

ZAKŁAD ARCHITEKTURY KOMPUTERÓW

ORGANIZACJA I ARCHITEKTURA KOMPUTERÓW

PROJEKT - NAPISANIE STEROWNIKA
KLAWIATURY USB DO SYSTEMU LINUX

Dokumentacja projektowa

Autorzy

MATEUSZ BĄCZEK
MAREK CHOIŃSKI
PIOTR TOCICKI

Prowadzący

MGR PRZEMYSŁAW
ŚWIERCZ

16 czerwca 2020

Spis treści

1	Wstęp	2
1.1	Cel projektu	2
2	Opis działania	3
2.1	Czym są sterowniki	3
2.2	Podstawy działania modułów jądra Linux	3
2.3	Moduły a sterowniki	3
2.4	Urządzenia HID	4
3	Rozwiązanie problemu	4
3.1	Język i środowisko	4
3.2	Proces kompilacji modułu	5
3.2.1	Ładowanie modułu i weryfikacja działania	5
3.3	Struktura programu	6
3.3.1	Ułatwienia, jakie Linux oferuje programistom	6
3.3.2	Rozpoznawanie i reagowanie na <i>Konami Code</i>	7
3.4	Obejście standardowego sterownika klawiatury USB	9
3.4.1	Problem z domyślnym sterownikiem urządzeń HID	9
3.4.2	Modyfikacja konfiguracji udev	9
3.5	Kompilacja jądra Linuxa, z naszym sterownikiem wbudowanym w jądro	10
3.5.1	Wymagana konfiguracja kernela	10
3.5.2	Kompilacja sterownika usbkbd.c - ryzykowna sztuczka	10
3.6	Spakowanie sterownika w kernel patch	13
4	Wnioski i obserwacje	13
5	Literatura	14

1 Wstęp

1.1 Cel projektu

Celem projektu było napisanie sterownika do klawiatury USB, który poza standardową obsługą klawiatury, będzie też rozpoznawał i reagował na *Konami Code* - sekwencję znaków, która często umieszczana jest w grach komputerowych jako ukryta niespodzianka.

Konami Code składa się z dziesięciu znaków:

- | | |
|--------------------|---------------------|
| 1. Strzałka w górę | 6. Strzałka w lewo |
| 2. Strzałka w górę | 7. Strzałka w prawo |
| 3. Strzałka w dół | 8. Strzałka w prawo |
| 4. Strzałka w dół | 9. Klawisz A |
| 5. Strzałka w lewo | 10. Klawisz B |

Wprowadzenie tych znaków w dokładnie takiej kolejności, jak opisana powyżej, aktywuje ukryte funkcje i niespodzianki w wielu grach komputerowych. W naszym sterowniku, po wprowadzeniu kodu, niezależnie od przyciskanych przycisków na klawiaturze, do systemu operacyjnego będą wysyłane kolejne znaki tworzące napis *konami*. Po ponownym wprowadzeniu kodu, klawiatura zacznie znów działać poprawnie.

Po poprawnym napisaniu sterownika, udało nam się też skompilować go wraz z kernelem, jako wbudowany w jądro sterownik. Wykorzystaliśmy do tego dystrybucję Gentoo Linux, która nastawiona jest na samodzielną kompilację komponentów systemu przez użytkownika (source-based distribution).

2 Opis działania

2.1 Czym są sterowniki

Sterownik urządzenia to fragment programu odpowiadający za dane urządzenie i pośredniczący pomiędzy nim a resztą systemu komputerowego. Zwykle uabstrakcyjnia pewne cechy urządzenia, choć może jedynie zajmować się kwestiami uprawnień dostępu i udostępniać urządzenie bez żadnej ingerencji.

Często sterownik urządzenia jest odpowiedzialny za obsługę urządzeń, które fizycznie nie istnieją. W systemach operacyjnych Linux i Unix znajduje się wiele sterowników urządzeń wirtualnych.

1. sterownik urządzenia NULL (dostęp przez `/dev/null`) - abstrakcyjne urządzenie, do którego można zapisywać, co nie przynosi żadnego efektu;
2. sterownik urządzenia RANDOM (dostęp przez `/dev/random`) - dane odczytywane z tego urządzenia są losowe;
3. `/dev/zero` - nieskończone źródło znaków (o kodzie 0x00);
4. sterowniki systemów plików - tworzą abstrakcję nad sterownikami dysków, pozwalają na wygodny dostęp do danych bez znajomości sposobu ich zapisu

2.2 Podstawy działania modułów jądra Linux

Moduły to fragmenty kodu, które mogą być ładowane i "wyłączane" z kernela na żądanie. Rozszerzają możliwości systemu bez potrzeby rekompilowania kernela. Przykładem takiego modułu mogą być np. sterowniki do urządzenia, które zezwalają kernelowi na dostęp do sprzętu podłączonego do systemu. Bez modułów kernel musiałby mieć formę monolitu, a nowe funkcjonalności musiałyby być dodawane bezpośrednio do obrazu kernela.

Możemy sprawdzić, które moduły są załadowane do kernela za pomocą komendy `lsmod`, która wyświetla te informacje na podstawie pliku `/proc/modules`.

2.3 Moduły a sterowniki

Sterowniki nie muszą być modułami kernela - możliwe jest wbudowanie ich bezpośrednio w kernel podczas kompilacji (2). Sterownik wbudowany w jądro systemu zawsze jest aktywny i nie ma możliwości wyłączenia go, bez uciekania się do rekompilacji kernela.

```

<*> The Extended 4 (ext4) filesystem
[*]   Use ext4 for ext2 file systems
[*]   Ext4 POSIX Access Control Lists
[*]   Ext4 Security Labels
[ ]   Ext4 debugging support
[ ]   JBD2 (ext4) debugging support
<M> Reiserfs support
[ ]   Enable reiserfs debug mode

```

Rysunek 1: Zrzut ekranu konfiguracji kernela, wybór funkcjonalności, które zostaną skompilowane. Wsparcie dla systemu plików *ext4* oznaczone jest symbolem gwiazdki - zostanie wbudowane w kernel. Wsparcie dla *ReiserFS* oznaczone jest znakiem **M** - będzie dostępne jako moduł.

2.4 Urządzenia HID

HID (*ang. Human Interface Device*) - jest to metoda podczas której człowiek wchodzi w interakcję z systemem przy pomocy poprzez wprowadzanie danych lub zwracanie danych wyjściowych. Istnieje wiele urządzeń HID, są to na przykład: klawiatury, myszy, joysticks, głośniki, kamery. Wszystkie urządzenia zapewniające interfejs pomiędzy użytkownikiem a maszynami komputerowymi są uważane za identyfikatory HID.

3 Rozwiązanie problemu

3.1 Język i środowisko

Kernel Linuxa napisany jest w języku C, zdecydowana większość dokumentacji zakłada, że developer chce wykorzystywać język C.

Możliwe jest pisanie modułów w innych językach, jednak nie jest to rozpowszechniona ani wspierana oficjalnie praktyka (jest to jednak możliwe - przykładowo, istnieje eksperymentalny framework pozwalający pisać moduły w języku Rust - <https://github.com/fishinabarrel/linux-kernel-module-rust>).

Ponieważ nikt z drużyny projektowej nie miał doświadczenia z tak niskopoziomym programowaniem w Linuxie, postanowiliśmy wykorzystać standardową dokumentację i język C.

3.2 Proces kompilacji modułu

Proces kompilacji modułów kernela przebiega inaczej niż kompilacja standardowych programów (1) - wymagana jest konfiguracja wielu parametrów kompilacji, aby poprawnie zbudować moduł.

Aby ułatwić ten proces, powstało narzędzie *Kbuild* - Kernel Build System - system organizujący proces budowania kernela i modułów (3). Dzięki niemu, wystarczy zainstalować nagłówki do kernela swojej dystrybucji (zazwyczaj dostępne poprzez pakiet `linux-headers`) oraz wykorzystać domyślny `Makefile` z dokumentacji *Kbuild*, aby zbudować prosty sterownik.

Domyślny Makefile:

```
1 obj-m += example_module.o
2
3 all:
4     make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
5
6 clean:
7     make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

3.2.1 Ładowanie modułu i weryfikacja działania

W systemie Linux dostępny jest szereg poleceń, pozwalający na kontrolę nad załadowanymi modułami i sprawdzenie ich stanu:

- `lsmod` - drukuje wszystkie załadowane do kernela moduły. Wykorzystywany najczęściej w połączeniu z `grep`, w celu znalezienia informacji o porządnym module.
- `insmod` - ładuje moduł podany jako argument wywołania.
- `rmmod` - odłącza moduł podany jako argument wywołania.
- `lsusb -t` - domyślnie, `lsusb` drukuje podłączone do systemu urządzenia USB. Flaga `-t` pozwala też dowiedzieć się, jaki sterownik zarządza danym urządzeniem. Wykorzystywaliśmy to polecenie, aby upewnić się czy system używa naszego sterownika.
- `lspci -k` - podobnie jak w przypadku `lsusb`, polecenie `lspci` domyślnie drukuje urządzenia podłączone do systemu, tym razem przez interfejs PCI. Flaga `-k` powoduje wydrukowanie też sterowników, które odpowiadają za podłączone urządzenia `dmesg -w` - odczyt logów systemowych. Flaga `-w`

powoduje pracę komendy w trybie ciągłym, nowe logi będą pojawiały się na ekranie w czasie rzeczywistym.

Przykład wykorzystania komendy `lsusb`. Można zauważyć urządzenia typu HID obsługiwane przez sterownik `usbhid`:

```
~ lsusb -t
/: Bus 02.Port 1: Dev 1, Class=root_hub, Driver=xhci_hcd/6p, 5000M
/: Bus 01.Port 1: Dev 1, Class=root_hub, Driver=xhci_hcd/12p, 480M
   |__ Port 1: Dev 5, If 1, Class=Human Interface Device, Driver=usbhid, 1.5M
   |__ Port 1: Dev 5, If 0, Class=Human Interface Device, Driver=usbhid, 1.5M
   |__ Port 5: Dev 2, If 0, Class=Chip/SmartCard, Driver=, 12M
   |__ Port 7: Dev 3, If 0, Class=Wireless, Driver=btusb, 12M
   |__ Port 7: Dev 3, If 1, Class=Wireless, Driver=btusb, 12M
   |__ Port 8: Dev 4, If 1, Class=Video, Driver=uvcdvideo, 480M
   |__ Port 8: Dev 4, If 0, Class=Video, Driver=uvcdvideo, 480M
```

3.3 Struktura programu

3.3.1 Ułatwienia, jakie Linux oferuje programistom

Zdecydowaliśmy się wykorzystać obecny w kodzie źródłowym Linuxa sterownik do klawiatury USB (4) i rozbudować go o funkcję rozpoznawania i reagowania na *Konami Code*. Kernel Linuxa jest otwarto-źródłowy, wystarczyło więc odnaleźć sterownik do klawiatury (`usbkbd.c`) w repozytorium.

Ponieważ urządzenia USB są szeroko rozpowszechnione na rynku, kod źródłowy Linuxa zawiera gotowe makra i uproszczenia(5), wspomagające programistów piszących sterowniki do urządzeń USB.

Przykład wykorzystania makra `module_usb_driver`. Wykorzystany zamiast standardowej funkcji `module_init`. pozwala od razu zarejestrować callbacki wymagane do pracy z urządzeniem USB.

```
1 static struct usb_driver usb_kbd_driver = {
2     .name = "oiak_usb",
3     .probe = usb_kbd_probe,
4     .disconnect = usb_kbd_disconnect,
5     .id_table = usb_kbd_id_table,
6 };
7
8 module_usb_driver(usb_kbd_driver);
```

3.3.2 Rozpoznawanie i reagowanie na *Konami Code*

Dane z urządzenia dostarczane są do sterownika poprzez wywoływanie funkcji, obsługującej pojawienie się nowych informacji. Funkcja `usb_fill_int_urb` pozwala na zarejestrowanie wybranej funkcji sterownika do obsługi nadchodzących danych. W naszym sterowniku jest to funkcja `usb_kbd_irq` (co można przetłumaczyć na "przerwanie z klawiatury USB").

Funkcja przyjmuje wskaźnik na strukturę typu `urb` (*USB Request Block*), służącą do wymiany informacji między urządzeniami podłączonymi przez interfejs USB (7).

Pojedynczy pakiet danych z klawiatury zawiera więcej informacji, niż sam kod naciśniętego guzika. Klawiatura informuje system, czy guzik został przyciśnięty czy zwolniony, wysyłane są też informacje dotyczące stanu przycisków-modyfikatorów - czy wciśnięty jest Alt, Ctrl lub Shift. Drugi bajt danych otrzymanych z klawiatury, zawiera kod wciśniętego klawisza. Wykorzystując tabelę kodów HID (6), możemy zakodować w nim *Konami Code*.

Zmienne zdefiniowane globalnie w kodzie modułu:

```
1 int current_konami_code_index = 0;
2 unsigned char konami_code[10] = { 82, 82, 81, 81, 80, 79, 80,
   ↪ 79, 5, 4 };
3
4 bool konami_mode_on = false;
5
6 // Napis 'konami'
7 unsigned char fake_input[6] = { 14, 18, 17, 4, 16, 12 };
8 int fake_input_index = 0;
```

Kiedy wiemy już, gdzie szukać nadchodzących danych, oraz jaki będą miały format, detekcja kodu jest trywialnym problemem. Jeśli użytkownik wciśnie guzik wprowadzający pierwszy znak kodu, inkrementowany jest licznik wskazujący na elementy tablicy z następnymi znakami kodu. Jeśli użytkownik wybierze teraz znak inny niż kolejny w kodzie, licznik jest resetowany. Gdy licznik dojdzie do 10, użytkownik wpisał cały *Konami Code*, i klawiatura przechodzi w tryb wypisywania napisu *konami*, niezależnie od wciskanych przycisków, aż do ponownego wprowadzenia kodu.

Fragменты функции usb_kbd_irq, obsługujące wykrywanie i reakcję na Konami Code:

```
1 unsigned char key_code = kbd->new[2];
2 if (key_code != 0)
3 {
4     // Without \n, printk is not forced to flush,
5     // And the logs are delayed
6     // printk(KERN_INFO "%d\n", key_code);
7     if (konami_code[current_konami_code_index] == key_code) {
8         printk(
9             KERN_INFO
10             "Progress %d\n",
11             ++current_konami_code_index
12         );
13
14         // Aktywacja trybu Konami
15         if(current_konami_code_index == 10) {
16             printk(KERN_INFO "Konami mode toggle\n");
17             current_konami_code_index = 0;
18
19             konami_mode_on = !konami_mode_on;
20
21             }
22         } else {
23             printk(KERN_INFO "Reset\n");
24             current_konami_code_index = 0;
25         }
26     }
27
28     /*
29     Jeśli aktywny jest tryb Konami,
30     kody wciskanych guzików są podmieniane.
31     */
32     if (konami_mode_on) {
33
34         kbd->new[2] = fake_input[fake_input_index++ % 6];
35
36     }
```

3.4 Obejście standardowego sterownika klawiatury USB

3.4.1 Problem z domyślnym sterownikiem urządzeń HID

Dzięki systemowi *Kbuild*, skompilowanie modułu z domyślnym sterownikiem klawiatury sprowadza się do przekopiowania kodu sterownika do katalogu projektu i wykonania polecenia `make`. Po załadowaniu modułu okazuje się jednak, że nie przejmuje on "steru" nad podłączoną do komputera klawiaturą USB - to miejsce zajmuje domyślnie sterownik `usbhid`. Zanim nasz sterownik będzie mógł podłączyć się do klawiatury, konieczne jest odebranie modułowi `usbhid` priorytetu w dostępie do urządzeń USB.

3.4.2 Modyfikacja konfiguracji `udev`

Od kernela 2.6, komponent `udev` pełni rolę menadżera urządzeń w systemie Linux. Administrator systemu może wprowadzać własne zasady, których `udev` będzie przestrzegał zarządzając urządzeniami. Aby tego dokonać, należy dodać pliki z treścią zasad do katalogu `/etc/udev/rules.d`.

Aby odebrać sterownikowi `usbhid` priorytet w zarządzaniu urządzeniami USB HID, należy wykonać następującą komendę, która dopisze do plików konfiguracyjnych `udev` nową zasadę.

```
echo 'SUBSYSTEMS=="usb", DRIVERS=="usbhid", ACTION=="add",  
    ↪ ATTR{authorized}="0" ' >  
    ↪ /etc/udev/rules.d/99-disable-hid.rules
```

Po dopisaniu tej zasady i zresetowaniu komputera, nasz sterownik zaczął działać poprawnie - mogliśmy zweryfikować czy zarządza klawiaturą, wykorzystując polecenie `lsusb -t` lub monitorując jego logi za pomocą `dmesg -w`.

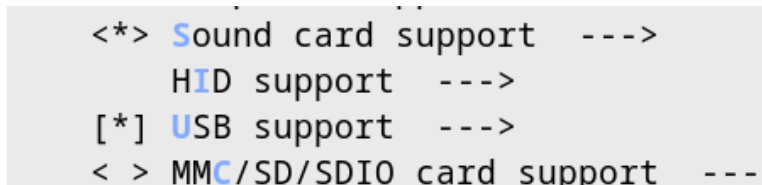
3.5 Kompilacja jądra Linuxa, z naszym sterownikiem wbudowanym w jądro

W ramach dodatkowego wyzwania, postanowiliśmy spróbować skompilować kernel Linuxa, który zawierałby nasz sterownik, tym razem nie jako moduł, lecz jako sterownik wbudowany w kernel.

Dystrybucja Gentoo Linux pozwala w prosty sposób pobrać, skompilować i zainstalować jądro, pozwalając użytkownikom na szerokie i niskopoziomowe modyfikacje systemu.

3.5.1 Wymagana konfiguracja kernela

Aby kernel obsługiwał urządzenia USB HID, potrzebne są sterowniki do hubów USB oraz sterowniki do obsługi urządzeń HID. Te sterowniki są załączone w domyślnej konfiguracji kernela, więc zazwyczaj po prostu działają. Więcej informacji dostępnych jest tutaj: <https://wiki.gentoo.org/wiki/USB/Guide>.



```
<*> Sound card support --->
      HID support --->
[*] USB support --->
< > MMC/SD/SDIO card support ----
```

Rysunek 2: Zrzut ekranu konfiguracji kernela, sekcja Device Drivers - widoczne opcje obsługi USB i HID.

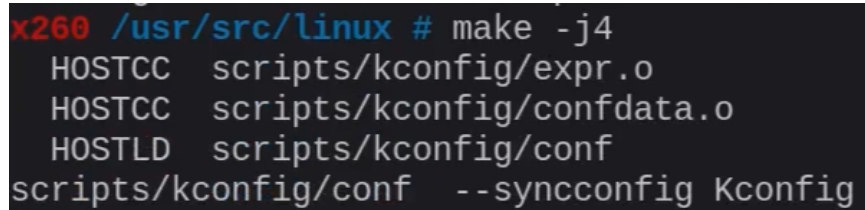
Pozostała część konfiguracji, kompilacji i instalacji kernela przebiega zgodnie ze standardowym procesem, opisanym na: https://wiki.gentoo.org/wiki/Complete_Handbook/Building_the_Linux_kernel.

3.5.2 Kompilacja sterownika `usbkbd.c` - ryzykowna sztuczka

Wprowadziliśmy kod naszego sterownika w miejsce oryginalnego `usbkbd.c`, zmieniając nazwę sterownika na `oiak_usb`. Okazuje się jednak, że moduł `usbkbd.c` nie jest używany przy kompilacji na komputery klasy PC, lecz pozostaje w kodzie źródłowym na potrzeby prostych urządzeń embedded. Wcześniej łądownaliśmy go ręcznie do systemu jako moduł, teraz okazało się że zwyczajnie nie jest on kompilowany wraz z domyślnymi modułami do urządzeń USB (można to sprawdzić

obserwując proces kompilacji, choć znacznie łatwiej jest pozostawić niekompilowalny kod w pliku `usbkbd.c` - kernel i tak zostanie zbudowany bez błędów).

Przeglądając pliki sterowników HID (`/drivers/hid/usbhid`), okazuje się, że podłączenie pliku `usbkbd.c` do kodu kernela w trakcie kompilacji odbywa się poprzez zmienną konfiguracyjną `CONFIG_USB_KBD`. Nie udało się odnaleźć tej zmiennej w domyślnym menu konfiguracji kernela (dostępnym przez wykonanie komendy `make menuconfig` w katalogu z kerneliem). Zmienna ta najpewniej jest przestarzała, lub też zrozumienie sposobu jej poprawnej aktywacji wymaga głębszego rozeznania w temacie, zdecydowaliśmy się więc na improwizację - w `Makefile`'u odpowiedzialnym za kompilację sterowników `usbhid`, podmieniliśmy ścieżkę do sterownika `hid-core` na ścieżkę do naszego sterownika. Kernel skompilował się poprawnie, nasz sterownik został poprawnie wbudowany w kod źródłowy - teraz domyślnie obsługuje klawiatury USB podłączone do komputera.



```
x260 /usr/src/linux # make -j4
HOSTCC scripts/kconfig/expr.o
HOSTCC scripts/kconfig/confdata.o
HOSTLD scripts/kconfig/conf
scripts/kconfig/conf --syncconfig Kconfig
```

Rysunek 3: Kompilacja kernela po nałożeniu patcha

```
wint3rmute@x260 ~ $ neofetch
      _/oyddmdhs+:.
    -odNNNNNNNNNNmhy+-`
    -yNNNNNNNNNNNNNNNNmdhy+-
    `omNNNNNNNNNNNNmdmmddhhy/`
    omNNNNNNNNNNNNhhyyyohmddhhhd`
    .ydNNNNNNNNNNdhs++so/smdddhhhd+`
    oyhdmNNNNNNNNdyooydmdddhhyhNd.
    :oyhhdNNNNNNNNNNmmdddhhyymMh
    .:~sydNNNNNNNNNNmmdddhhyhmMny
    /mNNNNNNNNNNmmdddhhyhmMNs:
    `oNNNNNNNNNNmmdddhdmMNs+`
    `sNNNNNNNNNNmmdddmMNs/.
    /NNNNNNNNNNmmdddmMNdso:~
    +MMNNNNNNNNNNmmddmMNdso/-
    yMMNNNNNNNNNNmmMhs+/-`
    /hMMNNNNNNNNMdhhs++/-`
    `ohdmmddhys+++/:.
    `~////////:--

wint3rmute@x260 ~ $ lsusb -t | grep oiak_usb
    |__ Port 1: Dev 5, If 0, Class=Human Interface Device, Driver=oiak_usb, 1.5M

wint3rmute@x260 ~ $

[ 205.336548] Progress 4
[ 205.504550] Progress 5
[ 205.672561] Progress 6
[ 205.872552] Progress 7
[ 206.000552] Progress 8
[ 206.216574] Progress 9
[ 206.528555] Progress 10
[ 206.528556] Konami mode toggle
[ 207.560573] Progress 1
[ 207.752558] Progress 2
[ 207.952560] Progress 3
[ 208.120579] Progress 4
[ 208.320567] Progress 5
[ 208.496563] Progress 6
[ 208.664585] Progress 7
[ 208.824526] Progress 8
[ 209.008565] Progress 9
[ 209.344567] Progress 10
[ 209.344568] Konami mode toggle
[ 213.157134] usb 1-1: USB disconnect, device number 8
[ 226.477255] usb 1-1: new low-speed USB device number 9 using xhci_hcd
[ 226.609874] usb 1-1: New USB device found, idVendor=1c4f, idProduct=0002, bcdDevice= 5.50
[ 226.609875] usb 1-1: New USB device strings: Mfr=1, Product=2, SerialNumber=0
[ 226.609876] usb 1-1: Product: USB Keyboard
[ 226.609877] usb 1-1: Manufacturer: SIGMACHIP
[ 226.611257] input: SIGMACHIP USB Keyboard as /devices/pci0000:00/0000:00:14.0/usb1/1-1/1-1:1.0/input/input12
[ 226.722353] elogind-daemon[10588]: New session c10 of user wint3rmute.
[ 226.733310] elogind-daemon[10588]: Removed session c10.
```

Rysunek 4: Zrzut ekranu w trakcie testów, po skompilowaniu kernela z wbudowanym sterownikiem i podłączeniu klawiatury. Zmieniona została nazwa kernela, widać też że nasz sterownik jest podłączony do klawiatury. W oknie niżej znajduje się output komendy `dmesg -w`, wyświetlającej logi sterownika.

3.6 Spakowanie sterownika w kernel patch

Popularną metodą na dystrybucję drobnych modyfikacji do dużych projektów jest wykorzystanie programu patch oraz diff. W trakcie projektu, zmodyfikowaliśmy w kernelu pliki `usbkbd.c` oraz `Makefile`. Wykorzystując gita, można utworzyć plik `.diff`, pozwalający na zapisanie dokonanych przez nas zmian i odtworzenie ich na innej instalacji/innym kernelu.

Programy patch i diff są wbudowane w git, co pozwala na łatwiejsze generowanie i nakładanie patchy. Warto jednak wspomnieć, że te programy są też dostępne w standardowym pakiecie *GNU Software* (9).

Aby wytworzyć plik `.diff`, należy wykonać komendę `git diff` w repozytorium ze zmodyfikowanym kodem źródłowym kernela i przekierować jej wynik do pliku:

```
git diff > /somewhere/only/we/know/konami_kbd.diff
```

Aby nałożyć patch na niezmodyfikowany kernel, należy w roocie jego repozytorium wykonać:

```
git apply /path/to/konami_kbd.diff
```

Następnie wystarczy skompilować i zainstalować kernel.

4 Wnioski i obserwacje

Najtrudniejszą częścią projektu było wdrożenie się w nie zawsze przejrzystą dokumentację Linuxa. Wiele informacji było rozsianych po starych forach, strony nie zmieniane od prawie dekady stanowiły normę. Łatwo też było trafić na poradniki z przestarzałymi informacjami.

Ciężko też przyjąć na raz natłok bardzo wielu nowych terminów, nie zawsze prostych do conceptualnego przyswojenia. Zalew trzyliterowymi skrótami - HID, URB, IRQ, PCI - potrafił naprawdę utrudnić zbudowanie pełnego obrazu sytuacji.

Finalnie udało się zrealizować założenia projektu, niskopoziomowe odchłanianie Linuxa okazały się być nieco mniej straszne, niż się wydają, choć z pewnością czeka w nich jeszcze wiele wyzwań. Dobrze natomiast posiadać ogólny ogląd rzeczy, może też odrobinę większą pewność siebie.

5 Literatura

Literatura

- [1] Peter Jay Salzman, Michael Burianm, Ori Pomerantz: *The Linux Kernel Module Programming Guide* by. <http://tldp.org/LDP/lkmpg/2.6/html/lkmpg.html>
- [2] Gentoo Foundation: *Gentoo Kernel Configuration Guide* https://wiki.gentoo.org/wiki/Kernel/Gentoo_Kernel_Configuration_Guide
- [3] Linux Foundation: *Kernel Build System Documentation* <https://www.kernel.org/doc/html/latest/kbuild/kbuild.html>
- [4] Kod źródłowy kernela Linux: *usbkbd.c* <https://github.com/torvalds/linux/blob/master/drivers/hid/usbhid/usbkbd.c>
- [5] *Writing USB Device Drivers* https://kernel.readthedocs.io/en/sphinx-samples/writing_usb_driver.html
- [6] Materiały użytkownika MightyPork (PotężnaWieprzowina), dostępne na serwisie GitHub: *USB HID Keyboard scan codes* <https://gist.github.com/MightyPork/6da26e382a7ad91b5496ee55fdc73db2>
- [7] Linux Foundation: *USB Request Block* <https://www.kernel.org/doc/html/latest/driver-api/usb/URB.html>
- [8] Forum Stack Overflow: *Prevent usbhid from autoloading when USB HID device is plugged in* <https://stackoverflow.com/questions/38786343/prevent-usbhid-from-autoloading-when-usb-hid-device-is-plugged-in>
- [9] Free Software Foundation, Inc: *GNU Software* <https://www.gnu.org/software/software.html>
- [10] Wikipedia: *Human Interface Device* https://en.wikipedia.org/wiki/Human_interface_device