически и содержит счетчик ссылок, что позволяет освободить urb автоматически, когда освобождается блок последним пользователем.

Структура struct urb не должна быть создана статически в драйвере или внутри другой структуры, потому что это нарушит схему подсчёта ссылок, используемую USB ядром для urb-ов. Она должна быть создана вызовом функции usb_alloc_urb. Для того, чтобы сказать USB ядру, что драйвер закончил работу с urb-ом, драйвер должен вызвать функцию usb_free_urb.

Для правильной инициализации urb-а, посылаемого в конечную точку устройства используются функции usb_fill_int_urb, usb_fill_bulk_urb, usb_fill_control_urb, которые используются для urb-ов прерывания, поточных и управляющих urb-ов соответственно. Изохронные Urb не имеют функции инициализации, поэтому должны быть проинициализированны вручную.

После того, как urb был надлежащим образом создан и проинициализирован USB драйвером, он готов быть отправленным в USB ядро для передачи в USB устройство. Это делается с помощью вызова функции usb_submit_urb.

Если вызов usb_submit_urb был успешен, передавая контроль над urb в USB ядро, функция возвращает 0; иначе возвращается отрицательное число ошибки. Если функция завершается успешно, завершающий обработчик urb (задаваемый указателем на функцию complete) вызывается только один раз, когда urb завершается. Когда вызывается эта функция, ядро USB завершает работу с URB и контроль над ним теперь возвращается драйверу устройства.

1.5.3 Регистрация USB-драйвера

USB драйвер является драйвером устройства, т.е. он должен подключиться к реальному устройству в пространстве аппаратных средств. Загрузка подходящего

драйвера осуществляется по комбинации PID/VID (Product ID/Vendor ID). Регистрация устройства выполняется с помощью команд usb_register, в которую передается указатель на струкруту usb_driver.

В этой структуре описываются сведения о драйвере. Она должна быть заполнена драйвером USB. В соответствующих полях должны быть указаны имя устройства - name, идентификационная таблица – id_table, используемая для автоматического обнаружения конкретного устройства, и две функции обратного вызова – probe и disconnect, которые вызываются ядром USB при горячем подключении и отключении устройства, соответственно.

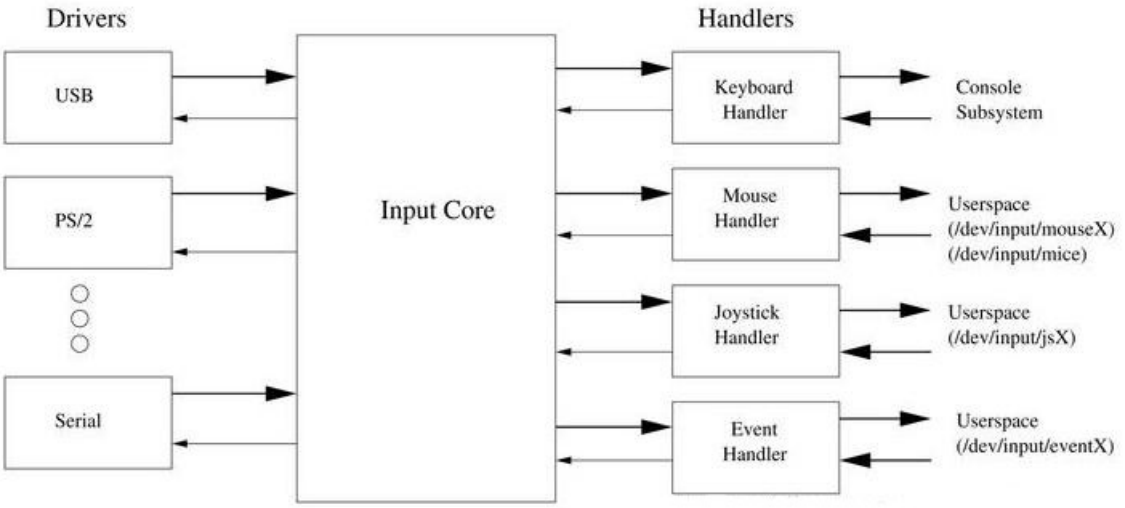
Структура struct usb_device_id содержит список различных типов USB устройств, которые поддерживает этот драйвер. Этот список используется ядром USB, чтобы решить, какой драйвер предоставить устройству, или скриптами горячего подключения, чтобы решить, какой драйвер автоматически загрузить, когда устройство подключается к системе. Эта структура записывается в поле id_table структуры usb_driver.

1.6 Драйвера ввода

Для того, чтобы объединить рассеянные драйверы устройств ввода данных, была создана подсистема ввода ядра [2]. Эта подсистема дает следующие преимущества:

- Единообразную обработку функционально похожих устройств ввода.
- Удобный интерфейс для событий отправки пользовательским приложениям сообщений о вводе.
- Выделение из входных драйверов общих частей. Это упрощает драйверы.

Рис 1.2.



Подсистема ввода [5]

Спецификациями usb, связанными с устройствами взаимодействия с человеком(HID) предусмотрен протокол, по которому для взаимодействия используются usb-клавиатуры, мыши, наборы кнопок и другие устройства ввода. В Linux это осуществляется через клиентский драйвер usbhid. Он регистрирует себя в качестве драйвера устройства ввода. Таким образом, если стоит задача написать свой драйвер устройства ввода, сначала необходимо выгрузить модуль usbhid, иначе он установит стандартный драйвер для устройства.

Чтобы зарегистрировать драйвер в качестве драйвера устройства ввода, сначала необходимо выделить память для структуры input_dev с использованием функции input_allocate_device. Затем необходимо объявить, какие события будет генерировать устройство ввода с помощью функции set_bit. После этого можно непосредственно зарегистрировать устройство ввода, вызвав функцию input_register_device.

Для генерации событий используются функции input_report_key, input_report_rel и другие, в зависимости от типа события.

1.7 Обработка прерываний

Обратный вызов urb, описанный ранее, выполняется в контексте прерывания, поэтому к нему применимы те же методы, что и для обработки прерываний.

Когда операционная система получает прерывания из-за некоторого аппаратного события, обработка начинается в контексте прерывания. Обычно прерывание приводит к выполнению большого объема работы. Однако обработчику прерывания необходимо завершиться быстро и не держать прерывания надолго заблокированными. Эти две потребности конфликтуют друг с другом.

Решить эту проблему можно, разделив обработчик прерывания на верхнюю и нижнюю половину ядра. Нижняя половина является процедурой, которая планируется верхней половиной, чтобы быть выполненной позднее, в более безопасное время.

Верхняя половина сохраняет данные устройства в зависимый от устройства буфер, планирует свою нижнюю половину и выходит: эта операция очень быстрая. Затем нижняя половина выполняет все необходимые операции. Эта установка позволяет верхней половине обслужить новое прерывание, пока нижняя половина всё ещё работает [1].

Существует несколько способов реализации нижней половины обработчика: гибкое прерывание (softirq), такслет (tasklet) и очереди работ (workqueue).

1.7.1 Тасклеты

Softirq-функции изначально были созданы в виде вектора из 32 записей softirq, поддерживающих разнообразные программные прерывания. Сегодня только девять векторов используются для softirqs, один из которых TASKLET_SOFTIRQ. Хотя softirqs-функции все еще существуют в ядре, рекомендуется использовать тасклеты и рабочие очереди вместо размещения новых векторов softirq [4].

Тасклеты являются структурами отложенного исполнения, которые вы можете запланировать на запуск позже в виде зарегистрированных функций. Конкретный

тасклет будет работать только на одном процессоре (для которого он запланирован), и один и тот же тасклет никогда не будет работать более чем на одном заданном процессоре одновременно. Но различные тасклеты могут одновременно работать на разных процессорах. Тасклеты представлены в виде структуры tasklet_struct (см. рис.2), в которой содержатся все данные, необходимые для управления тасклетом и поддержания его работы.

1.7.2 Очереди работ

Вместо того, чтобы предлагать однократную схему отложенного исполнения, как в случае с тасклетами, очереди работ являются обобщенным механизмом отложенного исполнения, в котором функция обработчика, используемая для очереди работ, может "засыпать".

В очередях работ предлагается обобщенный механизм, в котором отсроченная функциональность переносится в механизмы выполнения нижних половин. В основе лежит очередь работ (структура workqueue_struct), которая является структурой, в которую помещается данные объект work. Работа (т.е. объект work) представлена структурой work_struct, в которой идентифицируется работа, исполнение которой откладывается, и функция отложенного исполнения, которая будет при этом использоваться. Потоки ядра выбирают работу (т.е. объект work) из очереди работ и активируют один из обработчиков нижней половины.

После того, как будет инициализирована структура для объекта work, следующим шагом будет помещение этой структуры в очередь работ. Это можно сделать это несколькими способами. Можно просто добавить работу (объект work) в очередь работ с помощью функции queue_work(которая назначает работу текущему процессору). Либо с помощью функции queue_work_on можно указать процессор, на котором будет выполняться обработчик [4].

1.8 Выводы

В аналитическом разделе была выполнена постановка задачи. Были рассмотрены различные типы драйверов. Было установлено, что для графического

планшета Wacom Bamboo CTH-670 необходимо будет написать драйвер устройства ввода, который позволит реализовать весь необходимый функционал. Для обработки нижних половин прерываний будет использоваться очередь работ.

2 Конструкторский раздел

В конструкторском разделе будет описана структура разрабатываемого драйвера, функции драйвера, будут приведены схемы алгоритмов.

2.1 Инициализация модуля

Для инициализации модуля необходимо сначала заполнить структуру usb_device_id, которая описывает типы устройств, поддерживаемых данным драйвером.

На листинге 2.1 представлено заполнение этой структуры. Для заполнения используется макрос USB_DEVICE.

Листинг 2.1 Пример заполнения структуры usb_device_id

```
static struct usb_device_id tablet_devices_ids [] = {
{ USB_DEVICE(ID_VENDOR_TABLET, ID_PRODUCT_TABLET) },
{ },
};
```

Чтобы заполнить эту структуру, необходимо знать Vendor ID и Product ID устройства. Эти сведения можно получить с помощью команды lsusb, которая выводит сведения обо всех подключенных usb-устройствах.

Затем необходимо заполнить структуру usb_driver. В ней в соответствующих полях должны быть указаны имя устройства - name, идентификационная таблица – id_table (ранее заполненная структура usb_device_id), используемая для автоматического обнаружения конкретного устройства, и две функции обратного вызова – probe и disconnect, которые вызываются ядром USB при подключении и отключении устройства, соответственно.

На листинге 2.2 представлено заполнение структуры usb_driver.

Листинг 2.2 Пример заполнения структуры usb_driver.

```
static struct usb_driver tablet_driver = {
.name = DRIVER_NAME,
.probe = tablet_probe,
.disconnect = tablet_disconnect,
.id_table = tablet_devices_ids,
};
```

Затем необходимо определить функцию инициализации модуля. Эта функция будет использовать tablet_driver — заполненную ранее информацию о драйвере.

На рисунке 2.1 представлена схема алгоритма инициализации модуля.

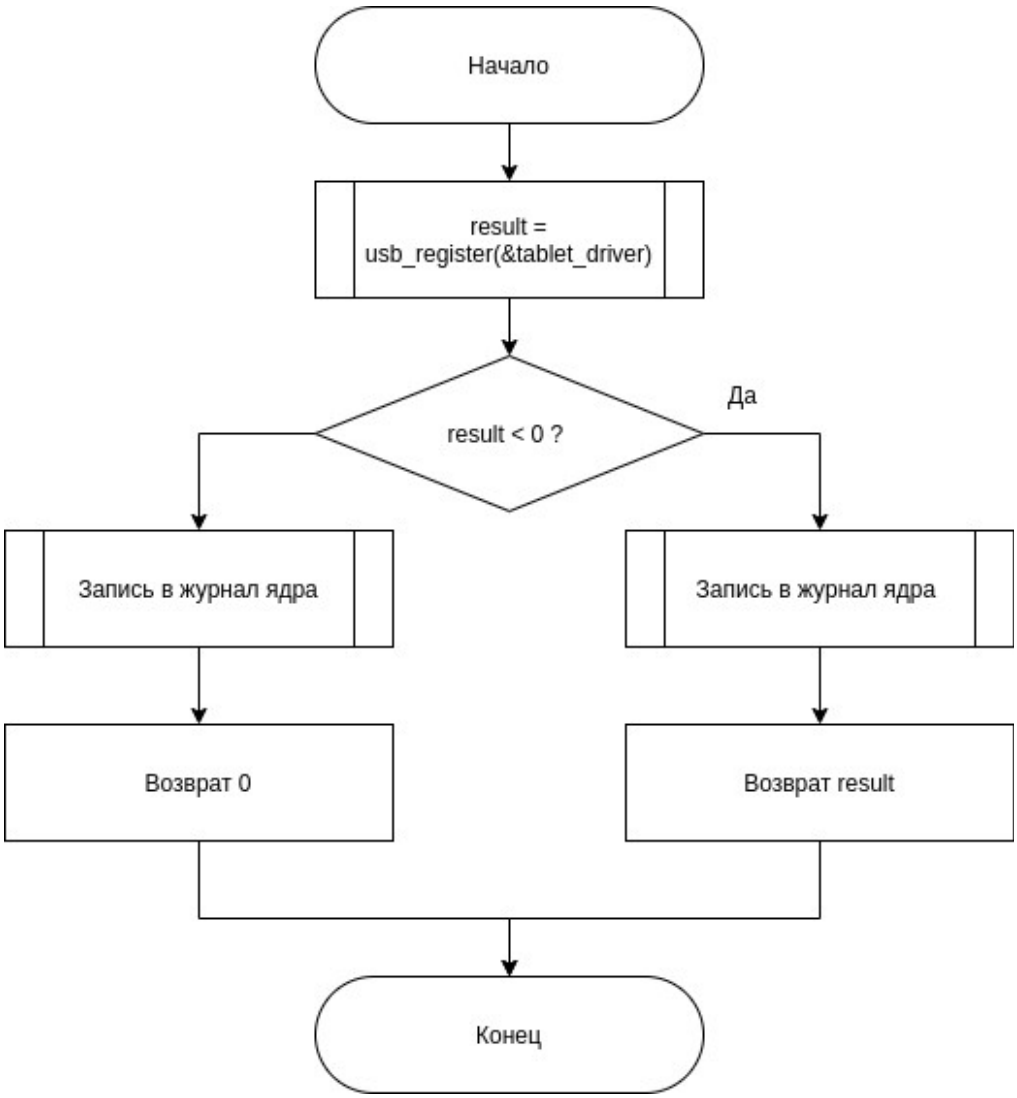


Рис 2.1 Схема инициализации модуля

2.2 Завершение работы модуля

Также необходимо определить функцию завершения работы модуля. Она должна выполнить отмену регистрации модуля.

На рисунке 2.2 представлена схема работы функции завершения работы модуля.

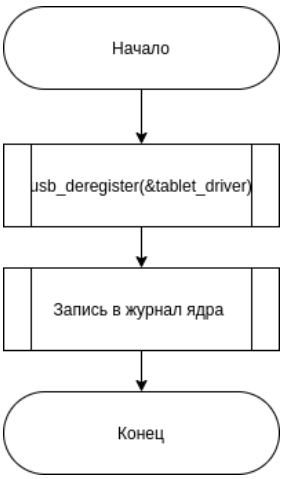


Рис 2.2 Схема завершения работы модуля

2.3 Подключение устройства

На рисунках 2.3-2.6 представлена схема работы функции, вызываемой при подключении устройства.

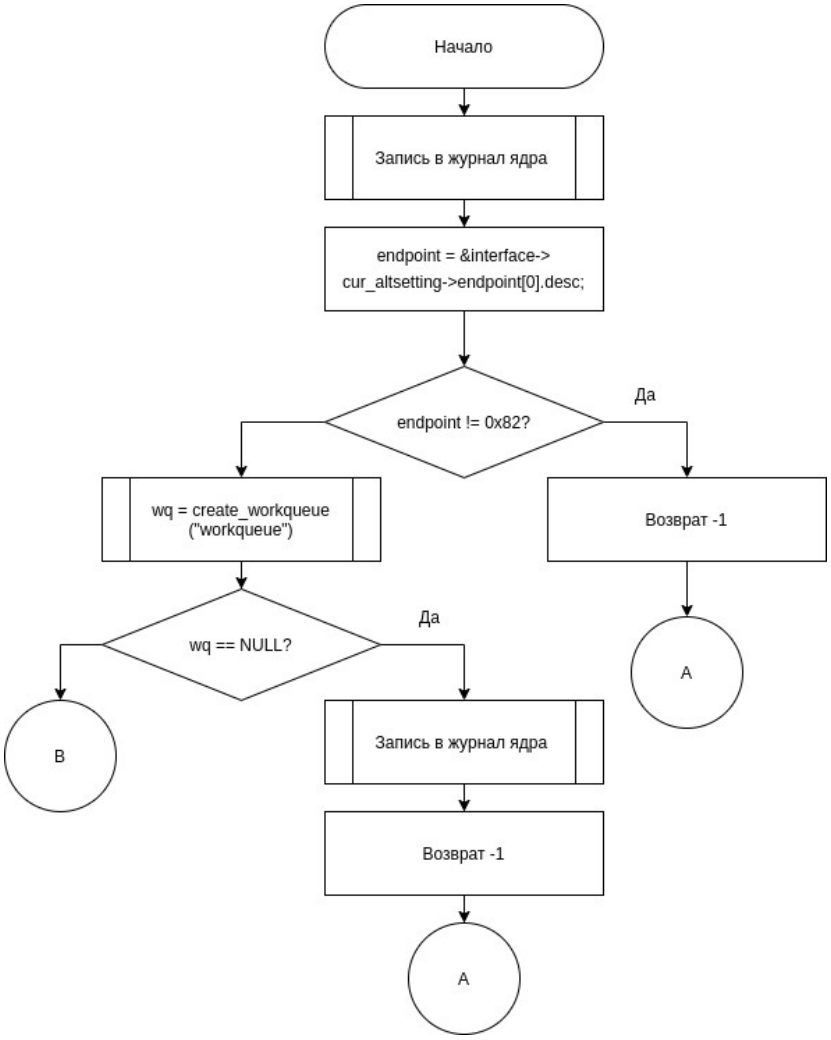


Рис 2.3 Схема функции, срабатывающей после подключения устройства (часть 1)

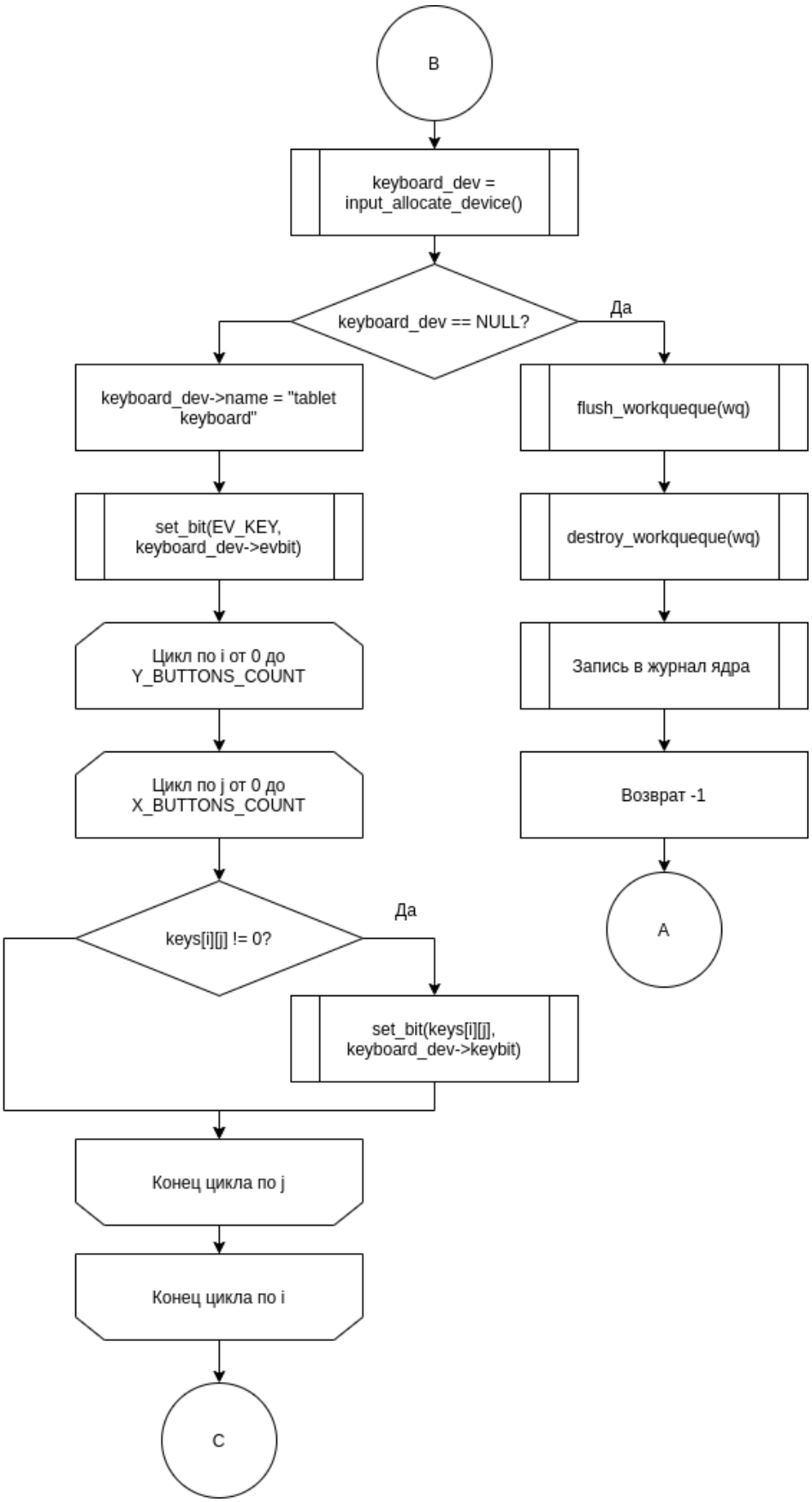


Рис 2.4 Схема функции, срабатывающей после подключения устройства(часть 2).

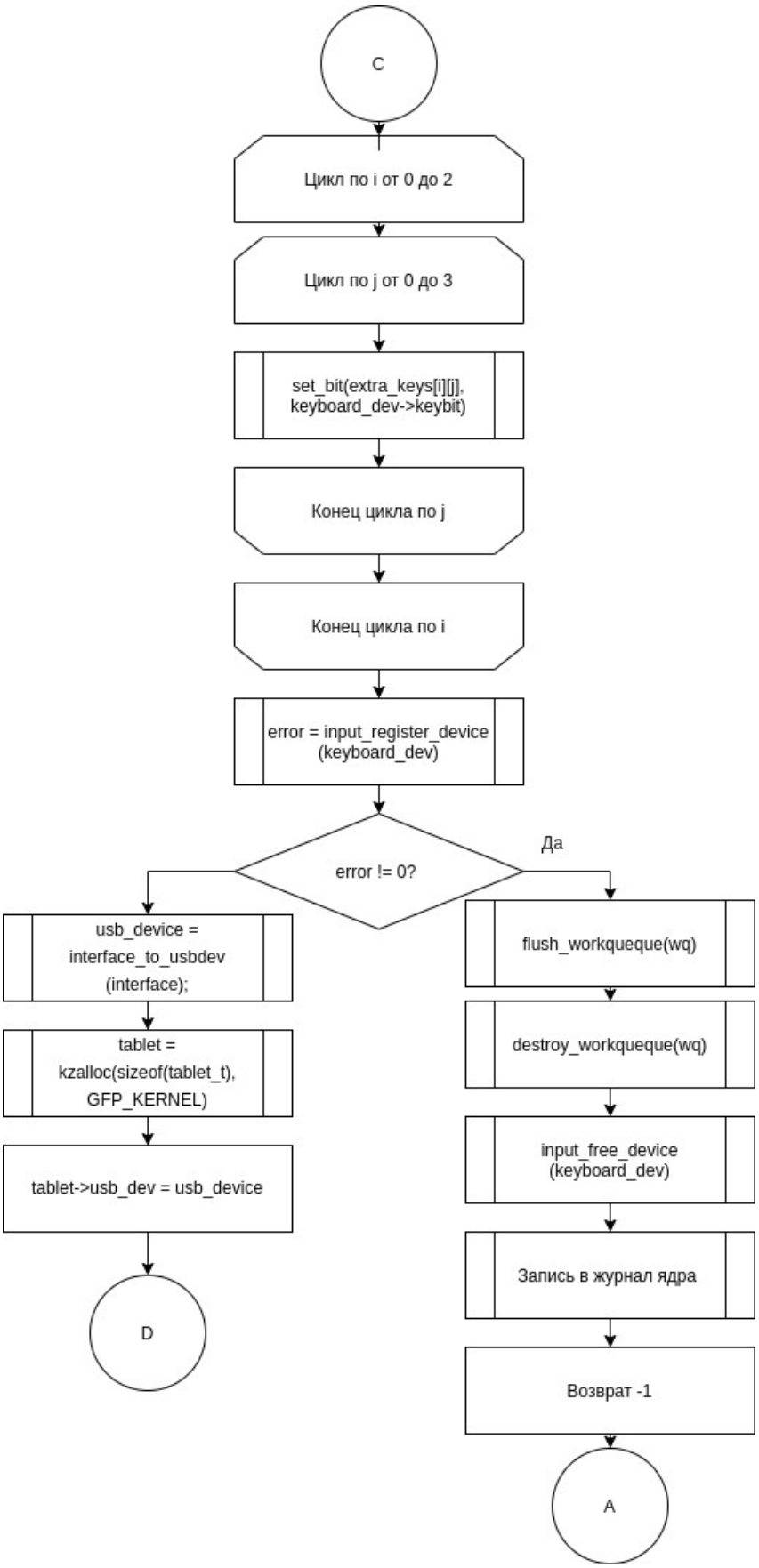


Рис 2.5 Схема функции, срабатывающей после подключения устройства (часть 3)

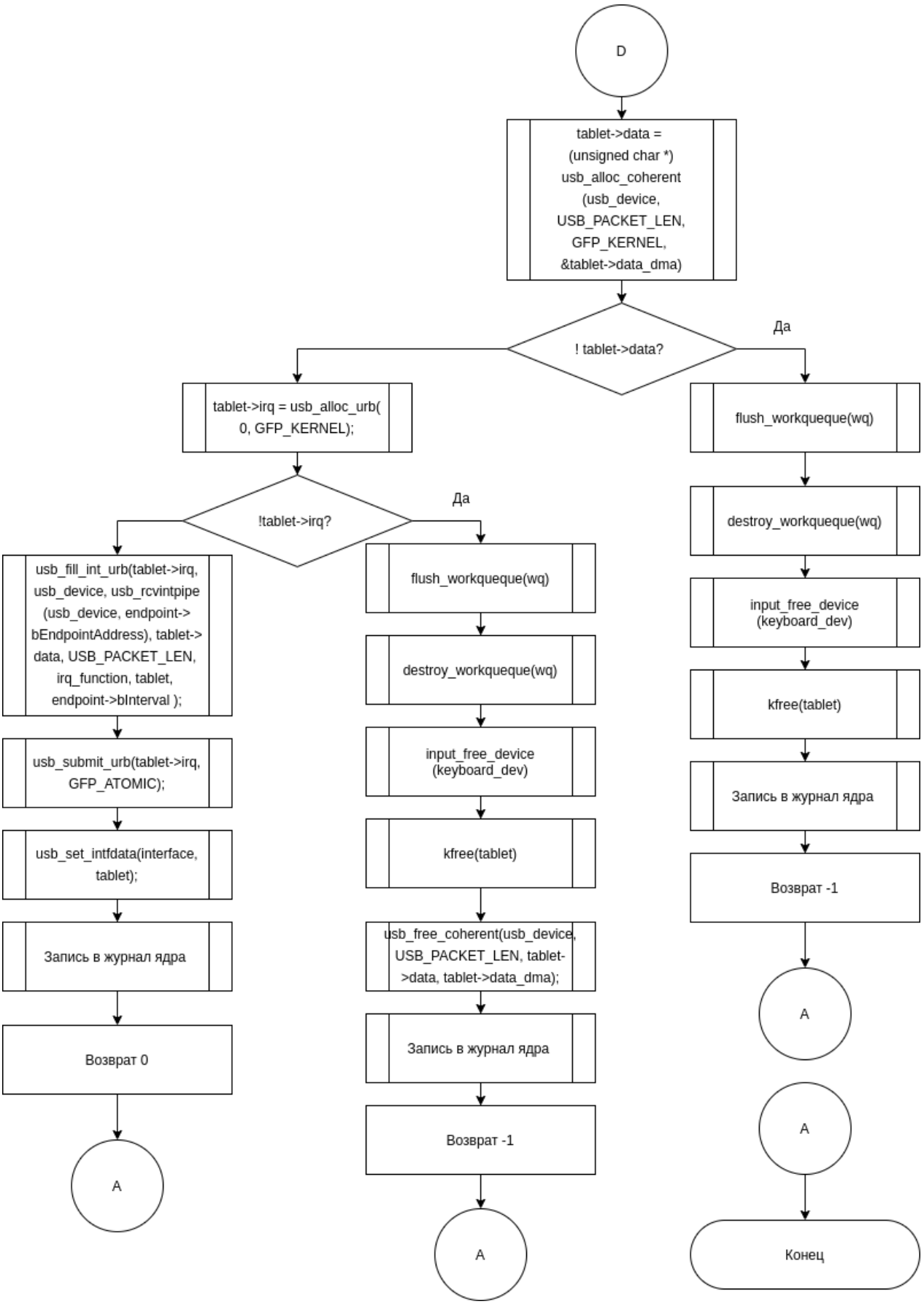


Рис 2.6 Схема функции, срабатывающей после подключения устройства (часть 4)

2.4 Отключение устройства

На рисунке 2.7 представлена схема алгоритма, срабатывающего после отключения устройства.

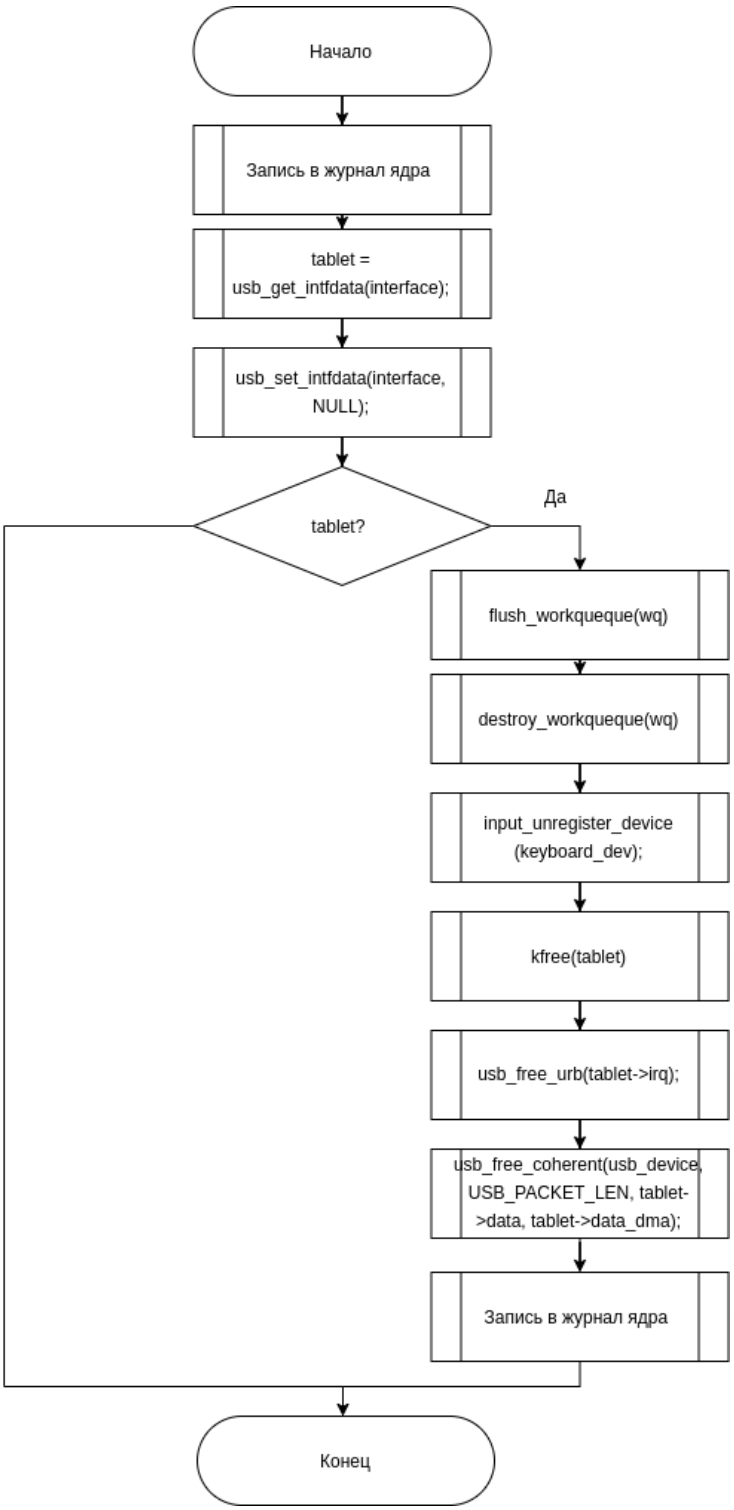


Рис 2.7 Схема функции, срабатывающей при отключении устройства.

2.5 Обработка прерываний устройства

На рисунке 2.8 представлена схема обработки верхней половины прерывания.

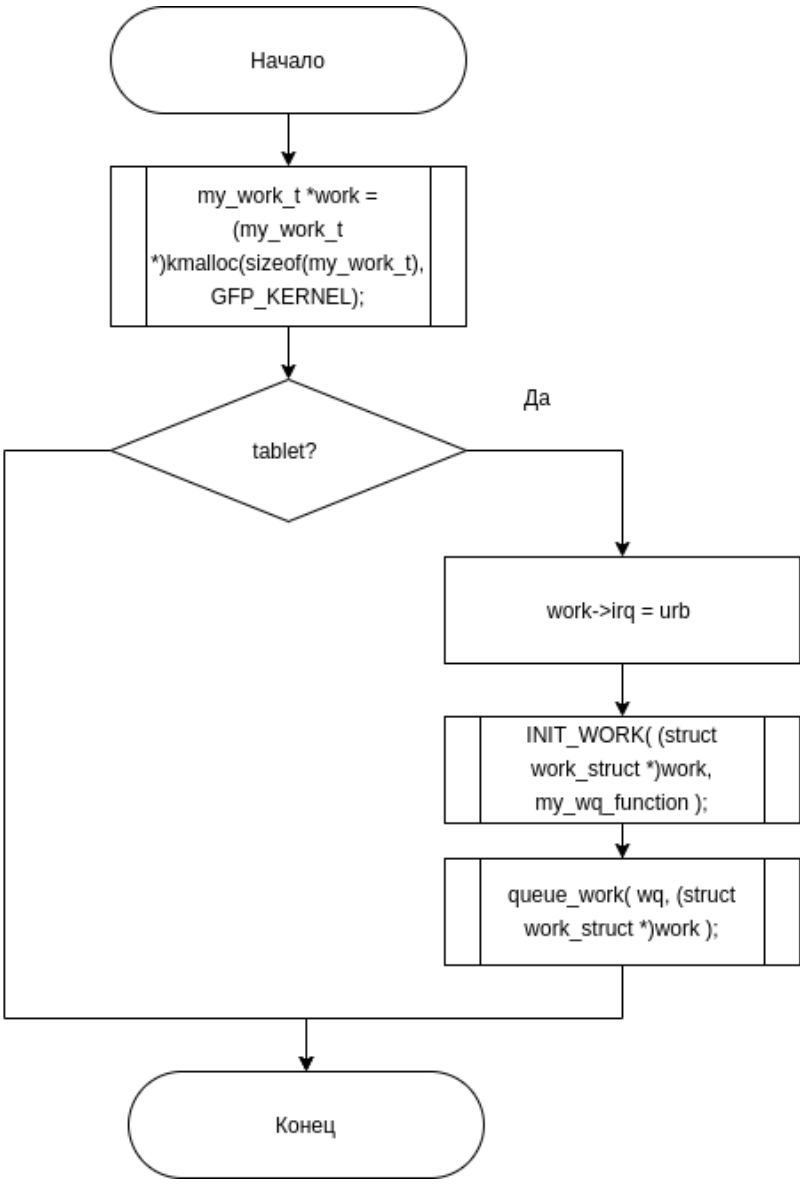


Рис 2.8 Схема обработки верхней половины прерывания

Эта функция планирует нижние половины. В нижних половинах происходит определение, что произошло, какая клавиша была нажата, генерируются события клавиатуры.

На рисунках 2.9- 2.10 представлена схема обработки нижних половин.

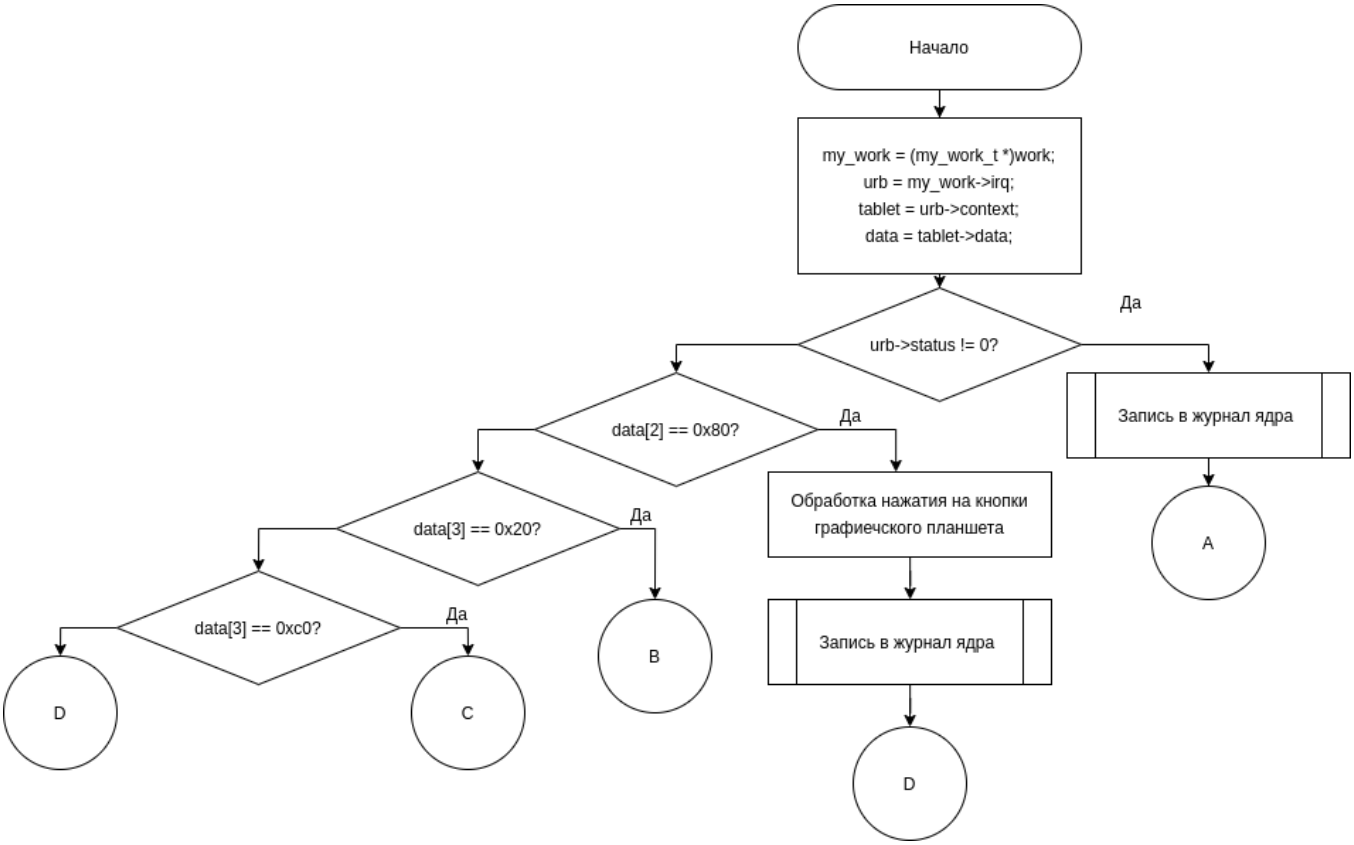


Рис 2.9 Схема обработки нижних половин. Определение, какое событие произошло.

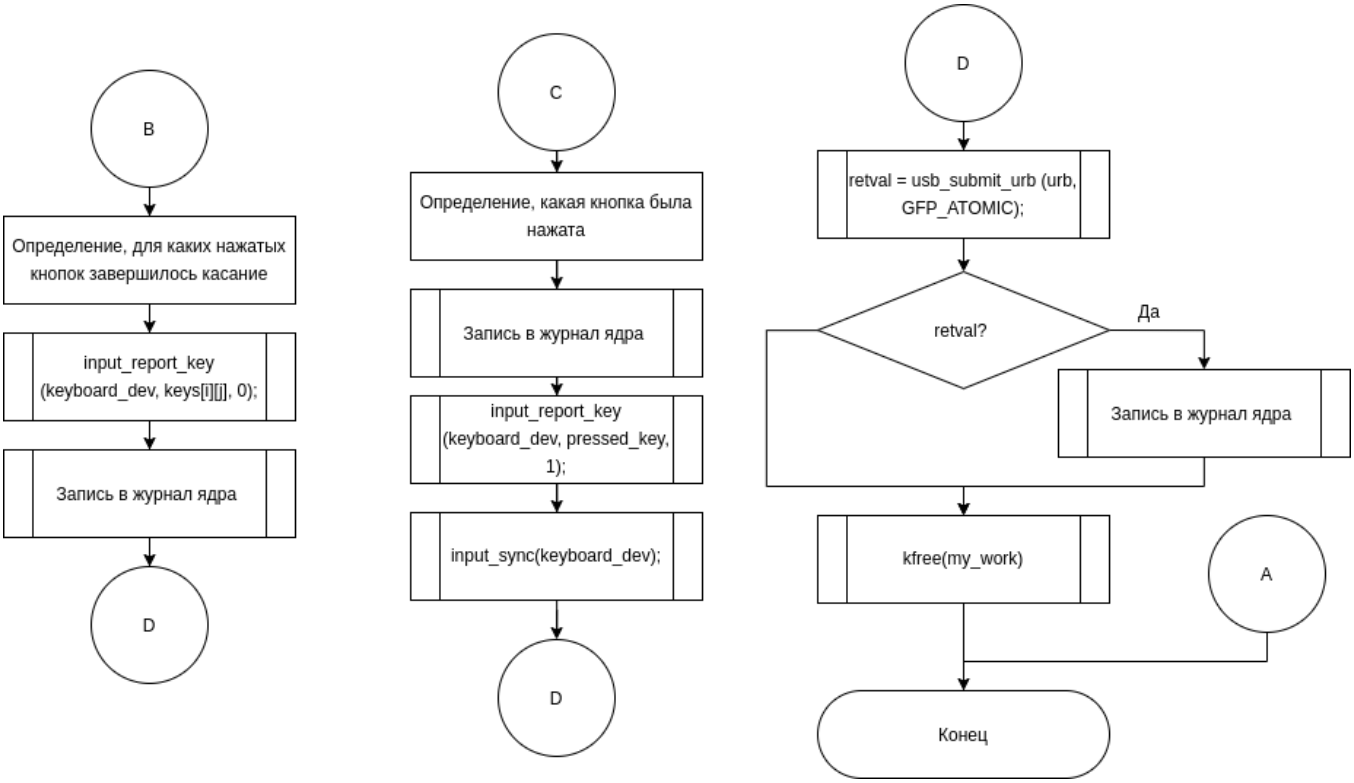


Рис 2.10 Схема обработки нижних половин. Алгоритм обработки нажатия на кнопку, окончания касания и завершение обработки.

2.6 Разделение поверхности планшета на кнопки

На рисунке 2.11 показано, как была разделена поверхность планшета на кнопки.

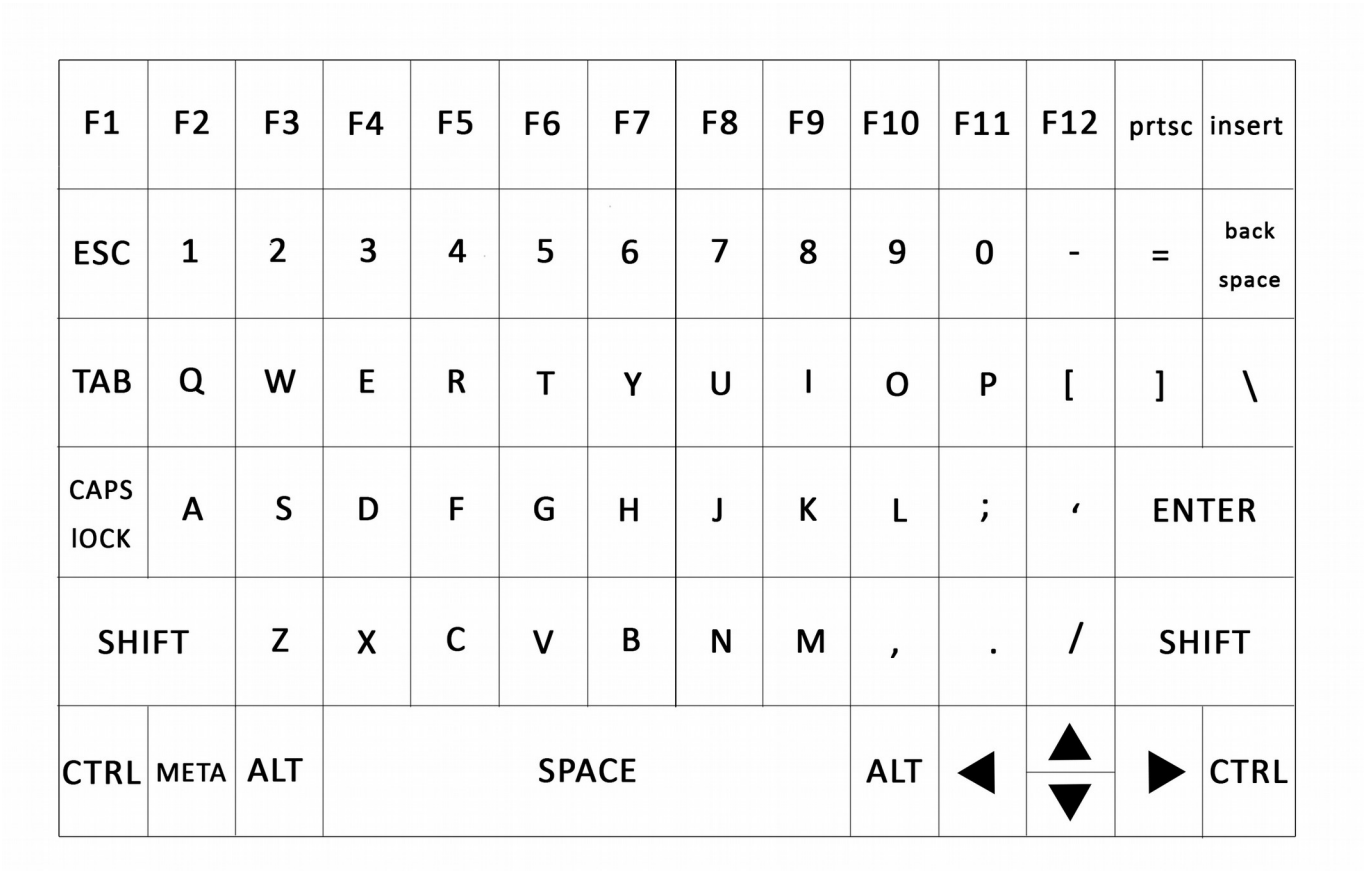


Рис 2.11 Разделение поверхности планшета на кнопки

2.7 Выводы

В конструкторском были приведены схемы алгоритмов, описана структура драйвера.

3 Технологический раздел

В технологическом разделе будет выполнен выбор языка и среды программирования, описана среда разработки. Будут приведены выбранные структуры данных и разработанные функции, реализующие поставленную задачу. Будет приведен make-файл.

3.1 Выбор языка программирования

Все модули ядра и драйверы операционной системы Linux реализованы на языке C. Этот язык и был выбран для реализации ПО. Для сборки модулей использовалась утилита make.

Была выбрана среда разработки Visual Studio Code, так как она позволяет удобно работать с файловой системой компьютера, а также использовать все возможности стандартной консоли, не переключаясь между окнами. Она бесплатная, кроссплатформенная и имеет множество плагинов для расширения функционала.

3.2 Структура модуля

Полученный модуль состоит из следующих функции:

- tablet_driver_init – точка входа в драйвер;
- tablet_driver_exit – точка выхода из драйвера;
- tablet_probe – функция, вызываемая при подключении устройства;
- tablet_disconnect – функции, вызываемая при отключении устройства;
- irq_function – функция обработки urb;
- my_wq_function – функция обработки нижней половины прерывания;

Были определены структуры tablet_data и my_work. Первая отвечает за внутренние данные драйвера, вторая позволяет передавать дополнительные данные между верхней и нижней половиной прерывания.

Листинги кода программы представлены в приложении А.

3.3 Сборка и запуск модуля

Сборка модуля осуществляется командой make. На листинге 3.1 представлено содержимое Make-файла.

Листинг 3.1 Makefile

```
ifneq ($(KERNELRELEASE),)
obj-m := tablet_driver.o
else
CURRENT = $(shell uname -r)
KDIR = /lib/modules/$(CURRENT)/build
PWD = $(shell pwd)

all:
$(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
rm -rf .tmp_versions
rm *.o
rm *.mod.c
rm *.symvers
rm *.order
rm .*.cmd
endif
```

Модуль можно загрузить в ядро с помощью команды sudo insmod tablet_driver.ko. Выгрузить из ядра — с помощью команды sudo rmmod tablet_driver.ko.

3.4 Примеры работы модуля

На рисунке 3.1 показаны логи загрузки модуля и подключения к нему планшета. В момент загрузки модуля планшет уже был подключен к компьютеру. Можно заметить, что подключается сначала первый интерфейс с конченой точкой 81, однако модуль не обрабатывает его. Это связано с тем, что прерывания от нажатия на планшет передаются только на конечную точку 82.

```
[ 4120.371730] tablet_keyboard_driver: tablet probe
[ 4120.371733] tablet_keyboard_driver: endpoint 81
[ 4120.371739] tablet_keyboard_driver: probe of 1-3:1.0 failed with error -1
[ 4120.371744] tablet_keyboard_driver: tablet probe
[ 4120.371745] tablet_keyboard_driver: endpoint 82
[ 4120.371918] input: virtual tablet keyboard as /devices/virtual/input/input24
[ 4120.372041] tablet_keyboard_driver: device is conected
[ 4120.372106] usbcore: registered new interface driver tablet_keyboard_driver
[ 4120.372107] tablet_keyboard_driver: module loaded
```

На рисунке 3.2 показаны логи работы драйвера. Можно увидеть информацию о том, какая кнопка когда была нажата, и когда завершилось нажатие.

```
[ 5814.934286] tablet_keyboard_driver: pressed key O
[ 5815.022303] tablet_keyboard_driver: touch ends
[ 5815.324288] tablet_keyboard_driver: pressed key L
[ 5815.392297] tablet_keyboard_driver: touch ends
[ 5815.734280] tablet_keyboard_driver: pressed key N
[ 5815.772301] tablet_keyboard_driver: touch ends
[ 5816.164282] tablet_keyboard_driver: pressed key H
[ 5816.222306] tablet_keyboard_driver: touch ends
[ 5816.464287] tablet_keyboard_driver: pressed key T
[ 5816.532295] tablet_keyboard_driver: touch ends
[ 5816.694294] tablet_keyboard_driver: pressed key 6
[ 5816.782304] tablet_keyboard_driver: touch ends
[ 5816.964284] tablet_keyboard_driver: pressed key 7
[ 5817.032311] tablet_keyboard_driver: touch ends
[ 5817.214282] tablet_keyboard_driver: pressed key 6
[ 5817.282307] tablet_keyboard_driver: touch ends
[ 5817.504289] tablet_keyboard_driver: pressed key R
[ 5817.572339] tablet_keyboard_driver: touch ends
[ 5817.774290] tablet_keyboard_driver: pressed key F
[ 5817.862316] tablet_keyboard_driver: touch ends
[ 5818.064289] tablet_keyboard_driver: pressed key G
[ 5818.162311] tablet_keyboard_driver: touch ends
[ 5818.364287] tablet_keyboard_driver: pressed key J
[ 5818.462308] tablet_keyboard_driver: touch ends
[ 5818.644275] tablet_keyboard_driver: pressed key L
[ 5818.762310] tablet_keyboard_driver: touch ends
[ 5818.934287] tablet_keyboard_driver: pressed key I
[ 5819.022309] tablet_keyboard_driver: touch ends
[ 5819.194287] tablet_keyboard_driver: pressed key T
[ 5819.272309] tablet_keyboard_driver: touch ends
[ 5819.504293] tablet_keyboard_driver: pressed key D
[ 5819.582290] tablet_keyboard_driver: touch ends
[ 5819.814297] tablet_keyboard_driver: pressed key E
[ 5819.892315] tablet_keyboard_driver: touch ends
[ 5820.074284] tablet_keyboard_driver: pressed key 5
[ 5820.162310] tablet_keyboard_driver: touch ends
[ 5820.344297] tablet_keyboard_driver: pressed key U
[ 5820.422317] tablet_keyboard_driver: touch ends
[ 5820.584299] tablet_keyboard_driver: pressed key U
[ 5820.652340] tablet_keyboard_driver: touch ends
[ 5820.784300] tablet_keyboard_driver: pressed key 7
[ 5820.872313] tablet_keyboard_driver: touch ends
[ 5821.004292] tablet_keyboard_driver: pressed key 7
[ 5821.092304] tablet_keyboard_driver: touch ends
[ 5821.174285] tablet_keyboard_driver: pressed key 6
[ 5821.252301] tablet_keyboard_driver: touch ends
```

Рис 3.2 Логи работы драйвера

На рисунке 3.3 изображены логи отключения планшета и выгрузки модуля.

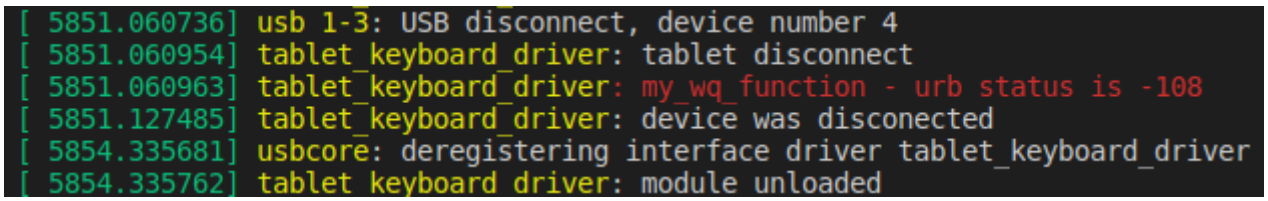


Рис 3.3 Отключение устройства, выгрузка модуля

3.5 Выводы

В конструкторском разделе была рассмотрена структура полученной программы, а также приведены примеры работы программы.

Заключение

В ходе работы были выполнены поставленные цели и задачи:

- Были изучены способы реализации драйвера ОС Linux.
- Было определено, к какому типу устройств относится графический планшет, каким образом драйвер сможет получать информацию о нажатии на графическим планшет.
- Были изучены способы имитации работы клавиатуры.
- Был разработан драйвер графического планшета.

Таким образом, достигнута основная цель курсового проекта и разработано необходимое программное обеспечение.

Список литературы

1. Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman. Linux Device Drivers, Third Edition - O’Reilly Media, 2005.
2. Venkateswaran, Sreekrishnan- Essential Linux device drivers / Sreekrishnan Venkateswaran.-- 1st ed. - Prentice Hall, 2008
3. linux.input [Электронный ресурс] — Режим доступа: https://valadoc.org/linux/Linux.Input.html
4. API ядра Linux, Часть 2: Функции отложенного выполнения, тасклеты ядра и очереди работ [Электронный ресурс] — Режим доступа: http://rus-linux.net/nlib.php?name=/MyLDP/kernel/api/kernelapi2.html
5. Подсистема ввода Linux [Электронный ресурс] — Режим доступа: https://russianblogs.com/article/8248247507/

ПРИЛОЖЕНИЕ А

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/usb/input.h>
#include <linux/slab.h>
#include <linux/errno.h>
#include <linux/input.h>
#include <linux/workqueue.h>

#define DRIVER_NAME    "tablet_keyboard_driver"
#define DRIVER_LICENSE "GPL"

MODULE_LICENSE(DRIVER_LICENSE);

#define ID_VENDOR_TABLET  0x056a /* Wacom Co. */
#define ID_PRODUCT_TABLET 0x00df /* CTH-670 */

#define USB_PACKET_LEN  64

#define MAX_VALUE 0xFF

#define X_BUTTONS_COUNT 14
#define Y_BUTTONS_COUNT 6

#define X_BUTTON_LEN ( MAX_VALUE / X_BUTTONS_COUNT )
#define Y_BUTTON_LEN ( MAX_VALUE / Y_BUTTONS_COUNT )

struct tablet_data {
   unsigned char     *data;
   dma_addr_t        data_dma;
   struct usb_device *usb_dev;
   struct urb        *irq;
};

typedef struct tablet_data tablet_t;

typedef struct {
  struct work_struct my_work;
  struct urb        *irq;
} my_work_t;

static struct input_dev *keyboard_dev;
static struct workqueue_struct *wq;

static int pressed[Y_BUTTONS_COUNT][X_BUTTONS_COUNT];
static int dop_pressed[2][1];
```

```
static int buttons_keys[Y_BUTTONS_COUNT][X_BUTTONS_COUNT] = {
   { KEY_F1, KEY_F2, KEY_F3, KEY_F4, KEY_F5, KEY_F6, KEY_F7, KEY_F8, KEY_F9,
KEY_F10, KEY_F11, KEY_F12, KEY_PRINT, KEY_INSERT},
   { KEY_ESC, KEY_1, KEY_2, KEY_3, KEY_4, KEY_5, KEY_6, KEY_7, KEY_8, KEY_9, KEY_0,
KEY_MINUS, KEY_EQUAL, KEY_BACKSPACE },
   { KEY_TAB, KEY_Q, KEY_W, KEY_E, KEY_R, KEY_T, KEY_Y, KEY_U, KEY_I, KEY_O,
KEY_P, KEY_LEFTBRACE, KEY_RIGHTBRACE, KEY_BACKSLASH },
   { KEY_CAPSLOCK, KEY_A, KEY_S, KEY_D, KEY_F, KEY_G, KEY_H, KEY_J, KEY_K,
KEY_L, KEY_SEMICOLON, KEY_APOSTROPHE, KEY_ENTER, KEY_ENTER },
   { KEY_LEFTSHIFT, KEY_LEFTSHIFT, KEY_Z, KEY_X, KEY_C, KEY_V, KEY_B, KEY_N,
KEY_M, KEY_COMMA, KEY_DOT, KEY_SLASH, KEY_RIGHTSHIFT, KEY_RIGHTSHIFT },
   { KEY_LEFTCTRL, KEY_LEFTMETA, KEY_LEFTALT, KEY_SPACE, KEY_SPACE,
KEY_SPACE, KEY_SPACE, KEY_SPACE, KEY_SPACE, KEY_RIGHTALT, KEY_LEFT,  0,
KEY_RIGHT, KEY_RIGHTCTRL },
};

static char* buttons[Y_BUTTONS_COUNT][X_BUTTONS_COUNT] = {
   { "f1", "f2", "f3", "f4", "f5", "f6", "f7", "f8", "f9", "f10", "f11", "f12", "PRTSC", "INSERT"},
   { "ESC", "1", "2", "3", "4", "5", "6", "7", "8", "9", "0", "-", "=", "BACKSPACE" },
   { "TAB", "Q", "W", "E", "R", "T", "Y", "U", "I", "O", "P", "[", "]", "\\" },
   { "CAPSLOCK", "A", "S", "D", "F", "G", "H", "J", "K", "L", ";", "'", "ENTER","ENTER" },
   { "LEFTSHIFT", "LEFTSHIFT", "Z", "X", "C", "V", "B", "N", "M", ",",".","/" , "RIGHTSHIFT",
"RIGHTSHIFT" },
   { "LEFTCTRL", "LEFTMETA", "LEFTALT", "SPACE", "SPACE", "SPACE","SPACE", "SPACE",
"SPACE", "RIGHTALT", "LEFT", 0,  "RIGHT" ,"RIGHTCTRL" },
};

static int dop_keys[2][1] = {
   { KEY_UP},
   { KEY_DOWN},
};

static char * dop_buttons[2][1] = {
   {  "UP" },
   { "DOWN" },
};

static void my_wq_function( struct work_struct *work)
{
   my_work_t *my_work;
   int retval;
   u16 x, y;
   struct urb *urb;
   tablet_t *tablet;
   unsigned char *data;
   int pressed_key;

   my_work = (my_work_t *)work;
   urb = my_work->irq;
   tablet = urb->context;
```

```
data = tablet→data;

if (urb->status != 0) {
   printk(KERN_ERR "%s: %s - urb status is %d\n", DRIVER_NAME, __func__, urb->status);
   return;
}

if (data[2] == 0x80) {
   if (data[3] == 0x1) {
        printk(KERN_INFO "%s: left left click\n", DRIVER_NAME);
   }
   else if (data[3] == 0x2) {
        printk(KERN_INFO "%s: left center click\n", DRIVER_NAME);
   }
   else if (data[3] == 0x10) {
        printk(KERN_INFO "%s: right center click\n", DRIVER_NAME);
   }
   else if (data[3] == 0x8) {
        printk(KERN_INFO "%s: rigtn rigth click\n", DRIVER_NAME);
   }
   else if (data[3] == 0x0 && data[4] == 0x0 && data[5] == 0x0) {
       printk(KERN_INFO "%s: click ends\n", DRIVER_NAME);
   }
   else  {
      int i = 0;
      while (i < 10) {
         printk(KERN_INFO "%s: data[%d] = %x\n", DRIVER_NAME, i,  data[i]);
         i++;
      }
   }
}
else if (data[3] == 0x20) {
   int i;
   int j;
   for (i = 0; i < Y_BUTTONS_COUNT; i++) {
      for (j = 0; j < X_BUTTONS_COUNT; j++)
      {
         if (pressed[i][j]) {
            input_report_key(keyboard_dev, buttons_keys[i][j], 0);
            input_sync(keyboard_dev);
            pressed[i][j] = 0;
            printk(KERN_INFO "%s: touch ends\n", DRIVER_NAME);
         }
      }
   }
   for (i = 0; i < 2; i++) {
      for (j = 0; j < 1; j++)
      {
         if (dop_pressed[i][j]) {
            input_report_key(keyboard_dev, dop_keys[i][j], 0);
```

```
                input_sync(keyboard_dev);
                dop_pressed[i][j] = 0;
                printk(KERN_INFO "%s: touch ends\n", DRIVER_NAME);
            }
        }
    }
  }
  else if (data[3] == 0xc0) {
    x = data[4];
    y = data[5];
    pressed_key = buttons_keys[y / Y_BUTTON_LEN][x / X_BUTTON_LEN];
    pressed[y / Y_BUTTON_LEN][x / X_BUTTON_LEN] = 1;
    if (pressed_key == 0) {
      pressed_key = dop_keys[y % Y_BUTTON_LEN / (Y_BUTTON_LEN / 2)][x %
X_BUTTON_LEN / (X_BUTTON_LEN )];
      dop_pressed[y % Y_BUTTON_LEN / (Y_BUTTON_LEN / 2)][x % X_BUTTON_LEN /
(X_BUTTON_LEN )] = 1;
      printk(KERN_INFO "%s: pressed key %s\n", DRIVER_NAME, dop_buttons[y %
Y_BUTTON_LEN / (Y_BUTTON_LEN / 2)][x % X_BUTTON_LEN / (X_BUTTON_LEN)]);
    } else {
       printk(KERN_INFO "%s: pressed key %s\n", DRIVER_NAME, buttons[y /
Y_BUTTON_LEN][x / X_BUTTON_LEN]);
    }

    input_report_key(keyboard_dev, pressed_key, 1);
    input_sync(keyboard_dev);
  }

  retval = usb_submit_urb (urb, GFP_ATOMIC);
  if (retval)
    printk(KERN_ERR "%s: %s - usb_submit_urb failed with result %d\n", DRIVER_NAME,
__func__, retval);
  kfree(my_work);
}

static void irq_function(struct urb *urb) {
  my_work_t *work = (my_work_t *)kmalloc(sizeof(my_work_t), GFP_KERNEL);

  if (work) {
    work->irq = urb;
    INIT_WORK( (struct work_struct *)work, my_wq_function );
    queue_work( wq, (struct work_struct *)work );
  }
}

static int tablet_probe(struct usb_interface *interface, const struct usb_device_id *id) {
  struct usb_endpoint_descriptor *endpoint;
  struct usb_device *usb_device;
  tablet_t *tablet;
  int error = -ENOMEM, i, j;
```

```
printk(KERN_INFO "%s: tablet probe\n", DRIVER_NAME);
endpoint = &interface→cur_altsetting→endpoint[0].desc;

printk(KERN_INFO "%s: endpoint %x\n", DRIVER_NAME, endpoint→bEndpointAddress);

if (endpoint->bEndpointAddress != 0x82) {
   return -1;
}

wq = create_workqueue("workqueue");
if (wq == NULL) {
   printk(KERN_ERR "%s: allocation workqueue error\n", DRIVER_NAME);
   return -1;
}

keyboard_dev = input_allocate_device();
if (keyboard_dev == NULL) {
   flush_workqueue(wq);
   destroy_workqueue(wq);
   printk(KERN_ERR "%s: allocation device error\n", DRIVER_NAME);
   return -1;
}

keyboard_dev->name = "virtual tablet keyboard";

set_bit(EV_KEY, keyboard_dev->evbit);
for (i = 0; i < Y_BUTTONS_COUNT; ++i) {
   for (j = 0; j < X_BUTTONS_COUNT; ++j) {
      if (buttons_keys[i][j] != 0) {
         set_bit(buttons_keys[i][j], keyboard_dev->keybit);
      }
   }
}
for (i = 0; i < 2; ++i) {
   for (j = 0; j < 1; ++j) {
      set_bit(dop_keys[i][j], keyboard_dev->keybit);
   }
}

error = input_register_device(keyboard_dev);
if (error != 0) {
   flush_workqueue(wq);
   destroy_workqueue(wq);
   input_free_device(keyboard_dev);
   printk(KERN_ERR "%s: registration device error\n", DRIVER_NAME);
   return error;
}

usb_device = interface_to_usbdev(interface);
tablet = kzalloc(sizeof(tablet_t), GFP_KERNEL);
```

```
    tablet->usb_dev = usb_device;
    tablet->data = (unsigned char *)usb_alloc_coherent(usb_device, USB_PACKET_LEN,
GFP_KERNEL, &tablet→data_dma);
    if (!tablet->data) {
        flush_workqueue(wq);
        destroy_workqueue(wq);
        input_free_device(keyboard_dev);
        kfree(tablet);
        printk(KERN_ERR "%s: error when allocate coherent\n", DRIVER_NAME);
        return error;
    }

    tablet->irq = usb_alloc_urb( 0, GFP_KERNEL);
    if (!tablet->irq) {
        flush_workqueue(wq);
        destroy_workqueue(wq);

        input_free_device(keyboard_dev);
        usb_free_coherent(usb_device, USB_PACKET_LEN, tablet->data, tablet->data_dma);
        kfree(tablet);
        printk(KERN_ERR "%s: error when allocate urb\n", DRIVER_NAME);
        return error;
    }

    usb_fill_int_urb(
        tablet->irq,
        usb_device,
        usb_rcvintpipe(usb_device, endpoint->bEndpointAddress),
        tablet->data,
        USB_PACKET_LEN,
        irq_function,
        tablet,
        endpoint->bInterval
    );
    usb_submit_urb(tablet->irq, GFP_ATOMIC);
    usb_set_intfdata(interface, tablet);
    printk(KERN_INFO "%s: device is conected\n", DRIVER_NAME);
    return 0;
}

static void tablet_disconnect(struct usb_interface *interface) {
    tablet_t *tablet;

    printk(KERN_INFO "%s: tablet disconnect\n", DRIVER_NAME);
    tablet = usb_get_intfdata(interface);
    usb_set_intfdata(interface, NULL);
    if (tablet) {
        flush_workqueue(wq);
        destroy_workqueue(wq);
        usb_kill_urb(tablet->irq);
```

```
        input_unregister_device(keyboard_dev);
        usb_free_urb(tablet->irq);
        usb_free_coherent(interface_to_usbdev(interface), USB_PACKET_LEN, tablet->data, tablet-
>data_dma);
        kfree(tablet);
        printk(KERN_INFO "%s: device was disconected\n", DRIVER_NAME);
    }
}

static struct usb_device_id tablet_devices_ids [] = {
    { USB_DEVICE(ID_VENDOR_TABLET, ID_PRODUCT_TABLET) },
    { },
};

MODULE_DEVICE_TABLE(usb, tablet_devices_ids);

static struct usb_driver tablet_driver = {
    .name       = DRIVER_NAME,
    .probe      = tablet_probe,
    .disconnect = tablet_disconnect,
    .id_table   = tablet_devices_ids,
};

static int __init tablet_driver_init(void) {
    int result = usb_register(&tablet_driver);
    if (result < 0) {
        printk(KERN_ERR "%s: usb register error\n", DRIVER_NAME);
        return result;
    }
    printk(KERN_INFO "%s: module loaded\n", DRIVER_NAME);
    return 0;
}

static void __exit tablet_driver_exit(void) {
    usb_deregister(&tablet_driver);
    printk(KERN_INFO "%s: module unloaded\n", DRIVER_NAME);
}

module_init(tablet_driver_init);
module_exit(tablet_driver_exit);
```