Sterownik typu filtr dla urządzenia typu Serial

Oprogramowanie systemowe

Karol Pergoł 184876 – GR.iV WETI Informatyka

# Wstęp

Podczas zajęć projektowych z przedmiotu Oprogramowanie Systemowe zrealizowałem projekt sterownika typu KMDF, będącego filtrem dla urządzeń typu Serial – urządzeń portu szeregowego. We wstępnych założeniach celem była realizacja jedynie przechwytywania odczytów (IRP\_MJ\_READ) oraz zapisów (IRP\_MJ\_WRITE) pomiędzy aplikacjami użytkownika, a urządzeniem. W toku realizacji zdania postanowiłem rozszerzyć realizacje o wykorzystanie aplikacji użytkownika do sterowania sterownikiem oraz wyświetlanie kluczowych danych pochodzących ze sterownika. Komunikacja pomiędzy sterownikiem, a aplikacją użytkownika została zrealizowana z wykorzystaniem komunikacji IOCTL (input/output control). Dodatkowo do sterownika dodana została funkcjonalność obsługi IRP\_MJ\_DEVICE\_CONTROL, pochodzących z zewnętrznego urządzenia Serial w celu możliwości odczytu komend konfiguracyjnych np. ustawienia baudrate.

# Punkt wejścia dla sterownika

Po załadowaniu działanie sterownika rozpoczyna się w pliku code.c, gdzie znajduje się funkcja DriverEntry, będąca punktem wejściem dla sterownika.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Blok kodu 1.

Powyższy blok kodu odpowiada za przypisanie funkcji UnloadDriver do wywołania podczas rozładowywania sterownika. Funkcja ta zapewni możliwość bezpiecznego usunięcia/wyłączenia sterownika oraz usunięcie obiektów (urządzeń) stworzonych w trakcie funkcjonowania sterownika. Następnie przypisujemy do wszystkich funkcji obsługi IRP, funkcję distDipatchPass, zawartość tej funkcji przekazuje IRP do kolejnego urządzenia w stosie urządzeń. Dzięki temu nieobsługiwane przez nas funkcję będą przezroczyste dla urządzenia oraz aplikacji.

Kolejno wywoływana jest metoda odpowiadająca za utworzenie oraz stworzenie linku symbolicznego dla urządzenia odpowiadającego za komunikację IOCTL z aplikacją kliencką.   
  
Obraz zawierający tekst

Opis wygenerowany automatycznie

***Adnotacja 1.***

*Ze względu na wykorzystywanie dwóch urządzeń, konieczne było zastosowanie funkcji odpowiadających za dystrybucję IRP pomiędzy tymi urządzeniami. W celu zachowania przejrzystości przyjęta została poniższa nomenklatura:*

*Funkcje rozpoczynające się: dist – funkcja dystrybucji, jej zawartość rozdziela IRP pomiędzy urządzeniami na podstawie otrzymanego wskaźnika urządzenia.*

*Funkcje rozpoczynające się: ext – funkcje dotyczące zewnętrznego urządzenie typu Serial*

*Funkcje rozpoczynające się: conn – funkcje dotyczące urządzenia komunikacyjnego z aplikacją klienta (dodatkowo obsługa kodów IOCTL widocznych w pliku connect.h)*

Blok kodu 2.

Dalej wykonywany jest kod odpowiadający za przypisanie odpowiednich operacji do MajorFunction sterownika. Przypisywanie następuje według reguł umieszczonych we fragmencie Adnotacja 1.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Blok kodu 3.

Nadpisywane są funkcję obsługi IRP\_MJ\_CREATE,IRP\_MJ\_CLOSE,IRP\_MJ\_DEVICE\_CONTROL, IRP\_MJ\_READ,IRP\_MJ\_WRITE. To już koniec funkcji DriverEntry.

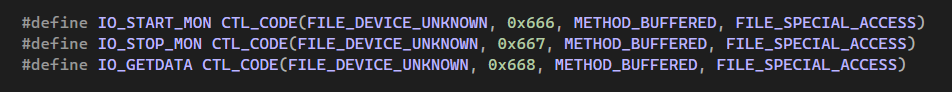
# Obsługa komunikacji z aplikacją kliencką

W celu realizacji komunikacji z aplikacją kliencką, obsługującą ten sterownik zdecydowałem się na wykorzystanie komunikacji IOCTL – I/O Control Code. Komunikacja występuje z wykorzystaniem urządzenia utworzonego w DriverEntry (opisanego w paragrafie 2.). Realizacja tej funkcjonalności znajduje się w plikach nazwanych connect.

Wykorzystywanymi operacjami są:

* connCreateCall – odpowiada za obsługę IRP rozpoczynającego połączenie
* connCloseCall – odpowiada za obsługę IRP kończącego połączenie
* connCreateAttach – odpowiada za utworzenie obiektu urządzenia komunikacyjnego typu FILE\_DEVICE\_UNKNOWN (wyjaśniona wcześniej).
* connDeviceControl – odpowiadającego za obsługę kodów kontrolnych, które zostaną opisane dokładniej poniżej.

Funkcja connDeviceControl, w pierwszej kolejności po wywołaniu odczytuje kod kontroli urządzenia, który zapisywany jest w zmiennej ControlCode. Obsługiwanymi kodami są te umieszczone w pliku nagłówkowym connnect.h.



Blok kodu 4. Obsługiwane kody

Po przesłaniu kodu IO\_START\_MON wraz z którym przesłana jest ścieżka dostępu do jednego z podłączonych urządzeń portu szeregowego, następuje dołączenie sterownika do urządzenia zewnętrznego Serial (utworzenie nowego urządzenia w stosie urządzeń) - funkcja extAttachDeviceByPath.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Blok kodu 5.

Po przesłaniu kodu IO\_STOP\_MON następuję odłączenie sterownika (usunięcie urządzenia ze stosu urządzeń), które wcześniej zostało wybrane z wykorzystaniem kodu IO\_START\_MON.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Blok kodu 6.

Najważniejszym jest jednak kod IO\_GETDATA, jest on odpowiedzialny za pobieranie danych, które zbiera filter driver oraz przekazywanie ich do aplikacji klienckiej. Aby uniknąć problemów z wywoływaniem funkcji z nieprawidłowym poziomem IRQL, zastosowałem prostą implementacji kolejki. Filter driver zapisuje wszystkie informacje przesyłane z/do zewnętrznego urządzenia portu szeregowego i zapisuje je w odpowiednim formacie do kolejki. Gdy aplikacja kliencka wysyła kod IO\_GETDATA, pobierane są informacje z kolejki (o ile są dostępne) i przekazywane do aplikacji klienckiej. Realizacja tego mechanizmu zaprezentowana została poniżej. W przypadku braku danych odsyłana jest odpowiednia informacja o zaistniałej sytuacji.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Blok kodu 7.  
 Obraz zawierający tekst

Opis wygenerowany automatycznie

Blok kodu 8. Struktura przekazywanych danych

Aplikacja kliencka stworzona została w języku C++ w trybie tekstowym. Poniżej prezentacja działania.  
Obraz zawierający tekst

Opis wygenerowany automatycznie Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 1. Wybór urządzenia Rysunek 2. Przebieg komunikacji

# Obsługa zewnętrznego urządzenia portu szeregowego

Ze względu na złożoność problemu obsługę urządzenia typu Serial postanowiłem rozdzielić pomiędzy kilka plików.

* ext\_dev – obsługa dołączania oraz odłączania sterownika do wybranego urządzenia Serial.sys
* ext\_ioctl – obsługa kodów IOCTL z/do urządzenia Serial oraz operacji odczytu i zapisu IRP\_MJ\_READ, IRP\_MJ\_WRITE.
* data\_queue – kolejka przechowująca zebrane dane urządzenia Serial

Gdy w aplikacji klienckiej zostanie wybrane jedno z podłączonych urządzeń typu Serial, następuje wywołanie funkcji extAttachDeviceByPath. Teraz prześledzę działanie tej funkcji.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Blok kodu 9. extAttachDevice Pt.1

Jako argument funkcja ta przyjmuje wskaźnik na obiekt sterownika oraz zmienną targetDevice typu UNICODE\_STRING, będzie ona zawierała ścieżkę dostępu do urządzenia Serial device przykładowo „\Device\Serial2”. Następnie wywołujemy funkcję IoCreateDevice – tworzy ona obiekt urządzenia, który może być następnie wykorzystywany przez sterownik. Jako rozszerzenie urządzenia wykorzystujemy wcześniej utworzony DEVICE\_EXTENSION. Zawiera on wskaźnik do kolejnego urządzenia w stosie urządzeń oraz IO\_REMOVE\_LOCK, zabezpieczający przed niepożądanym usunięciem urządzenia w momencie, gdy jest jeszcze użytkowane i takie usunięcie mogło by spowodować błąd.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Blok kodu 10. DEVICE\_EXTENSION

Obraz zawierający tekst

Opis wygenerowany automatycznie

Blok kodu 11. extAttachDeviceByPath Pt.2

W dalszej części funkcji extAttachDeviceByPath, ustawiane są flagi dla nowego utworzonego obiektu urządzenia. Ustawienie flagi DO\_BUFFERED\_IO odpowiada za ustawienie buforowania przy wykorzystaniu IRP. W takiej sytuacji występują dwa bufory jeden po stronie użytkownika drugi po stronie aplikacji. Usprawnia to przesył małych paczek danych ponieważ nie ma wtedy konieczności blokowania całej fizycznej strony pamięci. Wykorzystanie Wyłączamy także wstępną inicjalizację urządzenia.

Gdy sterownik zostanie już poprawnie podłączony do urządzenia portu szeregowego, możliwa jest obsługa operacji IRP\_MJ\_READ oraz IRP\_MJ\_WRITE jak i również IRP\_MJ\_DEVICE\_CONTROL.

### Obsługa IRP\_MJ\_DEVICE\_CONTROL

W pierwszej kolejności omówimy obsługę IRP\_MJ\_DEVICE\_CONTROL. W momencie otrzymania IRP - IO request packet trafia ona do funkcji dyspozytora distDeviceControl umieszczonej w pliku dist\_ioctl.c.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Blok kodu 12. Dyspozytor

Dyspozytor sprawdza do kogo skierowane jest IRP, jeśli jest to urządzenie zewnętrzne portu szeregowego wywołuje on funkcję extDeviceControl.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Blok kodu 13. extDeviceControl

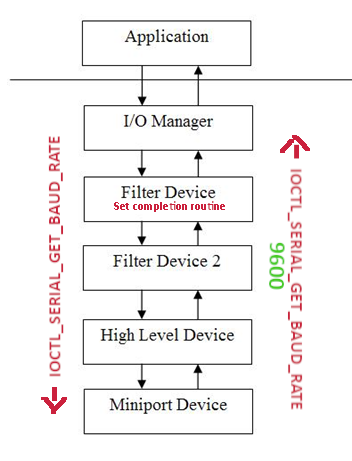
W extDeviceControl pobieramy rozszerzenie aktualnie używanego urządzenia. Następnie żądamy zablokowania możliwości usunięcia urządzenia z wykorzystaniem IO\_REMOVE\_LOCK. Jeśli żądanie zakończy się sukcesem możemy rozpocząć odczytywanie kodu kontroli, w przeciwnym wypadku pomijamy obsługę.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Blok kodu 14. IOCTL dispatcher

IoControlCode czyli kod kontroli odczytujemy ze stosu. Aktualne położenie stosu pobierane jest przez IoGetCurrentIrpStackLocation(Irp). Dalej znajduje się rozległa instrukcja typu switch, która dla każdego możliwego kodu kontroli wykonuje odpowiednią funkcję. W mojej realizacji istnieją dwie możliwości nazwa kodu kontroli rozpoczyna się od IOCTL\_SERIAL\_GET\_..., oznacza żądanie wartości od urządzenia portu szeregowego, może to być dla przykładu baudrate lub wartość linii kontrolnej. W takiej sytuacji ustawiamy rutynę kompletującą IoSetCompletionRoutine, która zbierze dane gdy IRP będzie wracało „od urządzenia” do warstwy aplikacji. Schemat takiego działania dobrze przedstawia grafika poniżej. „Filter driver” ustawia rutynę kompletującą dla IOCTL\_GET\_BAUD\_RATE, gdy dane wracają z urządzenia „zbierana” jest wartość 9600.



Rysunek 2. Completion Routine

Gdy dane są kompletowane wywoływana jest funkcja completePendedIrp, która odpowiada za pobranie tych danych ze stosu oraz przekazanie ich do kolejki – funkcja collectData. Po wykonaniu wszystkich niezbędnych kroków zwalniany jest remove lock. Działanie funkcji collectData będzie opisane w dalszej części raportu.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Blok kodu 15. completePendedIrp

W sytuacji gdy nazwa kodu kontroli rozpoczyna się od IOCTL\_SERIAL\_SET\_..., nie ma konieczności ustawiania rutyny kompletującej. Dane zbierane są bezpośrednio z buforu i przekazywane do kolejki jak pokazano we fragmencie kodu poniżej.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Blok kodu 16. extHandleSetIoctl

### Obsługa IRP\_MJ\_READ

Procedura odczytu (extDeviceRead) przebiega analogicznie do sytuacji gdy odczytywana jest wartość z wykorzystaniem IOCTL\_SERIAL\_GET\_..., ustawiana jest rutyna kompletująca, która odczytuje dane z przesyłane w kierunku od urządzenia do warstwy aplikacji użytkownika. Ponieważ operujemy bezpośrednio na urządzeniu konieczne jest wykorzystanie remove lock.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Blok kodu 17. extDeviceRead

### Obsługa IRP\_MJ\_WRITE

Procedura zapisu (extDeviceWrite) polega na odczytaniu danych z buforu oraz wstawienia ich do kolejki. Analogicznie do wcześniejszych metod wykorzystujemy remove lock, w celu uniknięcia ryzyka usunięcia urządzenia podczas wykonywania operacji. Po odczytaniu danych z buforu, IRP przekazujemy do kolejnego urządzenia w stosie wykorzystując metodę IoCallDriver.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Blok kodu 18. extDeviceWrite

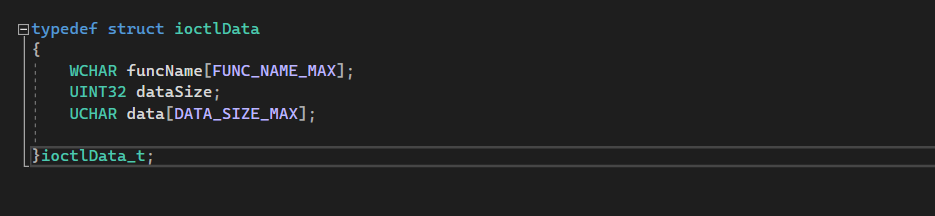
Możemy zauważyć, iż w sytuacji w której dane mają zostać przekazane do kolejki wywoływana jest funkcja collectData. Odpowiada ona za wstawienie danych w odpowiednim formacie ioctlData\_t.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Blok kodu 19. collectData

Jako pierwszy argument funkcji przyjmowana jest nazwa operacji, dalej rozmiar buforu danych oraz wskaźnik na tablicę danych. Tworzymy nowy obiekt o nazwie dataPack, który wypełniamy danymi skopiowanymi z buforu. Następnie dane wstawiane są do kolejki.



Blok kodu 20. Format dataPack

# Funkcje pomocnicze

### Makra wspomagające wpisywanie do konsoli

W wielu fragmentach kodu w projekcie można zauważyć wykorzystanie funkcji KPrint… są to makra nadpisujące funkcję DbgPrintEx, odpowiadająca za wypisywanie danych do debuggera systemu np. WinDbg. Dzięki zastosowaniu makr możliwe było zachowanie większego uporządkowania wypisywanych danych i zmniejszenie ilość używanego kodu. Co więcej gdy zdecyduję, iż makra nie mają być już wykonywane wystarczy ich odpowiedniego nadpisanie, bez konieczności mozolnej analizy kodu i usuwania linijka po linijce.

Obraz zawierający tekst, monitor, ekran, czarny

Opis wygenerowany automatycznie

Blok kodu 21. Zawartość pliku helpers.h

### Kolejka danych

Wartym opisania jest również implementacja kolejki przechowującej dane. Kolejka wykorzystuje funkcjonalność spin lock, który zmuszą cześć odczytującą oraz zapisującą do oczekiwania na możliwość dostępu do danych w nieskończonej pętli. Kolejnym zaimplementowanym zabezpieczeniem jest możliwość wywołania funkcji dodawania do kolejki wyłącznie z poziomu DISPATCH\_LEVEL.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Blok kodu 22. Funkcja inicjalizująca kolejkę

Obraz zawierający tekst

Opis wygenerowany automatycznie

Blok kodu 23. Funkcja zdejmowania danych z kolejki

# Podsumowanie

Projekt starałem się wykonać jak najdokładniej, dbając również o zabezpieczenie ewentualnych przypadków brzegowych. Kontynuując dalszą pracę nad sterownikiem zdecydował bym się na optymalizację komunikacji, pomiędzy aplikacją kliencką, a sterownikiem – zastosował bym pełen model inverted call. Sterownik może być przydatnym narzędziem jako „sniffer” portów szeregowych, oraz inżynierii wstecznej urządzeń korzystających z portu szeregowego. Ewentualny dalszy rozwój sterownika będzie widoczny na github’ie: https://github.com/rcp444/SerialSniffer