



# Sterownik typu filtr dla urządzenia typu Serial

OPROGRAMOWANIE SYSTEMOWE

KAROL PERGOŁ 184876 – GR.IV WETI INFORMATYKA

## 1. 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 realizację 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.

## 2. Punkt wejścia dla sterownika

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

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT driverObject, IN PUNICODE_STRING regPath)
{
    NTSTATUS status;
    driverObject->DriverUnload = UnloadDriver;

    KPrintInfo("=== Loading driver ===\n");

    for (int i = 0; i <= IRP_MJ_MAXIMUM_FUNCTION; i++)
    {
        driverObject->MajorFunction[i] = distDispatchPass;
    }

    /* Creating connection device for user application */
    status = connCreateAttach(driverObject);
    if (!NT_SUCCESS(status))
    {
        KPrintErrN("Problem with conn device creation!");
    }
    else
    {
        KPrintInfoN("Successfully created conn device.");
    }
}
```

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 **distDispatchPass**, zawartość tej funkcji przekazuje IRP do kolejnego urządzenia w stosie urządzeń. Dzięki temu nieobsługiwane przez nas funkcje będą przezroczyste dla urządzenia oraz aplikacji.

---

### 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 urządzenia typu Serial

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

---

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

```
NTSTATUS connCreateAttach(PDRIVER_OBJECT driverObject)
{
    NTSTATUS status = STATUS_SUCCESS;
    RtlInitUnicodeString(&dev, L"\\Device\\serialmondev");
    RtlInitUnicodeString(&dos, L"\\DosDevices\\serialmondev");

    status = IoCreateDevice(driverObject, 0, &dev, FILE_DEVICE_UNKNOWN, FILE_DEVICE_SECURE_OPEN, FALSE, &g_connDevice);
    if (!NT_SUCCESS(status))
    {
        KPrintErrN("Cannot create conn device!");
        return status;
    }

    status = IoCreateSymbolicLink(&dos, &dev);
    if (!NT_SUCCESS(status))
    {
        KPrintErrN("Cannot create symbolic link for conn device!");
        return status;
    }

    return status;
}
```

Blok kodu 2.

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

```
/* Connection open and close */
driverObject->MajorFunction[IRP_MJ_CREATE] = distCreateCloseCall;
driverObject->MajorFunction[IRP_MJ_CLOSE] = distCreateCloseCall;
driverObject->MajorFunction[IRP_MJ_CLEANUP] = distDispatchPass;

/* External device controls filter and user app ioctl */
driverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = distDeviceControl;
driverObject->MajorFunction[IRP_MJ_READ] = extDeviceRead;
driverObject->MajorFunction[IRP_MJ_WRITE] = extDeviceWrite;

UNREFERENCED_PARAMETER(regPath);

return status;
}
```

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*.

### 3. 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 connect.h.

```
#define IO_START_MON CTL_CODE(FILE_DEVICE_UNKNOWN, 0x666, METHOD_BUFFERED, FILE_SPECIAL_ACCESS)
#define IO_STOP_MON CTL_CODE(FILE_DEVICE_UNKNOWN, 0x667, METHOD_BUFFERED, FILE_SPECIAL_ACCESS)
#define IO_GETDATA CTL_CODE(FILE_DEVICE_UNKNOWN, 0x668, METHOD_BUFFERED, FILE_SPECIAL_ACCESS)
```

*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**.

```
if (ControlCode == IO_START_MON)
{
    UNICODE_STRING devicePath;
    RtlInitUnicodeString(&devicePath, SourceString: (PCWSTR)Irp->AssociatedIrp.SystemBuffer);

    status = STATUS_SUCCESS;
    KPrintInfoN("Start mon from user application!");
    KPrintInfoN("Target: %wZ", &devicePath);
    ByteIo = 0;
    status = extAttachDeviceByPath(deviceObject->DriverObject, targetDevice: devicePath);

    /*Output buffer that wil be passed to application */
    PINT16 statusOutput = (PINT16)Irp->AssociatedIrp.SystemBuffer;
    *statusOutput = 0;

    if (!NT_SUCCESS(status))
    {
        *statusOutput = 2;
        KPrintErr("Unable to attach device to stack!!! QUITTING!\n");
    }
    else
    {
        *statusOutput = 1;
        KPrintInfo("Successfully created and attached device.\n");
    }
    ByteIo = sizeof(*statusOutput);
}
```

*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.

```
else if (ControlCode == IO_STOP_MON)
{
    status = STATUS_SUCCESS;
    KPrintInfn("Stop mon from user application!");
    ByteIo = 0;
    extDetachDevice(deviceObject->DriverObject);
}
```

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 implementację 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.

```
19 else if (ControlCode == IO_GETDATA)
20 {
21     ioctldata_t* Output = (ioctldata_t*)Irp->AssociatedIrp.SystemBuffer;
22     queueNode_t* newData = queueGet();
23
24     if (newData != NULL)
25     {
26         *Output = newData->data;
27     }
28     else
29     {
30         ioctldata_t mockup;
31         WCHAR funcName[FUNC_NAME_MAX] = L"NODATA";
32         RtlZeroMemory(&mockup, sizeof(mockup));
33         RtlCopyMemory(mockup.funcName, funcName, FUNC_NAME_MAX);
34         *Output = mockup;
35     }
36
37     status = STATUS_SUCCESS;
38     ByteIo = sizeof(*Output);
39 }
```

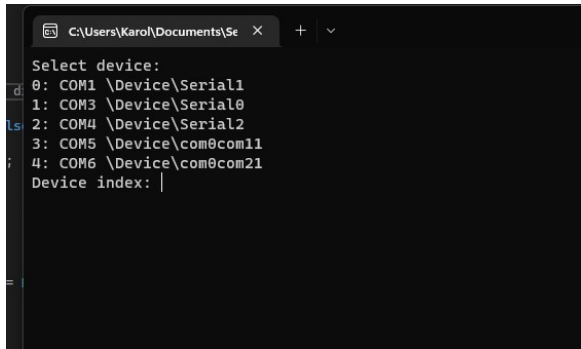
Blok kodu 7.

```
#define FUNC_NAME_MAX 100
#define DATA_SIZE_MAX 512

typedef struct ioctldata
{
    WCHAR funcName[FUNC_NAME_MAX];
    UINT32 dataSize;
    UCHAR data[DATA_SIZE_MAX];
}ioctldata_t;
```

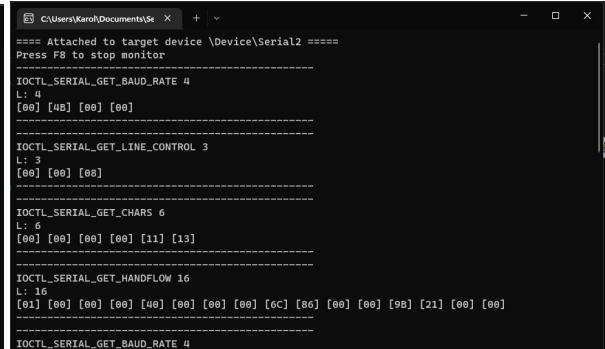
Blok kodu 8. Struktura przekazywanych danych

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



```
C:\Users\Karol\Documents\Se x + v
Select device:
0: COM1 \Device\Serial1
1: COM3 \Device\Serial0
2: COM4 \Device\Serial2
3: COM5 \Device\com0com11
4: COM6 \Device\com0com21
Device index: |
```

Rysunek 1. Wybór urządzenia



```
C:\Users\Karol\Documents\Se x + v
==== Attached to target device \Device\Serial2 ====
Press F8 to stop monitor

-----
IOCTL_SERIAL_GET_BAUD_RATE 4
L: 4
[00] [40] [00] [00]
-----

IOCTL_SERIAL_GET_LINE_CONTROL 3
L: 3
[00] [00] [00]
-----

IOCTL_SERIAL_GET_CHARS 6
L: 6
[00] [00] [00] [00] [11] [13]
-----

IOCTL_SERIAL_GET_HANDFLOW 16
L: 16
[01] [00] [00] [00] [40] [00] [00] [00] [0c] [06] [00] [00] [00] [21] [00] [00]
-----

IOCTL_SERIAL_GET_BAUD_RATE 4
```

Rysunek 2. Przebieg komunikacji

## 4. 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.

```
NTSTATUS status;  
status = IoCreateDevice(driverObject, DeviceExtensionSize: sizeof(DEVICE_EXTENSION),  
    DeviceName: NULL, DeviceType: FILE_DEVICE_SERIAL_PORT,  
    DeviceCharacteristics: 0, Exclusive: FALSE, DeviceObject: &g_extDevice);  
  
if (!NT_SUCCESS(status))  
{  
    KPrintErr("Unable to create target device object!\n");  
    return status;  
}  
else  
{  
    KPrintInfo("Successfully created target device.\n");  
}
```

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.

```
typedef struct  
{  
    PDEVICE_OBJECT LowerDevice;  
    /* This is used to know when we can remove device */  
    IO_REMOVE_LOCK ioRemLock;  
}DEVICE_EXTENSION, * PDEVICE_EXTENSION;
```

Blok kodu 10. `DEVICE_EXTENSION`



```

g_extDevice->Flags |= DO_BUFFERED_IO;
g_extDevice->Flags &= ~DO_DEVICE_INITIALIZING;

/* Zero'ing device extension */
RtlZeroMemory(g_extDevice->DeviceExtension, sizeof(DEVICE_EXTENSION));

/* Attaching device to drivers stack (top of the stack) */
status = IoAttachDevice(SourceDevice: g_extDevice, &targetDevice,
    AttachedDevice: &((PDEVICE_EXTENSION)g_extDevice->DeviceExtension)->LowerDevice);

if (!NT_SUCCESS(status))
{
    IoDeleteDevice(DeviceObject: g_extDevice);
    return status;
}

extDevAttached = TRUE;
/* Initializing remove lock*/
IoInitializeRemoveLock(&((PDEVICE_EXTENSION)g_extDevice->DeviceExtension)->ioRemLock, 0, 0, 0);

return STATUS_SUCCESS;

```

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 buforów jeden po stronie użytkownika drugi po stronie aplikacji. Usprawnia to przesył małych paczek danych ponieważ nie ma wtedy konieczności blokowania całej fizycznej strony pamięci. 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`.

#### a. 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`.

```

NTSTATUS distDeviceControl(PDEVICE_OBJECT deviceObject, PIRP Irp)
{
    if (deviceObject == g_extDevice)
    {
        /* External device */
        return extDeviceControl(deviceObject, Irp);
    }
    else
    {
        /* This device */
        return connDeviceControl(deviceObject, Irp);
    }
}

```

Blok kodu 12. Dyspozytor

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



```

NTSTATUS extDeviceControl(PDEVICE_OBJECT deviceObject, PIRP Irp)
{
    PDEVICE_EXTENSION pDevExt = NULL;
    NTSTATUS status = STATUS_SUCCESS;

    pDevExt = (PDEVICE_EXTENSION)deviceObject->DeviceExtension;

    status = IoAcquireRemoveLock(&pDevExt->ioRemLock, (PVOID)Irp);

    if (!NT_SUCCESS(status)) {
        KPrintErr("Error acquire remove lock! %d", status);
        extSkipPass(deviceObject, Irp);
        return status;
    }

    PIO_STACK_LOCATION pStack = NULL;
    pStack = IoGetCurrentIrpStackLocation(Irp);

```

*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.

```

PIO_STACK_LOCATION pStack = NULL;
pStack = IoGetCurrentIrpStackLocation(Irp);

switch (pStack->Parameters.DeviceIoControl.IoControlCode) {
case IOCTL_SERIAL_GET_BAUD_RATE:
    return extHandleGetIoctl(deviceObject, Irp);
    break;

case IOCTL_SERIAL_GET_HANDFLOW:
    return extHandleGetIoctl(deviceObject, Irp);
    break;

case IOCTL_SERIAL_GET_LINE_CONTROL:
    return extHandleGetIoctl(deviceObject, Irp);
    break;

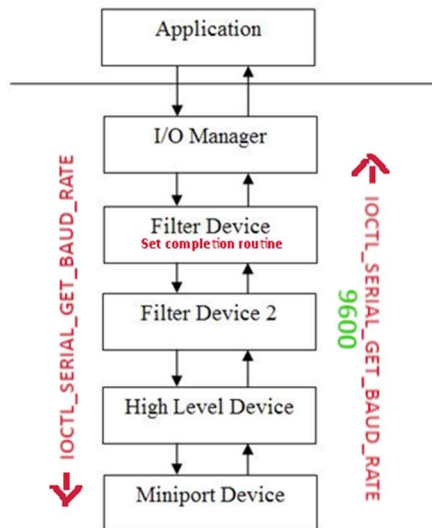
case IOCTL_SERIAL_GET_TIMEOUTS:
    return extHandleGetIoctl(deviceObject, Irp);
    break;

case IOCTL_SERIAL_SET_BAUD_RATE:
    return extHandleSetIoctl(deviceObject, Irp);
    break;

```

*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.

```

NTSTATUS completePendedIrp(PDEVICE_OBJECT deviceObject, PIRP Irp, PVOID Context)
{
    PDEVICE_EXTENSION pDevExt = NULL;
    PIO_STACK_LOCATION pStack = NULL;

    // Does device exist?
    if (deviceObject == NULL)
        return STATUS_UNSUCCESSFUL;

    if (Irp->PendingReturned)
        IoMarkIrpPending(Irp);

    pStack = IoGetCurrentIrpStackLocation(Irp);
    /* Here we can acquire both function name and data from the irp call */
    ULONG ctrCode = pStack->Parameters.DeviceIoControl.IoControlCode;
    WCHAR* funcName = evaluateSerialFuncName(IoControlCode: ctrCode);

    collectData(funcName, dataSize: pStack->Parameters.Write.Length, data: Irp->AssociatedIrp.SystemBuffer);

    pDevExt = (PDEVICE_EXTENSION)g_extDevice->DeviceExtension;
    IoReleaseRemoveLock(&pDevExt->ioRemLock, (PVOID)Irp);

    return STATUS_SUCCESS;
    UNREFERENCED_PARAMETER(Context);
}
  
```

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.

```

NTSTATUS extHandleSetIoctl(PDEVICE_OBJECT deviceObject, PIRP Irp)
{
    NTSTATUS status = STATUS_SUCCESS;
    PIO_STACK_LOCATION pStack = IoGetCurrentIrpStackLocation(Irp);

    PDEVICE_EXTENSION pDevExt = NULL;

    pDevExt = (PDEVICE_EXTENSION)deviceObject->DeviceExtension;
    status = IoAcquireRemoveLock(&pDevExt->ioRemLock, (PVOID)Irp);

    ULONG ctrCode = pStack->Parameters.DeviceIoControl.IoControlCode;
    WCHAR* funcName = evaluateSerialFuncName(IoControlCode: ctrCode);

    collectData(funcName, dataSize: pStack->Parameters.Write.Length, data: Irp->AssociatedIrp.SystemBuffer);

    IoReleaseRemoveLock(&pDevExt->ioRemLock, (PVOID)Irp);
    IoSkipCurrentIrpStackLocation(Irp);
    return IoCallDriver(((PDEVICE_EXTENSION)deviceObject->DeviceExtension)->LowerDevice, Irp);
}

```

Blok kodu 16. *extHandleSetIoctl*

#### b. Obsługa IRP\_MJ\_READ

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

```

NTSTATUS extDeviceRead(PDEVICE_OBJECT deviceObject, PIRP Irp)
{
    PDEVICE_EXTENSION pDevExt = NULL;
    NTSTATUS status = STATUS_SUCCESS;

    pDevExt = (PDEVICE_EXTENSION)deviceObject->DeviceExtension;
    status = IoAcquireRemoveLock(&pDevExt->ioRemLock, (PVOID)Irp);

    KPrintN("IRP_MJ_READ");

    IoCopyCurrentIrpStackLocationToNext(Irp);
    IoSetCompletionRoutine(Irp, CompletionRoutine: (PIO_COMPLETION_ROUTINE) extDeviceReadCompletion, Context: NULL, InvokeOnCompletion: TRUE);
    return IoCallDriver(((PDEVICE_EXTENSION)deviceObject->DeviceExtension)->LowerDevice, Irp);
}

```

Blok kodu 17. *extDeviceRead*

#### c. 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*.

```

NTSTATUS extDeviceWrite(PDEVICE_OBJECT deviceObject, PIRP Irp)
{
    PDEVICE_EXTENSION pDevExt = NULL;
    NTSTATUS status = STATUS_SUCCESS;
    PIO_STACK_LOCATION pStack;

    pDevExt = (PDEVICE_EXTENSION)deviceObject->DeviceExtension;
    status = IoAcquireRemoveLock(&pDevExt->ioRemLock, (PVOID)Irp);

    KPrintN("IRP_MJ_WRITE");

    pStack = IoGetCurrentIrpStackLocation(Irp);

    collectData(funcName: L"IRP_MJ_WRITE", dataSize: pStack->Parameters.Write.Length, data: Irp->AssociatedIrp.SystemBuffer);

    IoReleaseRemoveLock(&pDevExt->ioRemLock, (PVOID)Irp);

    IoSkipCurrentIrpStackLocation(Irp);
    return IoCallDriver(((PDEVICE_EXTENSION)deviceObject->DeviceExtension)->LowerDevice, Irp);
}

```

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`.

```

VOID collectData(WCHAR funcName[FUNC_NAME_MAX], UINT32 dataSize, UCHAR data[DATA_SIZE_MAX])
{
    /* Creating new data pack that will be stored in queue*/
    ioctlData_t dataPack;
    RtlFillMemory(&dataPack, sizeof(ioctlData_t), '\0');

    RtlCopyMemory(dataPack.funcName, funcName, FUNC_NAME_MAX);
    dataPack.dataSize = dataSize;

    if (dataSize > 0 && dataSize <= DATA_SIZE_MAX)
    {
        RtlCopyMemory(dataPack.data, data, dataSize);
    }

    KPrintN("%ws", funcName);
    if (dataPack.dataSize > 0)
        KPrintN("L: %d", dataPack.dataSize);

    for (unsigned int i = 0; i < dataSize; i++)
    {
        KPrint("[%02X] ", dataPack.data[i]);
    }

    if (dataPack.dataSize > 0)
        KPrint("\n");

    queueAdd(inData: dataPack);
}

```

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.

```

typedef struct ioctlData
{
    WCHAR funcName[FUNC_NAME_MAX];
    UINT32 dataSize;
    UCHAR data[DATA_SIZE_MAX];
}ioctlData_t;

```

Blok kodu 20. Format dataPack

## 5. Funkcje pomocnicze

### a. 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 ilości 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.

```
#pragma once
#define KPrint(s, ...) DbgPrintEx(ComponentId: 0, Level: 0, s, __VA_ARGS__);
#define KPrintInfo(s, ...) DbgPrintEx(ComponentId: 0, Level: 0, Format: "SMON:INFO:: " ## s, __VA_ARGS__);
#define KPrintInfoN(s, ...) DbgPrintEx(ComponentId: 0, Level: 0, Format: "SMON:INFO:: " ## s##"\n", __VA_ARGS__);
#define KPrintErr(s, ...) DbgPrintEx(ComponentId: 0, Level: 0, Format: "SMON:ERR:: " ## s, __VA_ARGS__);
#define KPrintErrN(s, ...) DbgPrintEx(ComponentId: 0, Level: 0, Format: "SMON:ERR:: " ## s##"\n", __VA_ARGS__);
#define KPrintN(s, ...) DbgPrintEx(ComponentId: 0, Level: 0, Format: "SMON:: " ## s##"\n", __VA_ARGS__);
```

Blok kodu 21. Zawartość pliku `helpers.h`

### b. Kolejka danych

Wartym opisanie jest również implementacja kolejki przechowującej dane. Kolejka wykorzystuje funkcjonalność spin lock, który zmusza część 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`.

```
BOOLEAN isQueueInitialized = FALSE;

VOID queueInitialize()
{
    pHead = pTail = NULL;
    queueSize = 0;
    KeInitializeSpinLock(&kSLock);
    isQueueInitialized = TRUE;
}
```

Blok kodu 22. Funkcja inicjalizująca kolejkę

```
queueNode_t* queueGet()
{
    queueNode_t* pRet = NULL;
    KIRQL kIrql;

    KeAcquireSpinLock(&kSLock, &kIrql);
    pRet = pHead;
    if (pRet != NULL) {
        pHead = pRet->next;
        queueSize--;
    }
    else {
        pTail = pHead;
    }
    KeReleaseSpinLock(&kSLock, kIrql);

    return pRet;
}
```

Blok kodu 23. Funkcja zdejmowania danych z kolejki

## 6. Podsumowanie

Projekt starałem się wykonać jak najdokładniej, dbając również o zabezpieczenie ewentualnych przypadków brzegowych. Kontynuując dalszą pracę nad sterownikiem zdecydowałem bym się na optymalizację komunikacji, pomiędzy aplikacją kliencką, a sterownikiem – zastosowałem 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>