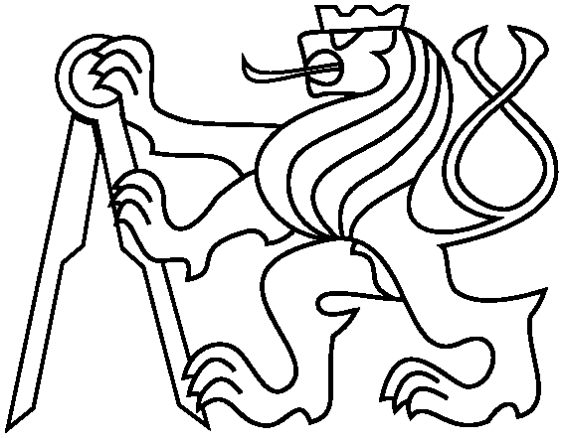
ČESKÉ VYSOKÉ UČENÍ TECHNICKE V PRAZE

Fakulta elektrotechnická

Katedra měření



Diplomová práce

Základní programové vybavení pro tester CAN

Jiří Šplíchal

Vedoucí práce: Doc. Ing. Jiří Novák, PhD.

Praha 2014

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická
Katedra měření

Akademický rok **2012-2013**

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Jiří Šplíchal**

Studijní program: **Kybernetika a robotika**
Obor: **Senzory a přístrojová technika**

Název tématu česky: **Základní programové vybavení pro tester CAN**

Název tématu anglicky: **Basic Software for the CAN Tester**

Pokyny pro vypracování:

Pro definované aplikační rozhraní navrhněte a implementujte ovladač a DLL knihovnu pro PCI kartu testeru sběrnice CAN v operačním systému Windows 7. Soustřeďte se zejména na vhodné rozdělení funkcí mezi ovladač karty a DLL knihovnu a na vhodné mechanismy komunikace mezi ovladačem a knihovnou, případně aplikací. Funkčnost výsledného řešení demonstrujte jednoduchou testovací aplikací.

Seznam odborné literatury:

[1] Oney, W.: Programming the Windows Driver Model, eBook, Microsoft 2003
[2] Haasz a kol.: Číslicové měřicí systémy, ČVUT, Praha 2000
[3] Standard CAN 2.0, Bosch 1991

Vedoucí diplomové práce: doc. Ing. Jiří Novák, Ph.D.

Datum zadání diplomové práce: 23. listopadu 2012

Platnost zadání do[1]: 30. června 2014

L.S.

Prof. Ing. Vladimír Haasz, CSc.
vedoucí katedry

Prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 23. 11. 2012

[1] Platnost zadání je omezena na dobu tří následujících semestrů.

ANOTACE

Tato diplomová práce se zabývá implementací ovladačů v operačním systému Microsoft Windows 7. Kromě podrobného rozboru problematiky týkající se všech aspektů programování ovladačů, především modelu Windows Driver Model, je v rámci práce navrhnut a implementován ovladač jádra pro PCI kartu testeru sběrnice CAN. Součástí práce je také vytvoření knihovny DLL, která zpřístupňuje služby ovladače.

ANOTATION

The master's thesis deals with the implementation of the device drivers in the Microsoft Windows 7. It describes all aspects of programming device drivers and it is primarily focused on the Windows Driver Model. Working design and implementation of kernel-mode device driver for CAN tester has been introduced together with DLL library which serves as user interface to driver.

Čestné prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne …….. …..…….

Podpis autora práce

Poděkování

Chtěl bych v první řadě poděkovat vedoucímu práce panu doc. Ing. Jiřímu Novákovi, Ph.D. za četné připomínky a návrhy, které vedly k úspěšnému dokončení této práce. Dále děkuji rodině a všem blízkým, kteří mě za celou dobu studia neustále podporovali.

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

S pronikáním elektroniky do automobilového průmyslu dochází v současnosti k zajištění optimální funkce automobilu a zvýšení bezpečnosti i komfortu řidičů. Významně se podílí na větší bezpečnosti silničního provozu. Případná chybná funkce může mít neblahé následky pro posádky vozidel. Nezbytností je spolehlivá komunikace elektronických jednotek se systémem vozidla. K zajištění náročných požadavků je třeba důsledné testování a analýza komunikace na sběrnici Controller Area Network (CAN), k čemuž by měla částečně přispět i tato diplomová práce.

Controller Area Network je sériový komunikační protokol, který byl původně vyvinut firmou Bosch pro nasazení v automobilovém průmyslu. S podporou předních výrobců, kteří implementovali podporu protokolu do integrovaných obvodů, dochází k využívání tohoto protokolu v jiných oblastech průmyslu. Výhodou je nízká cena, snadné nasazení a spolehlivost.

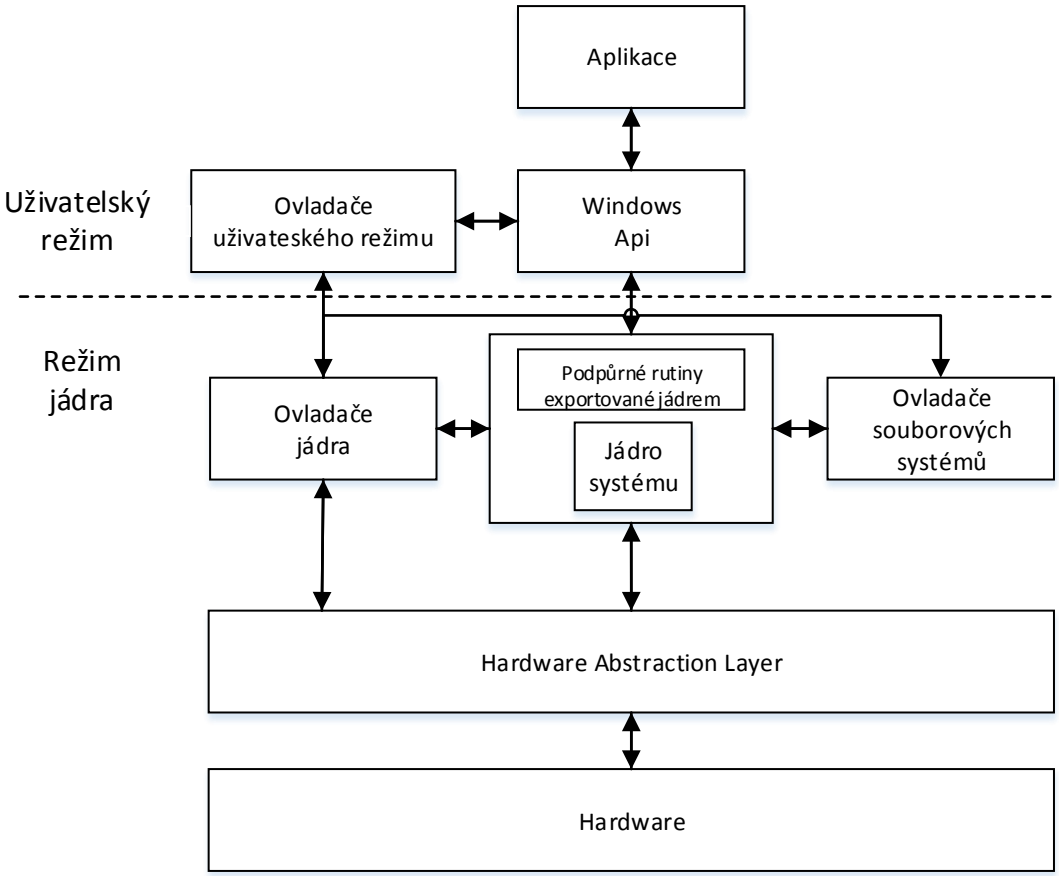
Cílem této diplomové práce je návrh a implementace ovladače pro PCI kartu v operačním systému Microsoft Windows 7. Tato karta je součástí testeru sběrnice CAN pro automobilové jednotky. Navrhované řešení vychází z původní verze PCI karty, která obsahovala pouze jeden řadič sběrnice CAN SJA1000. Nová verze disponuje čtveřicí těchto řadičů a rozdílným způsobem časování odesílání zpráv. Z těchto důvodů byla vytvořena modifikovaná verze ovladače a knihovny DLL podporující tato rozšíření.

V úvodních kapitolách této práce je popsáno začlenění ovladačů v architektuře Windows a dále jsou charakterizovány jejich jednotlivé typy. Následující kapitola detailně rozebírá problematiku implementace ovladačů v modelu Windows Driver Model (WDM). Čtvrtá kapitola představuje nový framework Kernel Mode Driver Framework (KMDF) pro vývoj ovladačů jádra. Další kapitola se zabývá detailněji vnitřní strukturou PCI karty testeru a jsou zde definovány hlavní požadavky na funkce ovladače. Šestá kapitola popisuje implementaci celého řešení. Závěrečná kapitola se věnuje testování vytvořeného ovladače.

2 Ovladače systému Microsoft Windows

2.1 Ovladače v architektuře Windows

Na obrázku 1 je schematicky znázorněné začlenění a komunikace ovladačů v architektuře operačního systému Microsoft Windows. Jak je z obrázku patrné, základní součásti Windows jsou rozděleny na komponenty, jejichž procesy jsou spuštěné buď v režimu jádra, nebo v uživatelském režimu.



Obrázek 1 – Architektura Microsoft Windows [2]

Každý procesor implementuje několik úrovní oprávnění, které přesně definují operace, které kód programu běžící na daném procesoru může využívat. Omezení se například vztahuje na instrukce nebo oprávnění přístupu do paměti. I když procesory kompatibilní s architekturou x86 a x64 definují čtyři úrovně oprávnění, operační systém Windows využívá pouze dvě – režim jádra a uživatelský režim. Tento přístup zajišťuje ochranu důležitých

systémových komponent (běžících v režimu jádra) před modifikací například zákeřnou aplikací a tím zaručuje stabilitu celého systému.

Přestože má každý proces v uživatelském režimu k dispozici privátní virtuální adresní prostor, všechen kód spuštěn v režimu jádra sdílí společný virtuální adresní prostor. Každá stránka virtuální paměti určuje úroveň oprávnění, jakou procesor musí mít pro zápis nebo čtení [2]. Sdílení adresního prostoru přináší velké nároky na stabilitu kódu spuštěného v režimu jádra, tedy i ovladačů, aby nenarušily integritu celého systému a nezpůsobily nestabilitu a pád.

Kernel-mode ovladače mohou využívat podpůrné rutiny, které jsou definované a poskytované různými komponentami jádra operačního systému, jako jsou například I/O, konfigurace, Plug and Play, power management, memory management a další služby operačního systému. Aplikace pak využívají ke komunikaci funkce definované v Microsoft Win32 API. Tyto rutiny pak volají požadované funkce jádra operačního systému.

Systém zpravidla nevytváří speciální proces pro kód ovladače. Systém spouští rutiny ovladače v kontextu vlákna, který je právě naplánovaný na daném procesoru. Kód a data ovladačů jsou namapovány ve virtuálním prostoru každého procesu na stejném místě, proto nezáleží, v jakém kontextu je kód ovladače spuštěn. Následující výčet uvádí několik pravidel pro určení, v jakém kontextu bude kód ovladače spuštěn.

- Přechod z uživatelského režimu do režimu jádra v rámci systémového volání nemění kontext vlákna ani procesu.
- Kód vykonávaný některým pracovním vláknem (jedná se většinou o asynchronní zpracování hardwaru na dřívější požadavek) běží vždy v kontextu *System*.
- Obsluha výjimek a přerušení probíhá v kontextu vlákna, které se nacházelo na procesoru okamžiku vzniklé události.

Pomocí funkcí exportovaných hlavním modulem jádra je možno vykonávat kód ovladače v libovolném kontextu běžícího procesu. Tento přístup se používá ve výjimečných případech.

Ovladače, přestože jsou spouštěné v režimu jádra, pro přístup k hardwaru využívají služby HAL (Hardware Abstraction Layer). HAL zajišťuje nezávislost jádra systému a dalších aplikací na použitém hardwaru, jako například použitém modelu procesoru. HAL

rutiny využívají metody, které jsou naopak závislé na platformě a poskytují rozhraní pro další software [1].

2.2 Rozdělení ovladačů

Operační systém Windows podporuje velké množství rozdílných typů a modelů ovladačů. Nejširší možné rozdělení je na ovladače režimu jádra a ovladače uživatelského režimu.

Ovladače v uživatelském režimu jsou spuštěny v neprivilegovaném modu procesoru spolu s ostatními aplikacemi. Tyto ovladače nemohou přistupovat k systémovým prostředkům jinak, než pomocí funkcí Win32 API, které pak volají systémové služby jádra. Ovladače mají neomezený zásobník a jsou snáze laditelné [5]. Následující výčet uvádí jednotlivé typy ovladačů v uživatelském režimu.

- **Virtual device driver** (VDD) – je user mode komponenta, která umožňuje aplikacím MS-DOS přistupovat k hardwaru na platformě Intel x86. VDD v podstatě simuluje chování hardwaru pro aplikace, které komunikují přímo s hardwarem.
- **Ovladače tiskáren** – tyto ovladače překládají nezávislé grafické požadavky na příkazy specifické danému zařízení. Tyto požadavky jsou pak přeposílány ovladači v režimu jádra jako je například USB printer port driver [2].
- **Ovladače architektury UMDF** – tyto ovladače zařízení jsou spuštěné v uživatelském režimu a s podpůrnou knihovnou UMDF v režimu jádra komunikují pomocí ALPC (pokročilé lokální volání procedur) [2].

Ovladače v režimu jádra jsou spuštěny jako součást systémové exekutivy. Většina ovladačů zařízení běží v tomto režimu. Tyto ovladače mohou vykonávat určité chráněné operace, na rozdíl od ovladačů v uživatelském režimu, ale za cenu obtížnějšího ladění. V případě nevhodného chování mohou narušit celý systém. Operační systém v tomto prostředí také méně kontroluje integritu a platnost dat. Ovladače v režimu jádra lze rozdělit do následujících základních kategorií.

- **File system driver** – implementuje standardní souborové systémy PC na lokálních nebo síťových discích.
- **Ovladače Plug and Play** – tyto ovladače zařízení implementují Plug and Play a power management protokol.

- **Ovladače Non-Plug and Play** – jsou ovladače nebo moduly, které rozšiřují funkcionalitu systému. Obvykle neimplementují Plug and Play ani power management protokol, protože nejsou spjaté s konkrétním hardwarem. Do této kategorie například spadá síťové API a ovladače protokolů [2].

Z důvodu kompatibility zdrojového kódu ovladače napříč všemi verzemi operačního systému Windows Microsoft zavádí několik frameworků, podle kterých je možno implementovat ovladač. Dalším kritériem dělení ovladačů je, podle jakého modelu jsou naprogramovány.

- **WDM** (Windows driver model) – je ovladač, který implementuje Plug and Play a power management protokol. Mezi WDM ovladači rozlišujeme:
    - Class driver – ovladače pro předdefinované skupiny zařízení,
    - Minidriver - poskytují podporu pro ovladače Class driver a implementují specifika výrobce,
    - Monoliticfunction driver – zajišťuje celkovou funkcionalitu pro hardware,
    - Filter driver – filtruje požadavky za účelem modifikace nebo přidání funkcionality již k existujícímu ovladači.

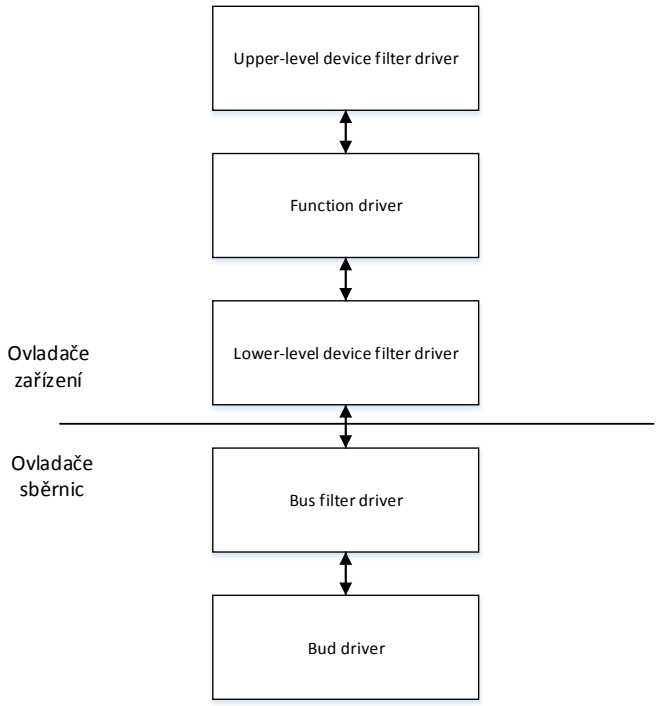
- **WDF** (Windows driver foundation) – WDF je sada nástrojů, které usnadňují vývoj ovladačů zařízení. Tento model představuje abstraktní vrstvu nad modelem WDM a umožňuje tak efektivní implementaci robustních a bezpečných ovladačů.

    - **User Mode Driver Framework** – UMDF jsou navrženy zejména pro podporu zařízení, která využívají standardizované protokoly a přidávají specifickou funkcionalitu. Podporované protokoly jsou: IEEE 1394, USB, Bluetooth a TCP/IP.
    - **Kernel Mode Driver Framework** – poskytuje jednoduché rozhraní pro model WDM, a tím skrývá jeho komplexnost před programátorem, aniž by pozměnil standardní architekturu WDM modelu. Hlavní předností je využití defaultních obslužných rutin pro Plug and Play a power management, čímž vývojáři umožňují zaměřit se více na samotnou funkcionalitu zařízení.

3 Windows Driver Model

3.1 Vrstvová architektura WDM

V modelu WDM (Windows driver model) není konkrétní zařízení obsluhováno pouze jedním ovladačem. Funkcionalita je rozdělena do více ovladačů, které jsou řazeny do několika vrstev. Pro každé zařízení existuje struktura ovladačů, která je schematicky znázorněna na obrázku 2. Pro každé zařízení ve WDM existují nejméně dva ovladače, a to *function driver* a *bus driver*. Následující seznam popisuje jednotlivé bloky struktury.



Obrázek 2 – Vrstvy ovladače WDM

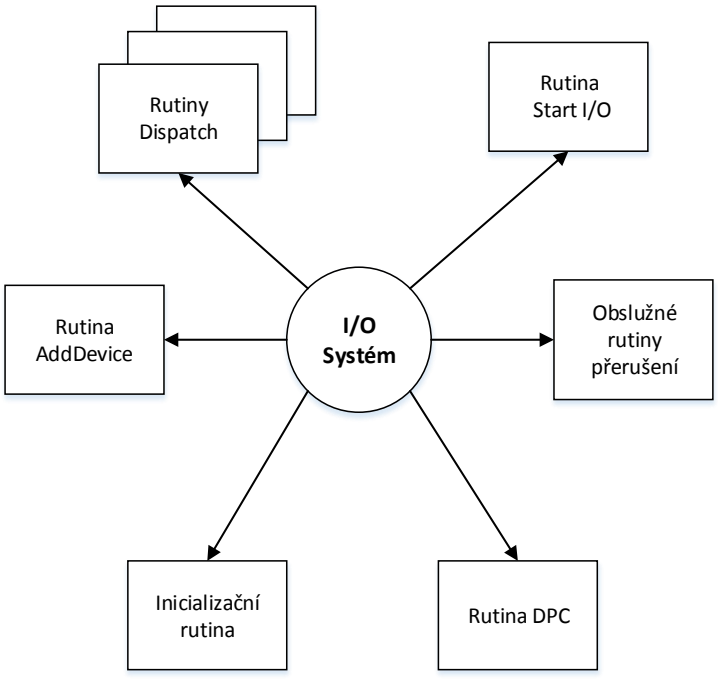
- **Function driver** - představuje základní funkcionalitu ovladače, představuje rozhraní mezi hardwarem a operačním systémem. Jeho hlavním úkolem je inicializace I/O operací, obsluha přerušení a poskytování způsobů řízení a komunikace pro koncového uživatele.
- **Bus driver** – zprostředkovává komunikaci mezi hardwarem a příslušnou sběrnicí. Tento ovladač je zodpovědný za enumeraci připojených zařízení ke sběrnici a poskytování této informace správci Plug and Play. Také spravuje napájení příslušné sběrnice.

- **Filter drivers** - jak název napovídá, filtrují (modifikují) požadavky postupující strukturou ovladačů. Mohou být umístěny nad nebo pod function driverem. Většinou pozměňují a doplňují již existující funkcionalitu standardního ovladače specifickou pro konkrétního výrobce zařízení.

3.2 Struktura ovladačů WDM

Ovladač zařízení si lze představit jako sadu obslužných rutin, které jsou volány správcem vstupně výstupních operací v závislosti na typu I/O požadavku, který zpracovává. Na obrázku 3 jsou zobrazeny základní obslužné rutiny ovladače.

- **Inicializační rutina** (DriverEntry) – tato rutina je vstupní bod ovladače a je volána správcem I/O operací při zavádění ovladače do systému. Jejím hlavním úkolem je globální inicializace ovladače, což v podstatě znamená vyplnit systémovou strukturu DRIVER_OBJECT, a tím registrovat zbytek obslužných rutin ovladače.
- **Rutina AddDevice** – každý ovladač podporující Plug and Play musí implementovat rutinu AddDevice. Rutina je zodpovědná za vytvoření objektu DeviceObject, který reprezentuje fyzické, logické nebo virtuální zařízení, pro které ovladač zpracovává I/O požadavky. Dále je zodpovědná za připojení objektu zařízení do vrstvové struktury, která je popsaná v kapitole 3.1. Tato rutina je volána správcem Plug and Play pro každé zařízení, které je ovladačem řízeno.



Obrázek 3 - Struktura WDM [2]

- **Rutiny Dispatch** – tyto rutiny slouží k obsluze jednotlivých typů IRP požadavků.
- **Rutina Start I/O** - ovladač může využít tuto rutinu k datovým přenosům „z“ nebo „do“ zařízení. Tato rutina je definovaná jen pro ovladače, které přenechají frontování požadavků správci I/O operací. Správce serializuje požadavky a zajišťuje, aby ovladač zpracovával v danou chvíli pouze jeden požadavek. Ovladače mohou zpracovávat více požadavků simultánně, ale serializace je typicky využívaná, protože zařízení nejsou schopna více požadavků obsloužit.
- **Obslužná rutina přerušení** (ISR) – pokud zařízení generuje přerušení, systém předá řízení právě této rutině. Kód této rutiny je spuštěn na zvýšené úrovni IRQL (viz kapitola 3.4), což klade zvýšené požadavky na rychlost, aby nebyla blokována přerušení s nižším IRQL. Z výše uvedených důvodů je v obsluze přerušení typicky naplánována rutina DPC (deferred procedure call), která běží na nižší úrovni IRQL, a tím umožní další přijetí přerušení.
- **Rutina DPC** (Deferred Procedure Call) – jak bylo naznačeno v předchozím odstavci, DPC rutina má za úkol dokončit I/O operaci, která byla zahájena v ISR rutině.

V průběhu instalace ovladače a enumerace nového zařízení jsou v systému vytvořeny objekty a struktury, které jsou popsány níže.

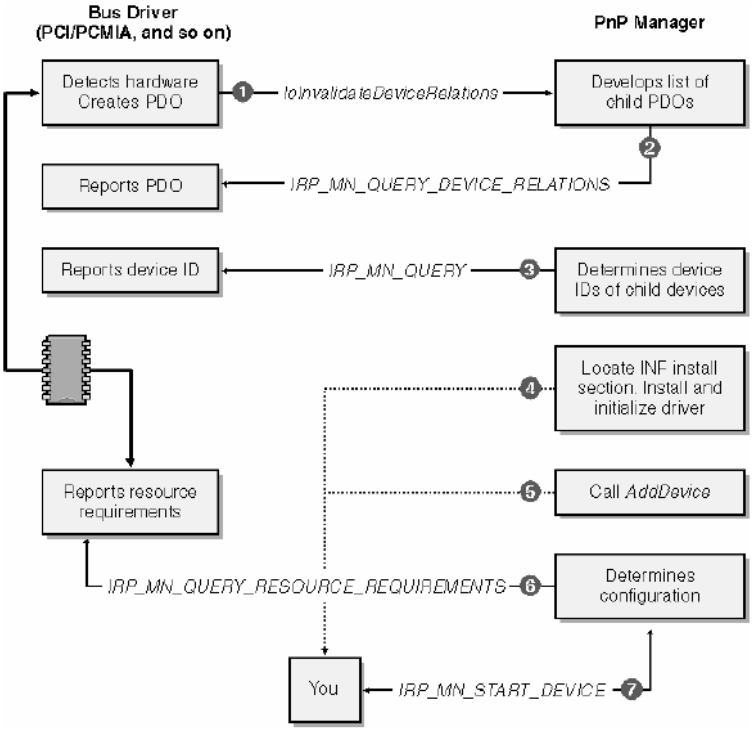
- **Driver Object** - tento objekt je vytvářen správcem vstupně výstupních operací pro každý ovladač instalovaný v počítači. Obsahuje adresy rutin obsluhující IRP požadavky a také ukazatel na strukturu *DeviceObject*. Adresy jsou vyplněny ovladačem v rutině *DriverEntry*.
- **Device Object** - v operačním systému je každé zařízení, ať už fyzické nebo virtuální, reprezentováno jedním nebo vice objekty *DeviceObejct*. Jak vyplývá ze struktury WDM ovladačů, popsané v kapitole 3.1, pro každý ovladač je tento objekt vytvořen. Správce vstupně výstupních operací pak zasílá IRP požadavky nejprve ovladači na vrcholu, ten pak požadavek dokončí nebo jej zašle ovladači nacházejícím se pod ním.
- **Device Extension** - jednou z nejdůležitějších struktur ovladače je *DeviceExtension*. Struktura je definovaná programátorem a jsou v ní uložena nejen data související s ovladačem, ale i objekty jádra, které ovladač využívá.

Těmito objekty mohou být například *DeviceObject, DpcObject, SpinLock, InterruptObject.*

3.3 Zavedení ovladače do systému

V závislosti na implementaci Plug and Play protokolu hardwarem, systém zavádí ovladač dvěma rozdílnými způsoby:

- Zařízení Plug and Play mají elektronickou identifikaci, kterou je systém schopný načíst. Ovladač sběrnice identifikuje nově přidaný hardware. Dle identifikace, registru a INF souborů systém automaticky zavede správný ovladač.
- Legacy ovladače nemají elektronickou identifikaci, a proto je nemůže systém automaticky zavést. Zde musí uživatel systému sdělit, o jaký hardware se jedná, pomocí „Přidat nový hardware“ v ovládacích panelech. Pak je systém schopen pomocí záznamu v registrech a souborů INF zavést správný ovladač stejně jako u Plug and Play zařízení.



Obrázek 4 - Zavedení ovladačů do systému [4]

Plug and Play zařízení

Ovladač sběrnice je schopen zjistit přítomná zařízení skenováním sběrnice při startu systému. Ovladače podporující hot-plugging pro nalezení nového zařízení monitorují hardwarové signály, které způsobí nové skenování sběrnice a vytvoření kolekce objektů PDO (Physical Device Object). Na Obrázku 4 je schematicky znázorněna interakce správce Plug and Play a ovladače sběrnice při zavedení nového Plug and Play zařízení do systému.

Pokud ovladač sběrnice indikuje vložení/odstranění zařízení na sběrnici, upozorní správce Plug and Play o změně počtu. Správce Plug and Play si nejprve vyžádá aktualizovaný seznam PDO objektů (2) a dále identifikátor, který jednoznačně specifikuje vložené zařízení (3). Na základě tohoto identifikátoru správce Plug and Play vyhledá v registrech systému příslušný hardwarový klíč.

Pokud je zařízení vloženo poprvé, systém se nejprve snaží nalézt INF soubor, který odpovídá danému identifikátoru. Na základě INF souboru systém vytvoří hardwarový klíč v systému a nainstaluje potřebný software.

Nakonec správce Plug and Play informuje ovladač o existenci nové instance zařízení (6) a požádá o specifikaci prostředků, jako jsou přerušení, adresy I/O portů, I/O paměti DMA kanálů (7).

3.4 Interrupt Reguest Level (IRQL)

Interrupt request level definuje hardwarovou prioritu, kterou ovladače specifikují při registraci obslužné rutiny přerušení a se kterou musí operační systém vykonávat její kód. Tato priorita je spjata s konkrétním procesorem, nikoliv s aktuálním procesem či vláknem. Zde platí pravidlo, že aktuální vlákno může být přerušeno pouze kódem s vyšší hardwarovou prioritou. Jádro systému Windows pracuje s několika předdefinovanými hardwarovými prioritami. Třiceti dvoubitová verze Windows definuje těchto hodnot třicet dva, šedesáti čtyřbitová verze pouze šestnáct. Hierarchie je patrná z tabulky 1.

Tabulka 1 - Hierarchie hardwarových priorit na platformě x86 a x64

31	HIGH_LEVEL
30	POWER_FAIL_LEVEL
29	IPI_LEVEL
28	CLOCK_LEVEL
27	SYNCH_LEVEL/PROFILE_LEVEL
26	Zařízení n
…	
3	Zařízení 1
2	DISPATCH_LEVEL
1	APC_LEVEL
0	PASSIVE_LEVEL

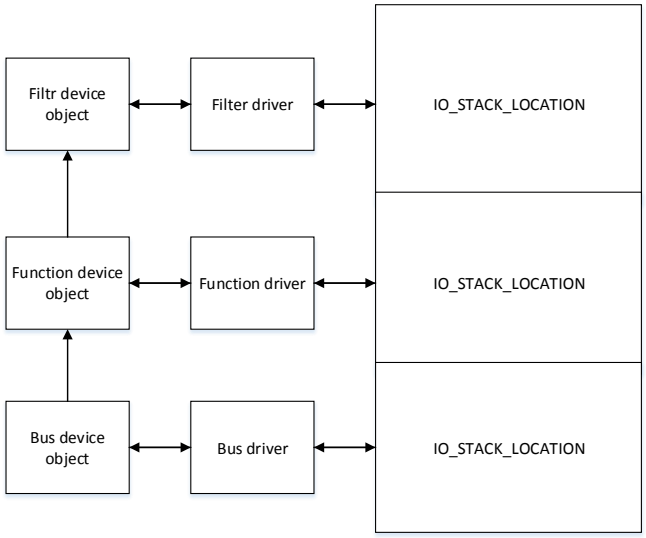
15	HIGH_LEVEL /PROFILE_LEVEL
14	IPI_LEVEL
13	CLOCK_LEVEL
12	SYNCH_LEVEL
11	Zařízení n
…	
3	Zařízení 1
2	DISPATCH_LEVEL
1	APC_LEVEL
0	PASSIVE_LEVEL

- HIGH_LEVEL – nejvyšší priorita je používaná pouze v rutinách souvisejících s modrou obrazovkou smrti.
- IPI_LEVEL – je využívána procesory při vzájemné komunikaci, například v případě synchronizace jejich vyrovnávacích pamětí.
- Zařízení 1 až Zařízení n – priority přerušení generovaných zařízeními. Konkrétní hodnoty přiděluje HAL.
- DISPATCH_LEVEL – na této úrovni již není možno alokovat stránkovanou paměť a je též poslední prioritou, kde lze alokovat paměť nestránkovanou. S touto prioritou běží například plánovač nebo rutiny DPC.
- APC_LEVEL – některé druhy asynchronních procedur probíhají na této úrovni. Zde je možno využívat i stránkovanou paměť a možnosti synchronizace.
- PASSIVE_LEVEL – veškeré uživatelské aplikace a většina kódu ovladačů jsou vykonávány s touto prioritou. Vlákna mohou bez omezení pracovat se stránkovaným fondem paměti a využívat synchronizační primitiva.

3.5 Vstupně výstupní model

3.5.1 IRP (I/O RequestPacket)

IRP je základní struktura správce I/O operací, pomocí které komunikuje s ovladači a umožňuje ovladačům komunikovat mezi sebou. Tato struktura obsahuje například ukazatele na buffery uživatelského režimu spojené s IRP, na systémové buffery spojené se čtením a zápisem dat do zařízení a dále na informaci o stavu v jakém se požadavek nachází.



Obrázek 5- Vztah mezi ovladačem a I/O stack location [4]

3.5.2 I/O Stack Locations

Pro každý IRP požadavek správce I/O vytvoří pole struktur IO_STACK_LOCATION pro každý ovladač v zásobníku ovladačů, viz obrázek 5. V této struktuře jsou obsaženy například informace jako kódy minor a major funkcí a ukazatel na rutinu pro dokončení daného IRP.

Synchronní a asynchronní IRP

Synchronní IRP náleží vláknu, v jehož kontextu bylo vytvořeno. To má několik specifik, které jsou popsané v následujícím seznamu [5].

- Vlákno je blokováno, dokud IRP není dokončeno.
- Pokud je vlákno ukončeno, správce I/O operací zruší všechna synchronní IRP, která vlákno vytvořilo.

- IRP nesmí být vytvořeno v systémovém vláknu.
- I/O manager automaticky uvolní buffery spojené s IRP.
- IRP musí být vytvořeno na úrovni PASSIVE_LEVEL. Správce I/O operací by nebyl schopen automaticky dokončit operaci, k čemuž využívá APC rutinu a zvednout IRQL na APC_LEVEL.

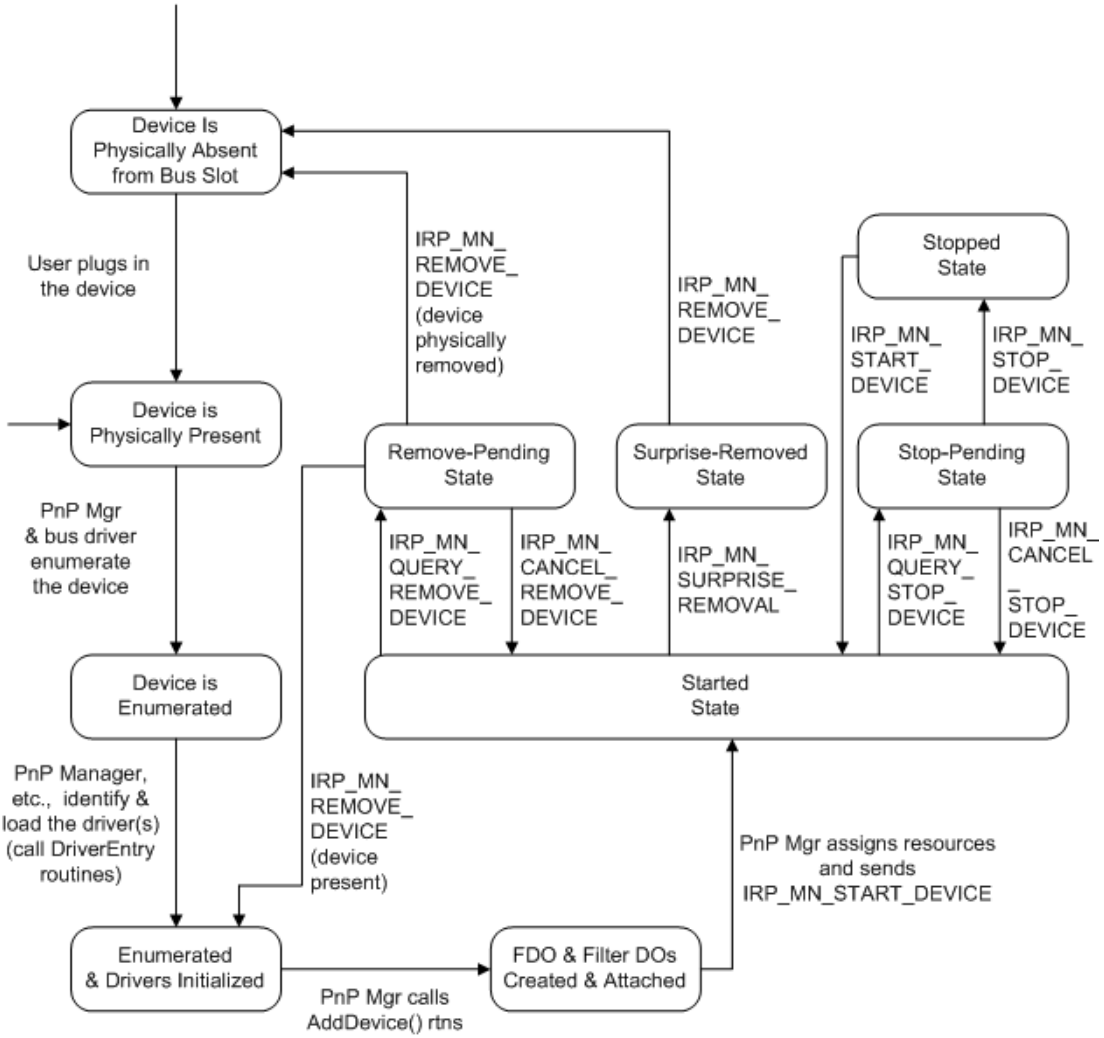
Asynchronní IRP oproti předcházejícímu nenáleží žádnému vláknu. Správce I/O operací neplánuje APC rutinu a po skončení IRP neuvolňuje žádnou alokovanou paměť. Důsledky jsou následující:

- Správce vstupních a výstupních operací nezruší žádné asynchronní IRP s ukončením vlákna.
- Vlákno nečeká na dokončení IRP.
- Lze vytvořit IRP i v systémovém vláknu.
- Ovladač musí implementovat rutiny pro dokončení a uvolnit buffery vázané k IRP.
- IRP může být vytvořeno na úrovni IRQL menší nebo rovné DISPATCH_LEVEL.

3.6 Plug and Play (Plug and Play)

Plug and Play je kombinace hardwarové a softwarové podpory, která umožňuje systému rozpoznat a přizpůsobit se změně hardwarové konfigurace, a to bez zásahu uživatele. Tato podpora vyžaduje úzkou spolupráci fyzického zařízení, operačního systému a ovladače zařízení [5]. Operační systém spolu s ovladačem zajišťují následující funkce:

- Automatické rozpoznání instalovaného hardwaru – systém rozpozná fyzické zařízení při prvotní instalaci systému, rozpozná změny mezi jednotlivými starty systému a reaguje na změny za chodu, jako jsou vyjmutí nebo přidání zařízení.
- Přidělení systémových prostředků – Správce Plug and Play si vyžádá hardwarové požadavky jednotlivých zařízení (jako jsou I/O porty, DMA kanály, přerušení, paměť) a následně provede jejich přiřazení. Také provádí realokaci zdrojů například v případě, kdy je přidáno nové zařízení a požadované prostředky jsou již použity.
- Načtení správného ovladače.
- Programové rozhraní pro interakci se správcem Plug and Play.
- Mechanismy zajišťující informovanost ovladačů a aplikací o změnách v hardwaru.



Obrázek 6 - Stavy zařízení Plug and Play [5]

3.6.1 Rutina Plug and Play Dispatch

Zodpovědnost za vykonávání požadavků s majoritní funkcí IRP_MJ_PLUG AND PLAY přebírá Plug and Play dispatch rutina. Tato rutina volá funkce v závislosti na minoritním kódu IRP požadavku. V tabulce 2 je výčet požadovaných kódů pro ovladač zařízení WDM [5]. Lze si povšimnout souvislosti se stavem zařízení a jeho změnami (přesuny) z Obrázku 6.

Tabulka 2 – Požadované IRP modelu WDM

IRP_MN_START_DEVICE
IRP_MN_QUERY_STOP_DEVICE
IRP_MN_STOP_DEVICE
IRP_MN_CANCEL_STOP_DEVICE
IRP_MN_QUERY_REMOVE_DEVICE
IRP_MN_REMOVE_DEVICE
IRP_MN_CANCEL_REMOVE_DEVICE
IRP_MN_SURPRISE_REMOVAL

Start zařízení

Plug and Play podpora zajišťuje automatické rozpoznání požadavků zařízení. V okamžiku, kdy je přiřazení zdrojů známé, správce Plug and Play oznámí tuto skutečnost ovladači zasláním požadavku s minoritním kódem IRP_MN_START_DEVICE.

Obslužná rutina musí v první řadě předat IRP dál ve vrstvové struktuře ovladačů a čekat na dokončení všemi spodními ovladači. Pak může spustit vlastní funkci *StartDevice*.

Hlavním úkolem ovladače je po přijetí požadavku uložit zdroje, které správce Plug and Play zařízení přiřadil. Přiřazené zdroje jsou uloženy v parametrech příslušné I/O_STACK_LOCATION ve dvou podobách:

- Alocated Resources – tyto zdroje jsou popsány adresami, které jsou relativní vůči sběrnici. Zpravidla ovladač poskytuje tyto adresy zařízení, které konfiguruje, respektive programuje.
- Alocated Translated Resources – tyto zdroje jsou popsány fyzickými adresami, které ovladač používá k přístupu k danému zdroji [5].

Zastavení zařízení

Správce Plug and Play instruuje zastavení zařízení v případě, kdy je potřeba znovu přidělit zdroje. Toto je nezbytné zpravidla, kdy je do systému přidáno nové zařízení s požadavky na zdroje, které jsou již využity.

Pokud ovladač přijme požadavek IRP_MN_STOP_DEVICE, zastaví zařízení, což znamená přerušení veškerých aktivit a zakázání generování dalších přerušení hardwarem. Dále musí ovladač uvolnit všechny zdroje, které využívá. Nakonec by měl být požadavek přeposlán v zásobníku ovladačů na nižší úroveň.

Před zasláním požadavku IRP_MN_STOP_DEVICE správce Plug and Play zasílá požadavek IRP_MN_QUERY_STOP_DEVICE, čímž žádá povolení k zastavení zařízení [5]. Tento požadavek nesmí být schválen v následujících situacích:

- Pokud byl ovladač upozorněn, že je zařízení použito pro stránkování, hibernaci, nebo crashdump soubor.
- Zdroje, které zařízení využívá, nemohou být uvolněny.
- Zařízení nemá implementované mechanismy frontování IRP požadavků a není přípustné zahodit I/O operaci.

Pokud je požadavek schválen, ovladač by měl implementovat mechanismy pro frontování IRP po dobu zastavení zařízení.

Pokud jakýkoliv ovladač ve vrstvové struktuře neschválí IRP_MN_QUERY_STOP_DEVICE, správce Plug and Play zasílá požadavek IRP_MN_CANCEL_STOP_DEVICE, čímž se zařízení vrátí do stavu WORKING.

Odstranění zařízení

Podobně jako u zastavení zařízení správce Plug and Play před odebráním zařízení informuje ovladač zasláním IRP_MN_QUERY_REMOVE_DEVICE pro získání povolení k odebrání, aby nedošlo k narušení funkce zařízení. Tento požadavek je zasílán i v případě aktualizace ovladače nebo při zakázání zařízení ve správci zařízení.

Správce Plug and Play může vrátit zařízení zpět do stavu WORKING pomoci IRP_MN_CANCEL_REMOVE_DEVICE.

Po úspěšném dotazu správce Plug and Play zasílá IRP_MN_REMOVE_DEVICE. Ovladač musí tento požadavek schválit, musí uvolnit alokovanou paměť, odpojit se ze zásobníku a smazat objekt *DeviceObject*, který vytvořil ve funkci *AddDevice*.

Odstranění SurpriseRemoval

Správce Plug and Play zasílá požadavek IRP_MN_SURPRISE_REMOVAL v případě, kdy uživatel vyjme zařízení bez použití nástrojů pro odebrání zařízení. Ovladač po přijetí tohoto požadavku má za úkol zastavit zpracovávaní všech zbývajících IRP, uvolnit použité zdroje, které zařízení používá, a zajistit, aby žádné komponenty nepřistupovaly k zařízení. Ovladač musí reagovat na tento požadavek nastavením stavu na STATUS_SUCCESS.

Po uzavření všech handlů k zařízení správce Plug and Play zašle IRP_MN_REMOVE_DEVICE.

3.6.2 Přiřazení systémových prostředků

Hlavním úkolem ovladače po přijetí IRP_MN_STRAT_DEVICE je uložit přiřazené systémové prostředky pro další komunikaci se zařízením. Seznam prostředků je obsažen ve struktuře CM_PARTIAL_RESOURCE_LIST, která obsahuje počet prostředků a pole struktur CM_PARTIAL_RESOURCE_DESCRIPTOR popisující jednotlivé prostředky. V tabulce 3 je uveden seznam nejdůležitějších typů. Ovladač ve smyčce prochází jednotlivé deskriptory a ukládá si jejich reference do *DeviceExtension*.

Tabulka 3 - Typy prostředků ovladače

Typ prostředku
Port
Memory
Interrupt
DMA

3.6.3 Obsluha přerušení

Každý ovladač zařízení, které komunikuje pomocí přerušení, jej musí nejprve nakonfigurovat, tedy každému přerušení registrovat obslužnou rutinu (ISR – Interrupt Service Routine).

Podle způsobu signalizace přerušení rozlišuje dva typy:

- **Line-based** - zařízení určené pro sběrnice PCI předcházející verzi 2.2 generují pouze přerušení pomocí elektrických signálů na linkách přerušení. Tedy obsahují piny určené výhradně k signalizaci. Windows předcházející Windows Vista podporují pouze tento typ přerušení.
- **Message-signaled** – tento způsob signalizace přerušení byl představen se specifikací PCI 2.2. Zařízení generují přerušení zapsáním hodnoty na určitou adresu, tedy neobsahují piny určené pro signalizaci. Tento způsob generování přerušení je dostupný od Windows Vista.

  Každému zařízení je přiřazena adresa. Zpráva se skládá z šestnáctibitové hodnoty. Zařízení může měnit pouze spodní čtyři bity, tedy může generovat šestnáct rozdílných zpráv. Pro specifikaci PCI 3.0 se zpráva skládá z adresy a neměnitelné třicetidvoubitové hodnoty. PCI 3.0 tedy podporuje až 2 048 rozdílných zpráv.

Registrace obslužné rutiny se provádí pomocí funkce *IoConnctInterruptEx*. Je možné specifikovat několik odlišných metod registrace obslužné rutiny.

- CONNECT_LINE_BASED – se používá pro registraci přerušení line-based. Systém automaticky rozpozná všechna přerušení a přiřadí je k danému zařízení.
- CONNECT_MESSAGE_BASED – systém automaticky rozpozná a přiřadí všechny message-signaled přerušení.
- CONNECT_FULLY_SPECIFIED – pomocí tohoto způsobu systém konfiguruje přerušení postupně. Ovladač musí manuálně specifikovat přerušení pomocí informací obdržených od správce Plug and Play. Tento způsob musí implementovat všechny ovladače pro Windows předcházející Windows Vista.

3.6.4 DPC (Deferred procedure calls)

Protože obslužná rutina přerušení je spuštěna na vyšší úrovni IRQL než DISPATCH_LEVEL, všechen kód a data musí být v nestránkované paměti. Měla by proběhnout co nejrychleji, aby zbytečně neblokovala systém. Na této úrovni je také velmi omezena sada funkcí exportovaných jádrem, kterých může obslužná rutina přerušení využít.

Z výše zmíněných důvodů systém nabízí odložené volání procedur. Obslužná rutina přerušení pak může zpracování dat odložit do doby, kdy IRQL procesoru klesne na nižší úroveň, a to DISPATCH_LEVEL.

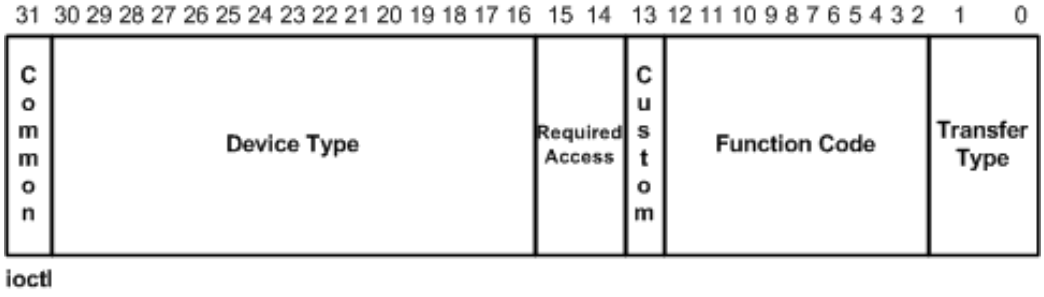
3.7 Komunikace ovladače a aplikace

Ovladače v režimu jádra jsou skryté před aplikacemi spuštěnými v uživatelském režimu. Zařízení jsou dostupná aplikacím pouze prostřednictvím objektů *FileObject,* které jsou kontrolovány správcem I/O operací. Aplikace pak přistupují k ovladači pomocí funkcí *read/write* nebo prostřednictvím požadavků *I/O controlrequest* specifikovaným IOCTL kódem.

3.7.1 Kódy I/O Control codes

I/O Control codes (IOCTL) se používají pro komunikaci mezi aplikací spuštěnou v uživatelském režimu a ovladačem. Pro většinu zařízení aplikace mohou pro zaslání kódu použít standardní funkci z Win32 API *DeviceIoControl*. Správce I/O zařízení reaguje na volání této funkce vytvořením IRP požadavku IRP_MJ_DEVICE_CONTROL, který obslouží rutina definovaná v ovladači.

Kódy mohou být veřejné nebo privátní. Veřejné jsou typicky definované systémem a zdokumentované buď v WDK nebo SDK. Oproti tomu privátní kódy jsou určené pro vývojáře a nejsou veřejně zdokumentované. Kód představuje třiceti dvoubitovou konstantu, jejíž struktura je patrná z obrázku 7.



Obrázek 7 - Struktura IOCTL kódu

- Device Type – tato šestnáctibitová hodnota určuje typ zařízení, pro které je kód určen. Tato hodnota musí být shodná s typem zařízení, které je specifikováno ve funkci

IoCreateDevice. Hodnoty pod 0x8000 jsou rezervované systémem. Vyšší hodnoty mohou využívat vývojáři ovladačů.

- Required Access – tyto dva bity určují, jaká oprávnění musí subjekt získat, aby mohl zprávu s daným kódem úspěšně poslat.
- Function Code – rozlišovací kód zprávy slouží více druhům zpráv pro jedno zařízení. Kódy s hodnotou nižší než 0x800 jsou rezervované systémem. Pro vývojáře jsou určené kódy s hodnotou vyšší, která také nastavuje bit *Custom*.
- Transfer Type – poslední dvojice bitů určuje způsob, jak správce vstupních a výstupních operací pracuje s I/O bufferem. Kapitola 3.7.2 detailněji popisuje jednotlivé způsoby.

3.7.2 Způsoby přenosu dat

Jedním z hlavních úkolů zásobníku ovladačů je přenášet data mezi aplikacemi spuštěných v uživatelském režimu a zařízeními, jejichž ovladače běží v režimu jádra. Operační systém Windows nabízí následující způsoby přenosů [1][5]:

- Bufferovaná metoda,
- Metoda přímého vstupu/výstupu,
- Metoda nulové režie.

Bufferovaná metoda

Tato metoda se využívá nejčastěji k přenosu malých objemů dat ze zařízení, jako jsou například klávesnice, myši, sériový a paralelní port.

Aplikace, která zahájí vstupně/výstupní operaci předá své buffery správci vstupně/výstupních operací pomocí virtuálních adres uživatelského režimu. Během procesu vytvoření IRP požadavku správce vstupně/výstupních operací alokuje systémový buffer z nestránkovaného fondu velikosti uživatelského bufferu a zkopíruje do něj celý obsah bufferů uživatelských. Ovladač pak může bezpečně přistupovat k těmto bufferům, které jsou alokované v nestránkované paměti bez toho, aby stránky byly nejprve uzamčené. V okamžiku, kdy vlákno aplikace je znovu aktivní, správce překopíruje obsah systémového bufferu do uživatelského a uvolní systémovou paměť.

Z důvodu alokace paměti v nestránkovaném fondu není tento způsob výměny dat vhodný k přenosům větších objemů dat, protože nestránkovaná paměť může být fragmentovaná a správce vstupních/výstupních zařízení by nebyl schopen alokovat potřebnou část spojité paměti v kuse.

Metoda přímého vstupu

Metoda je využívaná u ovladačů zařízení, které přenášejí velké objemy dat, typicky flash disky. Správce vstupně/výstupních operací uzamkne stránky paměti obsahující buffery uživatelského režimu a vytvoří pomocnou strukturu MDL (Memory descriptor list) k popisu uzamčených stránek. Správce v IRP požadavku zasílá ukazatel na tuto strukturu. Dokud ovladač nedokončí IRP požadavek stránky zůstávají uzamčeny.

Metoda nulové režie

Správce vstupně/výstupních operací přímo využívá virtuální adresy uživatelského režimu. Z tohoto důvodu ovladač, který využívá tuto metodu přístupu, musí být spouštěn stále ve stejném kontextu vlákna, které tento požadavek vytvořilo. Výhodou tohoto přístupu je nulová režie, a to jak paměťová tak časová. Komunikace s ovladačem tudíž probíhá rychleji než u předcházejících metod. Ovladače souborových systémů využívají právě metodu nulové režie.

3.8 Power Management

Se vzrůstajícími nároky na snížení spotřeby energie, ať už z důvodu prodloužení životnosti baterií přenosných zařízení nebo minimalizace času, který systém potřebuje ke spuštění, Microsoft představuje koncept správce napájení. Z těchto důvodů by měl každý ovladač modelu WDM implementovat tento koncept, čímž umožní snížení spotřeby energie v případě, kdy zařízení není používáno. Správce napájení pak informuje ovladač o změnách stavů spotřeby a tím umožní zařízení přejít do režimu snížené spotřeby a naopak.

Podle specifikace ACPI (Advanced Configuration and Power Interface) WDM definuje diskrétní stavy spotřeby energie zvlášť pro systém jako celek a pro jednotlivá zařízení. Tyto stavy jsou označeny S0 – S5 (pro systém) a D0 – D3 (pro zařízení). Číslo stavu

je inverzně vztažené ke spotřebě energie a k funkcionalitě zařízení. Tedy stavy D0 a S0 značí nejvyšší spotřebu energie a plnou funkcionalitu.

3.8.1 Systémové stavy napájení

Systémové stavy jsou charakterizovány následujícími body:

- Spotřeba elektrické energie,
- Softwarové obnovení,
- Hardwarová latence,
- Systémový kontext.

S0 – stav working

- Spotřeba elektrické energie – maximální
- Softwarové obnovení – není aplikovatelné
- Hardwarová latence – žádná
- Systémový kontext – veškerý kontext je zachován

S1 – režim spánku 1 – vypnuté hodiny procesoru a hodiny sběrnic

- Spotřeba elektrické energie – menší než S0, větší než S2
- Softwarové obnovení – ovládáni obnoveno po přechodu do S0
- Hardwarová latence – typicky dvě sekundy
- Systémový kontext – veškerý kontext zachován

S2 – režim spánku 2 – S1 + vypnutí napájení procesoru (ztracen kontext procesoru a obsah paměti cache), některé sběrnice mohou být odpojeny od napájení

- Spotřeba elektrické energie – menší než S1, větší než S3
- Softwarové obnovení – obnoveno z resetovacího vektoru procesoru
- Hardwarová latence – dvě sekundy nebo vyšší
- Systémový kontext – ztracen kontext procesoru a obsah paměti cache.

S3 – režim spánku 3 – S2 + některé obvody základní desky mohou být odpojeny od napájení

- Spotřeba elektrické energie – menší než S2, větší než S4
- Softwarové obnovení – ovládání obnoveno z resetovacího vektoru procesoru

- Hardwarová latence – dvě sekundy nebo vyšší
- Systémový kontext – ztracen kontext procesoru a chipsetu základní desky a obsah paměti cache

S4 – hibernace

- Spotřeba elektrické energie – žádná
- Softwarové obnovení – obnovení systému z uloženého souboru
- Hardwarová latence – nedefinovaná (závislá na startu PC)
- Systémový kontext – žádný kontext není zachován hardwarem, image paměti je uložena na pevném disku.

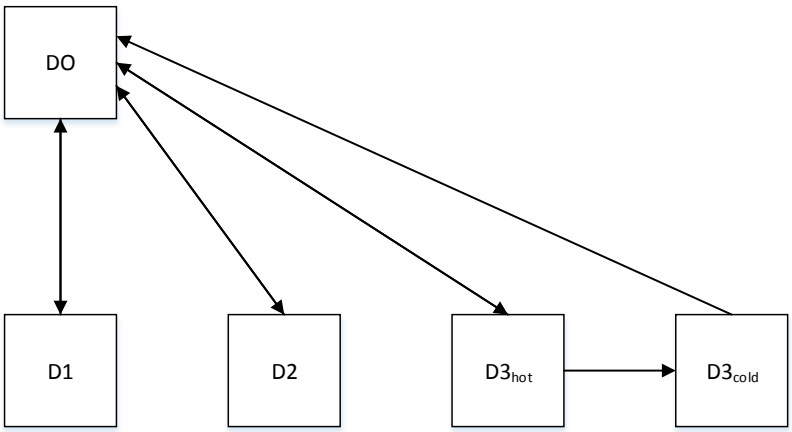
S5 – vypnutí

- Spotřeba elektrické energie – žádná
- Softwarové obnovení – nutný restart systému
- Hardwarová latence – nedefinovaná
- Systémový kontext – žádný kontext není zachován

3.8.2 Stavy napájení zařízení

Stavy napájení zařízení a jejich přechody jsou zobrazeny na obrázku 8. Jednotlivé stavy je možno definovat následujícími vlastnostmi.

- Spotřeba elektrické energie
- Kontext zařízení
- Funkce ovladače zařízení
- Čas obnovení
- Schopnost obnovení



Obrázek 8 – Stavy napájení zařízení

Stav D0

V tomto stavu je zařízení plně funkční, spotřebovává nejvíce energie, ovladač může provádět vstupně/výstupní operace a zařízení generuje přerušení. Změnu stavu napájení vždy zahajuje ovladač zařízení. Od verze Windows 8 může zařízení generovat přerušení bez ohledu na to, v jakém stavu se nachází. Ovladač pak může registrovat přerušení na úrovni IRQL PASSIVE_LEVEL, které slouží k přechodu do D0. Ve Windows předcházející verzi 8 (včetně Windows 7) nesmí zařízení generovat přerušení v jiném stavu než je D0 [5].

- Spotřeba elektrické energie – nejvyšší spotřeba
- Kontext zařízení – veškerý kontext zachován
- Funkce ovladače zařízení – normální funkce
- Čas obnovení – není aplikovatelné
- Schopnost obnovení – není aplikovatelné

Stav D1

- Spotřeba elektrické energie – menší než ve stavu D0, ale větší nebo rovna stavu D2. Zařízení v tomto stavu spotřebovává takové množství energie, aby si zachovalo svůj kontext.
- Kontext zařízení – zařízení si uchovává svůj kontext.
- Funkce ovladače zařízení – ovladač musí obnovit kontext, který byl ztracen v důsledku přechodu mezi stavy.
- Čas obnovení – obecně čas přechodu z D1 do D0 by měl být menší než z D2 do D0.
- Schopnost obnovení – zařízení ve stavu D1 může generovat wake-up signál.

Stav D2

- Spotřeba elektrické energie – menší nebo rovna stavu D1.
- Kontext zařízení – většina kontextu je ztracena, typicky si zařízení zachovává kontext pro signalizaci událostí probuzení.
- Funkce ovladače zařízení – ovladač musí uložit a obnovit veškerý kontext, který byl ztracen v důsledku přechodu mezi stavy.
- Čas obnovení – obecně čas přechodu z D1 do D0 by měl být menší než z D2 do D0.
- Schopnost obnovení – zařízení ve stavu D2 může generovat wake-up signál.

Stav D3

Stav D3 je nejnižší možná energetická náročnost zařízení. Od verze operačního systému Windows 8 je tento stav explicitně definovaný dvěma podstavy: D3cold a D3hot. Přestože předchozí verze Windows nedefinují tyto dva stavy, zařízení se implicitně nachází ve stavu D3hot v případě, kdy zařízení je v D3 a systém je ve stavu S0. Naproti tomu zařízení je ve stavu D3cold, když zařízení je ve stavu D3 a systém jako celek je ve stavu se sníženou spotřebou energie (S1 – S5) [5].

Od verze Windows 8 zařízení může přecházet mezi podstavy D3cold a D3hot, přestože systém jako celek zůstává ve stavu S0. Tyto podstavy jsou definovány následovně:

- **D3hot** – přechod do tohoto stavu nastává pouze z D0. Zařízení je na sběrnici, ke které je připojeno stále detekovatelné, proto sběrnice musí zůstat ve stavu D0. Tento přechod je iniciován ovladačem zařízení.
  - Spotřeba elektrické energie – absence napájení zařízení, ale počítač zůstává jako celek napájen.
  - Kontext zařízení – veškerý kontext je ztracen, ovladač jej musí obnovit, nebo znovu inicializovat zařízení.
  - Funkce ovladače zařízení – ovladač musí uložit a obnovit veškerý kontext.
  - Čas obnovení – typicky není o mnoho větší než z D2.
  - Schopnost obnovení – zařízení ve stavu D3hot může generovat wake-up signál.
- **D3cold** – přestože je zařízení fyzicky připojeno na sběrnici, přítomnost zařízení není detekovatelná. To může být zapříčiněno sběrnicí nebo zařízením, které se nachází v režimu snížené spotřeby.

- Spotřeba elektrické energie – absence napájení zařízení a popřípadě celého systému.
- Kontext zařízení – veškerý kontext je ztracen, ovladač jej musí obnovit, nebo znovu inicializovat zařízení.
- Funkce ovladače zařízení – ovladač musí uložit a obnovit veškerý kontext.
- Čas obnovení – nejvyšší
- Schopnost obnovení – zařízení ve stavu D3hot může generovat wake-up signál k probuzení celého počítače.

Přechod mezi stavy D3hot a D3cold není závislý na ovladači zařízení. Místo toho ovladač oznamuje systému schopnost přejít do stavu D3cold při přechodu z D0 do D3hot.

4 Windows Driver Foundation

4.1 Kernel-Mode Driver Foundation

Ovladače, které implementují model WDM, jsou podporovány i modelem KMDF, pokud vykonávají standardní I/O operace a manipulaci s IRP pakety. Model KMDF není vhodný pro ovladače, které přímo nevyužívají API jádra operačního systému, ale využívají zpětného volání již existujících ovladačů. Ovladače, které využívají svoje specifické funkce dispatch, jako jsou například IEEE 1394, ISA, PCI, PCMCIA, mohou být také implementovány v modelu KMDF [2].

Přestože KMDF ovladač poskytuje abstrakci nad modelem WDM, struktura ovladače je obdobná s obrázkem 3 v kapitole 3.2. Následující výčet uvádí povinné funkce, které musí každý KMDF ovladač implementovat.

- **Inicializační rutina** – Ovladač KMDF, stejně jako každý jiný ovladač, musí implementovat rutinu *DriverEntery*, která popíše daný ovladač frameworku, a tím jej inicializuje. Zařízení nepodporující Plug and Play této rutině vytváří *DeviceObject* [2].
- **Rutina AddDevice** – Činnost ovladače KMDF je založena na událostech a zpětných voláních. Zpětné volání *EvtDriverDeviceAdd* je jedno z nejdůležitějších, protože Plug and Play manažer využívá tuto rutinu při enumeraci nového zařízení podporovaného ovladačem. V této funkci je prováděna inicializace následujících komponent.
    - Vytvoření *DeviceObject*
    - Vytvoření I/O front pro příjem IRP paketů
    - Vytvoření rozhraní pro komunikaci s aplikacemi
    - Vytvoření rozhraní pro komunikaci mezi ovladači
    - Inicializace WMI
    - Vytvoření *InterruptObject* pro obsluhu přerušení
    - Povolení DMA

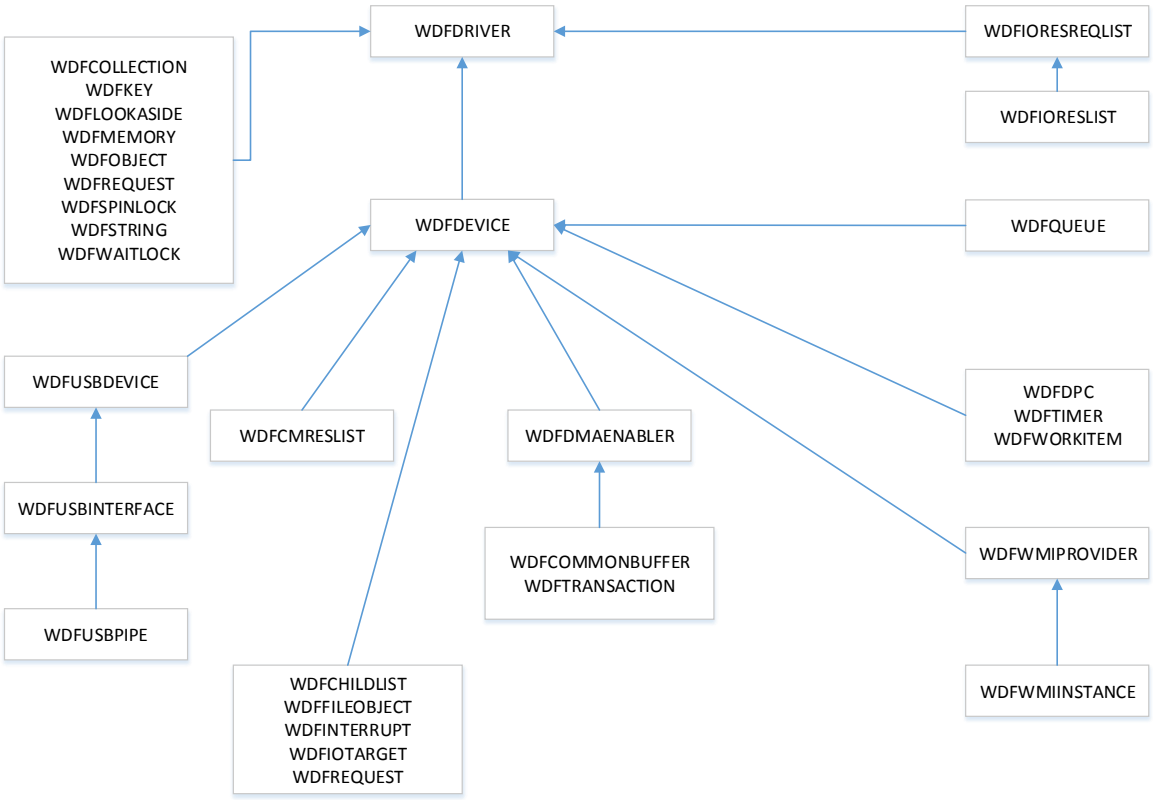
- **Rutiny EvtIo*** – Obdobně jako rutiny dispatch u WDM ovladačů využívají KMDF ovladače pro obsluhu specifických I/O požadavků tato zpětná volání. Ovladač typicky vytváří jednu nebo více konfigurovatelných front, do kterých KMDF umísťuje jednotlivé požadavky.

Nejjednodušší ovladač musí implementovat pouze inicializační a add-device rutiny, protože framework poskytne standardní obsluhu většiny typů I/O požadavků včetně Plug and Play a power managementu. Události odpovídají stavům systému, na které ovladač může reagovat nebo se účastnit. Pro události, které jsou pro ovladač kritické nebo potřebují speciální obsluhu, ovladač registruje svá zpětná volání.

4.2 KMDF Data Model

Model dat frameworku KMDF je objektově orientovaný. Ke správě objektů KMDF není využíván systémový správce objektů, ale objekty jsou spravovány v rámci KMDF. Ovladač pak pracuje s objekty pomocí referencí. Dále jsou poskytovány rutiny pro práci s danými objekty stejně jako metody set/get nebo assign/retrive.

Objekty modelu KMDF tvoří hierarchickou strukturu, což znamená, že většina objektů je spjata se svým rodičem. Kořenový objekt je struktura WDFDRIVER, která popisuje ovladač a je analogická struktuře DRIVER_OBJECT. Všechny další KMDF struktury jsou pak potomky WDFDRIVER. Na obrázku 9 je schematicky znázorněna hierarchická struktura objektů KMDF [3].



Obrázek 9 - Struktura objektu KMDF

Hierarchickou strukturu objektů je vhodné implementovat. Pokaždé, kdy je vytvořen potomek objektu, je přidána reference rodičovskému objektu, a proto při smazání rodičovského objektu jsou smazáni všichni jeho potomci. Proto je vhodné svázat i objekty jako jsou WDFSTRING nebo WDFMEMORY k danému objektu místo defaultního WDFDRIVER z důvodu automatického uvolnění paměti se zánikem rodičovského objektu.

Analogicky k *DeviceExtension* modelu WDM model KMDF definuje každému objektu *context*, který umožňuje ovladači připojit vlastní data vně frameworku. Framework umožňuje objektům vytvoření více *context*ů, čímž dovoluje odlišným vrstvám kódu ovladače pracovat se stejným objektem nezávisle, a tím umožňuje mechanismy podobné dědění.

4.3 KMDF I/O model

KMDF I/O model využívá obdobné mechanismy jako ovladače WDM. Framework KMDF vytváří své dispatch rutiny a obsluhuje veškerá IRP přijímaná ovladačem. Po zpracování požadavku jsou vytvořeny odpovídající objekty, které jsou posléze zařazeny do front, a ovladač je informován pomocí zpětného volání o výskytu události. Podle typu IRP KMDF provádí následující akce.

- Zašle IRP I/O handleru, který provádí standardní operace zařízení.
- Zašle IRP Plug and Play a power handleru, který zpracovávají tyto typy událostí a informují ostatní ovladače o změně stavu.
- Zašle IRP WMI handleru, který zpracovává sledování a logování.

Zpracování I/O požadavků je založeno na mechanismu frontování. Fronty modelu KMDF podporují mnoho funkcí – například sledování aktivních požadavků, podporu zrušení požadavku, synchronizaci a *I/O concurency* (umožňuje zpracování více I/O požadavků současně). Typicky ovladač KMDF vytváří nejméně jednu frontu, ke které přiřazuje události. Dále existuje několik možností nastavení front, které jsou uvedeny v následujícím výčtu.

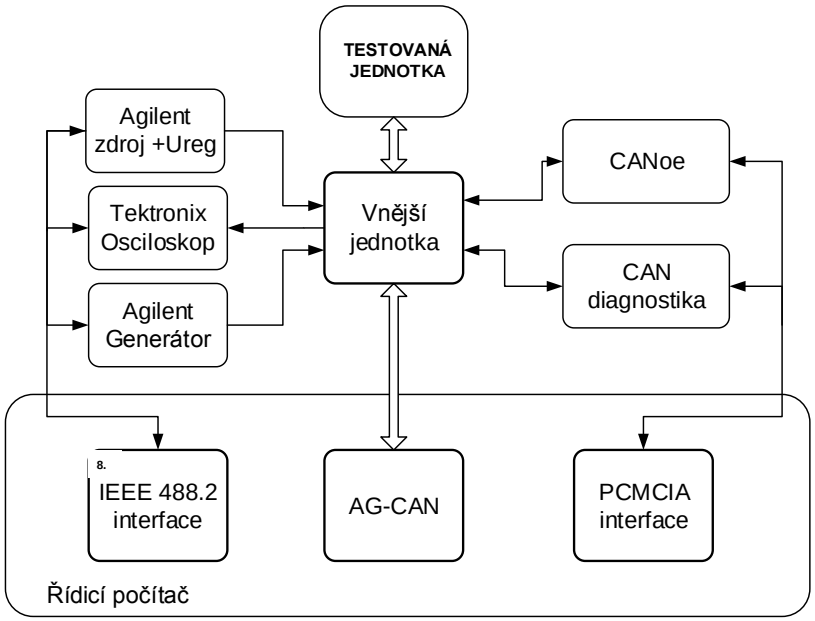
- Zpětná volání událostí, které jsou spojeny s frontou
- Stav správy napájení – I/O handler zajišťuje probuzení zařízení, informuje systém o nečinnosti a volá rutiny pro zrušení požadavků, pokud systém opouští stav working.

- Metoda odesílání požadavku – sekvenční – odeslání v jednu chvíli pouze jednoho požadavku (ovladač musí nejprve dokončit předchozí), paralelní – odeslání požadavku co nejdříve, manuální – ovladač si musí vyžádat požadavek z fronty sám.
- Možnost příjmu požadavku s nulovým bufferem – požadavek neobsahuje žádná data.

5 Návrh implementace

5.1 Testovací pracoviště

PCI karta testeru je součástí pracoviště, které komplexně testuje připojené zařízení na sběrnici CAN. Jak je patrné z obrázku 10, pracoviště se skládá z následujících komponent.



Obrázek 10 - Testovací pracoviště

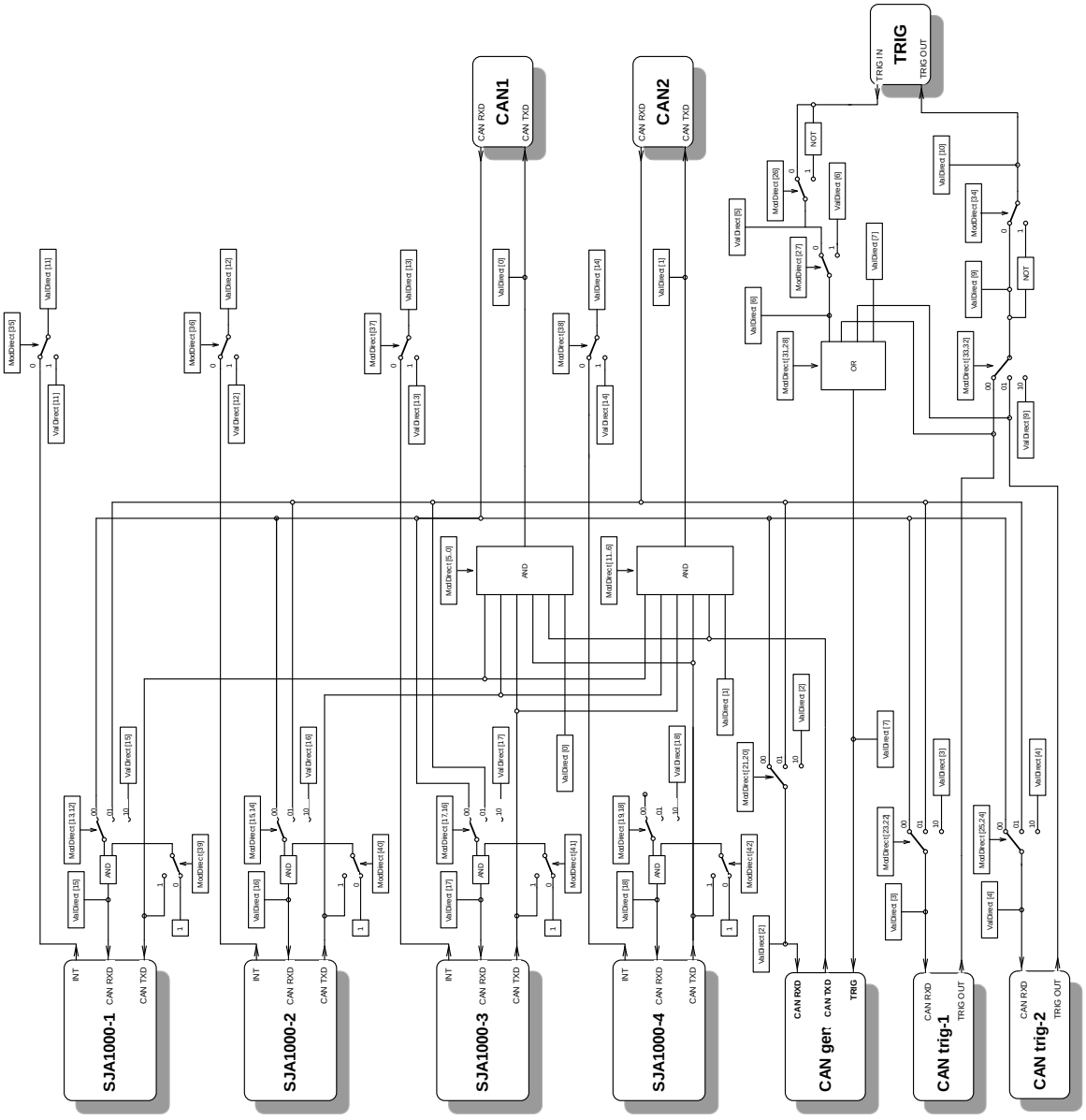
Testy jsou řízeny pomocí řídicího počítače. Měřicí přístroje jsou pak ovládány přes sběrnici GPIB a diagnostika pro CAN je řízena pomocí rozhraní PCMCIA.

Vnější jednotka plní zejména funkci přepínače mezi testovanou jednotkou a příslušnými měřicími přístroji a obsahuje další funkce jako napájení a ovládání CAN a LIN komunikace.

AG-CAN karta zastává funkci řízení vnější jednotky. Je na ní implementována čtveřice standardních řadičů CAN typu SJA1000, inteligentní CAN generátor a dvě sady trigrovacích jednotek, vše s časovým záchytným systémem. AG-CAN je v provedení PCI karty a je umístěna v řídicím počítači.

5.2 Popis funkce karty AG-CAN

Jak již bylo zmíněno v předchozí kapitole karta AG-CAN obsahuje několik funkčních bloků, které jsou schematicky znázorněny na obrázku 11 a které detailněji popisuje následující výčet.



Obrázek 11 - Funkční schéma PCI karty

- **Čtveřice řadičů CAN SJA1000** – obvody slouží k vysílání a příjmu rámců CAN. Obvody mají osmibitovou multiplexovanou sběrnici ovládanou stavovým automatem implementovaným v FPGA. Řadiče lze pomocí přepínačů nastavitelných pomocí řídicích registrů libovolně připojit na jednu ze dvou sběrnic CAN. Mapování registrů SJA do paměťového prostoru karty uvádí tabulka 4.
- **Generátor CAN** – generátor je určen k vysílání rámců s definovaným identifikátorem včetně rámců obsahující chyby. Generátor obsahuje instrukce, které lze využít k sestavení řídicího programu. Jednotlivé rámce jsou pak uloženy v paměti RAM. Lze také definovat délky jednotlivých bitů vysílaných na sběrnici. Detailnější popis funkce generátoru je popsán v [7].
- **Dvojice trigrovacích jednotek** – tato komponenta slouží k trigrování v případě zachycení zprávy CAN s definovaným identifikátorem. Jednotka obsahuje osm samostatných trigrovacích výstupů a jeden sumační výstup.

V hradlovém poli karty je dále implementován třicetidvoubitový čítač, který slouží jako časová základna pro řízení automatického odesílání zpráv CAN, které firmware karty podporuje pro všechna SJA.

Ze strany řídicího počítače se PCI karta jeví jako paměťová oblast o velkosti 1MB. Jednotlivé obvody jsou pak do paměti mapovány podle tabulky 4.

Tabulka 4 - Adresní prostor

pořadí	adresní prostor	Popis
1.	0x00000-0x000ff	Pole řídicích registrů
2.	0x20000-0x201ff	Obvod SJA1000-1
3.	0x20200-0x203ff	Obvod SJA1000-2
4.	0x20400-0x205ff	Obvod SJA1000-3
5.	0x20600-0x207ff	Obvod SJA1000-4
6.	0x40000-0x5ffff	CANgen – generátor CAN rámců
7.	0x60000-600ff	CANtrig1 – trigrovací jednotka od příjmu definovaného identifikátoru CAN zprávy.
8.	0x60100-601ff	CANtrig2 – trigrovací jednotka od příjmu definovaného identifikátoru CAN zprávy.
9.	0x80000-0x80fff	CANgen RAM, paměť generátoru CAN rámců

Z hlediska návrhu ovladače jsou dle požadavků, které jsou shrnuty v kapitole 5.4, nejdůležitější *řídicí registry* a registry obvodů SJA, do/ze kterých ovladač přímo zapisuje/čte.

5.3 Záchytná jednotka a přerušovací systém

Karta AG-CAN disponuje dvěma záchytnými jednotkami, které umožňují zachycení časové značky z volně běžícího čítače, jestliže nastane určitá události. Jednotlivé události jsou zřejmé z tabulky 5. Karta generuje přerušení na sběrnici PCI, pokud je daná událost povolena v maskovacích registrech karty a přerušení je povoleno globálně. Všechna přerušení jsou od záchytného systému, proto musí generování přerušení předcházet zachycení časové značky události. Zdroj přerušení lze posléze vyčíst z registru *Interrupt Status,* jehož struktura je patrná z tabulky 5.

Tabulka 5 - Struktura registru Interrupt Status

Bit	zdroj přerušení	popis
0	Tester Error	„vnější jednotka“ detekovala chybu konfigurace
1	Triger In	Aktivace trigeru na vstupu „vnější jednotky“
2	Triger Out	Aktivace trigeru na výstupu „vnější jednoky“
3	Config Done	Dokončení konfigurace „vnější jednotky“
4	SJA 1000-1	SJA1000-1 interrupt byl generován
5	SJA 1000-2	SJA1000-2 interrupt byl generován
6	SJA 1000-3	SJA1000-3 interrupt byl generován
7	SJA 1000-4	SJA1000-4 interrupt byl generován
8	CANTrig-1	Zachycení CAN zprávy s předdefinovaným ID modulem CANTrig-1
9	CANTrig-2	Zachycení CAN zprávy s předdefinovaným ID modulem CANTrig-2

5.4 Požadavky na funkce ovladače

Ovladač PCI karty AG-CAN má za úkol implementovat komunikaci mezi operačním systémem a aplikací s výše zmiňovanou kartou. Kromě implementace modelu WDM ovladač dále spravuje vysílání a příjem zpráv sběrnice CAN a umožňuje aplikaci zápis a čtení libovolné adresy paměti karty.

Jak již bylo zmíněno v předchozí kapitole, karta AG-CAN obsahuje čtyři standardní řadiče SJA1000. Pro všechny tyto řadiče ovladač zvlášť zajišťuje obsluhu frontovaní zpráv. Uživateli je umožněno nastavit každému řadiči libovolný počet vysílacích front, do kterých se frontují zprávy určené k odeslání. Přijaté zprávy jsou pak ukládány do přijímací fronty pro každé SJA. Každé zprávě je přiřazena časová značka a ovladač pak zajišťuje výběr a odeslání zprávy s nejnižší časovou značkou. Vysílanou zprávu lze označit příznakem zpětného příjmu, což zajistí uložení zprávy do přijímací fronty po odeslání. Tuto funkci je možno aktivovat globálně pro konkrétní frontu daného řadiče.

Uživatelská aplikace je informována o událostech, jako jsou: příjem zprávy, odeslaná zpráva z fronty s indexem nula a příjem jiného přerušení než od SJA.

Požadavky na aplikační rozhraní ovladače jsou shrnuty následujícími body, které ovladač a knihovna DLL implementuje.

- Zjištění verze ovladače
- Zápis hodnoty do paměti
- Čtení hodnoty z paměti
- Správa frontování
- Správa zpětného příjmu odeslané zprávy
- Zjištění počtu zpráv ve vysílacích frontách
- Zjištění počtu zpráv v přijímací frontě
- Odeslání/Příjem zprávy

6 Popis implementace

6.1 Implementace knihovny DLL

Tato kapitola popisuje implementaci knihovny DLL. Knihovna umožňuje aplikacím přistupovat ke službám ovladače pomocí exportovaných funkcí, jejichž výčet je uveden v závěru této kapitoly.

6.1.1 Datové typy

sMSG

Nejdůležitějším datovým typem používaným jak v ovladači, tak následně v knihovně DLL, je struktura sMSG, která je definovaná dle tabulky 6. Tato struktura reprezentuje zprávu sběrnice CAN a dále obsahuje časovou značku, která určuje čas vyslání/příjem zprávy vzhledem k časové základně třiceti dvoubitového čítače implementovaného na kartě, kód chyby a zbývající počet zpráv dostupných v přijímací frontě.

Tabulka 6 - Struktura sMSG

sMSG
id
dlc
data[8]
timeStamp
errCode
msgCount

s_Data

Struktura sloužící k přenosu adresy a dat mezi ovladačem a knihovnou DLL. Tuto strukturu využívají funkce *read* a *write* implementované knihovnou. Význam jednotlivých proměnných je zřejmý z tabulky 7.

Tabulka 7 - Struktura s_Data

s_Data
addr
value

sQueueCtrlData

Struktura je využívaná ovladačem a knihovnou k nastavení frontování. Je definovaná dle tabulky 8. Hodnota *ctrl* je binární proměnná určující zapnutí/vypnutí frontování. Proměněné *nbTQ* a *sjaID* v případě zapnutí specifikují počet aktivních front u daného SJA v případě zapnutí.

Tabulka 8 - Struktura sQueueCtrlData

sQueueCtrlData
ctrl
nbTQ
sjaID

sSetLoopbackData

Struktura sloužící k nastavení zpětného příjmu odeslané zprávy do přijímací fronty. Její definici uvádí tabulka 9. Proměnná *ctrl* zapíná/vypíná zpětný příjem zpráv. Pro specifikaci SJA a fronty jsou stále definované proměnné *tqID* a *sjaID.*

Tabulka 9 - Struktura sSetLoopbackData

sSetLoopbackData
ctrl
tqID
sjaID

sDEBUG_DATA

Struktura reprezentuje vybraná ladící data ovladače, viz tabulka 10. Patří sem počet volání DPC rutiny, počet volání DPC rutiny, které nevyvolalo přerušení od SJA, počet všech přijatých zpráv, počet přijatých chyb, počet ztracených zpráv a počet ztracených chyb.

Tabulka 10 - Struktura sDEBUG_DATA

sDEBUG_DATA
dpcCount
Dpc0Count
allCount
msgCount
errCount
lostMsgCount
lostErrCount

6.1.2 Komunikace s ovladačem

Knihovna DLL využívá ke komunikaci s ovladačem a přenosu dat výhradně kódů IOCTL. Detailněji tuto problematiku popisuje kapitola 3.7. Z knihovny je volána funkce Windows API *DeviceIoControl* s příslušnými parametry, která způsobí vytvoření IRP s hlavní funkcí IRP_MJ_DEVICE_CONTROL, která je následně zaslána ovladači. Kromě handlu na ovladač jsou v parametrech popsány oblasti vstupních a výstupních bufferů. Ovladač po přijetí tohoto IRP provede jednu z funkcí definovanou kódem. Funkce *DeviceIo Control* je volána jako synchronní, tedy volající vlákno je blokováno, dokud není operace dokončena. Seznam a význam jednotlivých kódů je podrobněji popsán v kapitole 0.

6.1.3 Funkce exportované DLL knihovnou

- *getDriverVersion* – tato funkce vrací verzi ovladače instalovaného v systému.
- *read8* – čte osmi bitovou hodnotu z paměti.
- *write8* – zapisuje osmi bitovou hodnotu do paměti.
- *read16* – čte šestnácti bitovou hodnotu z paměti.
- *write16* – zapisuje šestnácti bitovou hodnotu do paměti.
- *rea32* – čte třiceti dvoubitovou hodnotu z paměti.
- *write32* – zapisuje třiceti dvoubitovou hodnotu do paměti.
- *queueCtrl* – slouží k zapínání/vypínání frontování a specifikuje počet použitých vysílacích front.

- *setLoopback* – zapíná/vypíná zpětný příjem odeslaných zpráv z dané fronty, popřípadě lze zapnout loopback u všech front daného SJA najednou.
- *testRecQueue* – vrací počet zpráv v přijímací frontě paměti ovladače daného SJA.
- *testTrQueue* – vrací počet zpráv v dané vysílací frontě daného SJA, popřípadě počet zpráv ve všech frontách daného SJA.
- *getLastTrTimeStamp* – vrací časovou značku posledního odeslaného rámce.
- *readMsg* – vrací zprávu vyčtenou z přijímací fronty ovladače daného SJA.
- *writeMsg* – zapisuje zprávu do dané fronty daného SJA.
- *getEventHandle* – exportuje deskriptory událostí signalizovaných ovladačem popsaných v kapitole 6.2.4.

6.2 Implementace Ovladače

Jak již bylo řečeno v předchozí kapitole ovladač je implementován v modelu WDM. I když existuje novější framework KMDF, pro implementaci tohoto modelu je znalost a porozumění WDM nezbytná.

6.2.1 Implementace modelu WDM

V kapitole 3 byly popsány všechny náležitosti modelu WDM. Tato kapitola popisuje konkrétní implementaci modelu na ovladač vytvořený v rámci této práce.

DriverEntry

Vstupní rutina ovladače je *DriverEntry*. Ovladač zde vyplněním příslušných položek struktury *DriverObject* registruje obslužné rutiny jednotlivých IRP požadavků. Jak je patrné z následujícího kódu, ovladač zpracovává tyto požadavky: IRP_MJ_CREATE, IRP_MJ_CLOSE, IRP_MJ_DEVICE_CONTROL, IRP_MJ_PLUG AND PLAY, IRP_MJ_POWER.

```
NTSTATUS DriverEntry(
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath
    )
{
  DriverObject->MajorFunction[IRP_MJ_CREATE] = CanTestOpen;
  DriverObject->MajorFunction[IRP_MJ_CLOSE] = CanTestClose;
  DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] =
CanTestDeviceControl;
  DriverObject->MajorFunction[IRP_MJ_PLUG AND PLAY] = CanTestDispatchPlug
and Play;
  DriverObject->MajorFunction[IRP_MJ_POWER] = CanTestDispatchPower;
  DriverObject->DriverExtension->AddDevice = CanTestAddDevice;
  DriverObject->DriverUnload = CanTestUnload;
  return STATUS_SUCCESS;
      }
```

Dále jsou registrované funkce *AddDevice* a *DriverUnload*, které jsou volány správcem Plug and Play po vložení do systému, případně odstranění ze systému.

AddDevice

Jak již bylo nastíněno v kapitole 3.2, tuto funkci volá správce Plug and Play pro každé zařízení kontrolováné ovladačem při inicializaci systému a ve chvíli, kdy je přidáno zařízení nové. Úkolem této rutiny je vytvořit objekt *DeviceObject*, který reprezentuje dané zařízení a začlenit tento objekt do vrstvové struktury ovladačů. Reference na tento objekt je následně spolu s referencí *DeviceObject* nižšího ovladače uložena do *DeviceExtension*. Dále jsou inicializovány dva objekty typu *SpinLock*, které synchronizují přístup do vysílacích a přijímacích front, jejíž funkce je popsána v kapitole 6.2.2.

CanTestOpen

Tato rutina je volána po obdržení požadavku IRP_MJ_CREATE. Ten je zasílán správcem vstupně výstupních operací pokud byl vytvořen nebo otevřen soubor reprezentující daný ovladač. V tomto případě soubor vytváří knihovna DLL pomocí funkce *CreateFile*, jejímž parametrem je jméno *DeviceObject* vytvořeného ovladačem. Úkolem této rutiny je inicializace struktury *DeviceExtension* a vytvoření objektů synchronizačních událostí posaných v kapitole 6.2.4.

CanTestClose

Pokud jsou všechny handly k objektu *FileObject* reprezentující ovladač uzavřeny, správce vstupních a výstupních operací zasílá ovladači IRP IRP_MJ_CLOSE. Ten obsluhuje funkce *CanTestClose,* která je komplementární s funkcí *CanTestOpen*. Na tomto místě jsou tedy uzavřeny všechny handly na objekty synchronizačních událostí.

CanTestDispatchPlug and Play

Tato funkce obsluhuje požadavky s IRP kódem IRP_MJ_PLUG AND PLAY. Ten zasílá ovladači správce Plug and Play a pomocí minoritních kódů informuje ovladač o změnách stavu, které jsou podrobněji popsány v kapitole 3.6. Ovladač na základě minoritního kódu požadavku provede jednu z akcí popsaných níže a odešle požadavek ovladači, který se nachází ve vrstvové struktuře pod ním. Popis obsluhy jednotlivých minoritních kódů uvádí následující výčet.

- **IRP_MN_START_DEVICE** – Správce Plug and Play tímto způsobem informuje ovladač o enumeraci nového zařízení a o přidělení systémových prostředků. Ovladač volá funkci *CanTestStartDevice*, která zajistí uložení informací o přidělených zdrojích do struktury DeviceExtension. V tomto případě je ovladači přidělen zdroj typu Memory a Interrupt. V neposlední řadě je zde inicializovaná rutina dpc.
- **IRP_MN_QUERY_STOP_DEVICE** – Pokud je zapotřebí znovu přidělit systémové prostředky, správce Plug and Play zjišťuje, zda je možné zařízení zastavit. Protože ovladač neimplementuje žádané frontování IRP požadavků, které je zapotřebí v době zastavení zařízení, ovladač nastaví status požadavku na STATUS_UNSUCCESSFUL. Tím správci sděluje, že zařízení není schopno pozastavit svojí činnost. Z tohoto důvodu není vyžadována obsluha následujících požadavků:
  - IRP_MN_STOP_DEVICE
  - IRP_MN_CANCEL_STOP_DEVICE.
- **IRP_MN_QUERY_REMOVE_DEVICE** – Před odebráním zařízení ze systému je zaslán ovladači tento požadavek. V ovladači je tento požadavek schválen, čímž je sděleno správci Plug and Play, že zařízení je možno odebrat. Zároveň je požadavek přeposlán nižšímu ovladači.

- **IRP_MN_CANCEL_REMOVE_DEVICE** – Není zapotřebí žádná akce. Požadavek je jen přeposlán následujícímu ovladači.
- **IRP_MN_REMOVE_DEVICE** – Po odebrání zařízení ovladač obdrží tento požadavek. Zde je provedeno odregistrování přerušení, uvolnění fyzických adres z nestránkované paměti, odebrání ovladače z vrstvové struktury a smazání objektu *DeviceObject.*

CanTestUnload

Tato rutina je volána po odebrání všech zařízení řízených ovladačem. Tato rutina by měla uvolnit všechny prostředky, které inicializovala rutina *DriverEntery.* Přestože ovladač žádné prostředky neinicializoval, pro správnou funkci je nutné tuto rutinu implementovat. V tomto případě rutina pouze vrací hodnotu STATUS_SUCCESS.

6.2.2 Implementace frontování

Pro každý ze čtveřice obvodů SJA ovladač implementuje libovolný počet vysílacích front, který je shora omezen konstantou MAX_TQUEUE definovanou v hlavičkovém souboru *CanTestDrv.h.* K tomuto účelu je použita sada kruhových bufferů. Data reprezentující jednotlivé zprávy jsou pak uložena v datové struktuře sMSG, která je popsaná v kapitole 6.1.1.

Zápis do fronty obstarává obslužná rutina IOCTL kódu IOCTL_CanTest_WriteMsg. Vstupním bufferem je předávána struktura sMSG obsahující vysílanou zprávu. Specifikace indexu SJA a fronty je uložena v prvku errCode této struktury, přičemž spodních osm bitů odpovídá indexu fronty a horních osm bitů indexu SJA. Pokud jsou všechny fronty prázdné, ovladač zapíše zprávu přímo do registrů obvodu SJA a nastaví čas automatického vysílání, které je implementováno ve firmwaru karty dle časové značky zprávy. Současně uloží kopii zprávy do požadované fronty a SJA. V opačném případě je zpráva pouze uložena.

I když uživatelská aplikace zaručuje to, že jsou zprávy v jednotlivých frontách řazeny dle časové značky vzestupně, není tato podmínka splněna napříč všemi vysílacími frontami. Z tohoto důvodu může nastat situace, kdy zpráva, již připravená na odeslání a uložená v registrech SJA, má vyšší časovou značku, než zpráva nová. Ovladač pak čte z registrů *SJA1000-N_Write Timer* čas zbývající do odeslání a pokud je tento čas větší než jedna milisekunda, je zajištěno prohození zprávy v registrech SJA1000.

Pro příchozí zprávy je pro každou ze čtveřice SJA1000 implementovaná pouze jedna fronta. Zápis zpráv do fronty je uskutečněn na základě přerušení, které v tomto případě karta generuje. Tento mechanismus popisuje kapitola 6.2.3. Data jsou z fronty přenesena do uživatelského režimu po obdržení IOCTL kódu IOCTL_CanTest_ReadMsg. To zaručí, pokud je zpráva k dispozici, zkopírování jedné zprávy z přijímací fronty, případně vyčtené zprávy z registrů SJA (při vypnutém frontování) do výstupních bufferů IRP požadavku.

Informaci o počtu zpráv v odesílacích frontách poskytuje ovladač po obdržení IRP typu IRP_MJ_DEVICE_CONTROL i IOCTL kódem IOCTL_CanTest_TestTrQueue. Hodnota indexu SJA a fronty je předávána vstupním bufferem. Do výstupního bufferu je následně zkopírovaná hodnota odpovídající počtu zpráv v dané vysílací frontě daného SJA.

Počet zpráv v přijímací frontě je obdobně poskytnut po obdržení kódu IOCTL_CanTest_TestTrQueue, kdy je analogicky do výstupního bufferu zkopírovaná hodnota rovna počtu zpráv ve frontě přijímací daného SJA, které je určeno hodnotou vstupního bufferu I/O operace.

Ovladač podporuje možnost zpětného příjmu odeslané zprávy, kdy je právě vyslaná zpráva zkopírovaná v rámci ovladače zpět do přijímací fronty. Tuto možnost lze nastavit separátně u jednotlivých vysílacích front pomocí funkce DLL knihovny *setLoopback,* která vytvoří požadavek s IOCTL kódem IOCTL_CanTest_SetLoopback, což má za následek nastavení příznaku zpětného příjmu v ovladači. Tuto funkcionalitu lze definovat i na úrovni zprávy pomocí logického součtu identifikátoru zprávy s konstantou MSG_LOOP, která je definovaná v hlavičkovém souboru *CanTestDrv.h*. K rozlišení zpráv takto přijatých je do proměnné errCode přidán příznak MSG_REC, který je definován ve stejném souboru.

6.2.3 Rutina DPC

Jak již bylo řečeno, PCI karta informuje ovladač o událostech prostřednictvím přerušení, pokud je daný zdroj přerušení povolen. Pro implementaci frontování popsaného v předchozí kapitole je nezbytné reagovat na přerušení zejména od obvodů SJA. Ty mohou generovat přerušení v situacích popsaných v [8]. Ovladač reaguje pouze na následující události.

- **Receive interrupt** – toto přerušení nastává, pokud je přítomna zpráva ve FIFO řadiče.

- **Transmit Interrupt** – přerušení je generováno v případě, odeslání zprávy řadičem
- **Bus Error Interrupt** – přerušení nastává, pokud řadič detekuje chybu na sběrnici CAN

Po přijetí přerušení na sběrnici PCI je naplánovaná rutina DPC a generování přerušení je do doby obsloužení zakázáno. Čtením registru *Interrupt status,* popsaném tabulkou 5, je zjištěno, který zdroj toto přerušení vyvolal. Pokud je přerušení vyvoláno jedním z řadičů SJA, je pro rozlišení typu události čten *Interupt Register* z obvodu SJA. Pomocí záchytné jednotky je uložen čas, kdy je generováno přerušení a ten je možno vyčíst s registrů karty. Tento čas je pak použit jako časová značka přijaté zprávy. Pokud je generováno přerušení na sběrnici PCI, ale čtením registru *Interupt Register* není detekováno přerušení od SJA, je nastavena událost CanTestIrqEvent a generování dalšího přerušení je zakázáno.

Receive interrupt

Toto přerušení je generováno v případě přítomnosti zprávy v paměti FIFO řadiče. Počet dostupných zpráv v řadiči udává registr RX message counter. Ovladač pak ve smyčce ukládá jednotlivé přijaté zprávy do přijímací fronty toho SJA, které přerušení generovalo. Po přijaté zprávě je nastavena synchronizační událost CanTestMsgEvent do signálního stavu.

Transmit interrupt

Řadič SJA generuje toto přerušení, pokud je úspěšně vyslaná zpráva na sběrnici CAN. Ovladač kontroluje, zda je frontě, ze které byla zpráva odeslána, nastaven příznak zpětného příjmu nebo je tento příznak nastaven na úrovni zprávy. Pokud je příznak nastaven, je kopie zprávy přidána do přijímací fronty. Pomocí cyklu jsou kontrolovány všechny aktivní fronty daného SJA a je vybrána fronta obsahující zprávu s nejnižší časovou značkou. Zde se předpokládá, že zprávy jsou řazeny v jednotlivých frontách dle časové značky vzestupně. Zpráva je pak zkopírovaná do registrů SJA a pokud je zpráva vyslaná z fronty s indexem nula, je nastavena synchronizační událost CanTestBufEvent do signálního stavu.

Error bus Interrupt

Pokud se na sběrnici během přenosu zprávy vyskytne chyba, řadič generuje toto přerušení. Zpráva je z řadiče uložena do přijímací fronty a informace o nastalé chybě

je vyčtena z registru řadiče error code capture do errCode struktury sMSG. Význam jednotlivých kódů lze nalézt v [8].

6.2.4 Synchronizační události

Ovladač spolu s knihovnou umožňují informovat uživatelskou aplikaci, pokud nastane některá z událostí, které byly definované v požadavcích na ovladač v kapitole 5.4. Tato funkcionalita je zabezpečena pomocí synchronizačních událostí, které jsou vytvořeny v jádru systému pomocí funkce *IoCreateSynchronizationEvent*. Pro každé SJA ovladač vytvoří trojici pojmenovaných událostí, které jsou popsány v tabulce 11.

Tabulka 11 - Synchronizační události

CanTestIrqEvent	Karta generuje přerušení jiné než od SJA
CanTestMsgEvent	Příjem zprávy, zápis zprávy do přijímací fronty následkem zpětného příjmu
CanTestBufEvent	Odeslání zprávy z fronty s indexem 0

Knihovna DLL pak získá na tyto události handle pomocí funkce *CreateEvent,* jejímž parametrem je právě jméno události vytvořené v ovladači. Takto získané handly poskytuje uživatelské aplikaci skrze exportovanou funkci *getEventHandle.*

Ovladač jádra vytváří objekt události ve jmenném prostoru \BaseNamedObject. Od verze operačního systému Windows Vista aplikace běžící v uživatelském režimu nesdílí stejný jmenný prostor s jádrem operačního systému z důvodu existence velkého množství objektů a případné kolize jmen. Standardně je pro každého uživatele vytvořena tzv. relace (Session), vyhrazená pro objekty daného uživatele. Aby knihovna byla schopna získat handle objektu události vytvořeného v režimu jádra, musí jméno události využít prefix \Global.

IOCTL kódy

Tato kapitola uvádí pro úplnost všechny IOCTL kódy, které ovladač podporuje, a stručně popisuje jejich význam.

IOCTL_CanTest_GetDriverVersion

Po obdržení tohoto kódu je vrácena aktuální verze ovladače instalovaného v systému.

IOCTL_CanTest_ReadX a IOCTL_CanTest_WriteX

Tento požadavek zajistí čtení respektive zápis hodnoty z adresového prostoru PCI karty. Data a adresa jsou mezi knihovnou a ovladačem předávány pomocí struktury s_Data, která je popsána v kapitole 6.1.1.

IOCTL_CanTest_LastTrTimeStamp

Požadavek do bufferu příslušného IRP zkopíruje časovou značku poslední zprávy odeslané z daného SJA.

IOCTL_CanTest_SetLoopback

Žádost vypíná/zapíná zpětný příjem odeslaných zpráv. Data jsou přenášena pomocí struktury sSetLoopbackData.

IOCTL_CanTest_QueueCtrl

Žádost vypíná/zapíná frontování. V případě zapnutí je ovladač uveden do výchozího stavu, kdy jsou všechny vnitřní buffery a všechny doplňující informace ovladače vymazány.

IOCTL_CanTest_TestRecQueue

Požadavek do výstupního bufferu zkopíruje hodnotu, která odpovídá počtu zpráv v příchozí frontě daného SJA.

IOCTL_CanTest_TestTrQueue

Požadavek do výstupního bufferu zkopíruje počet zpráv v odchozí frontě. Obslužná funkce také kontroluje přítomnost zprávy již nahrané v registrech řadiče SJA.

IOCTL_CanTest_ReadMsg

Tento požadavek zajistí zkopírování přijaté zprávy z ovladače do výstupního bufferu IRP požadavku.

IOCTL_CanTest_ReadDebugData

Požadavek vrací ladící informace ovladače pomocí struktury sDEBUG_DATA. Popis této struktury je v kapitole 6.1.1.

IOCTL_CanTest_WriteMsg

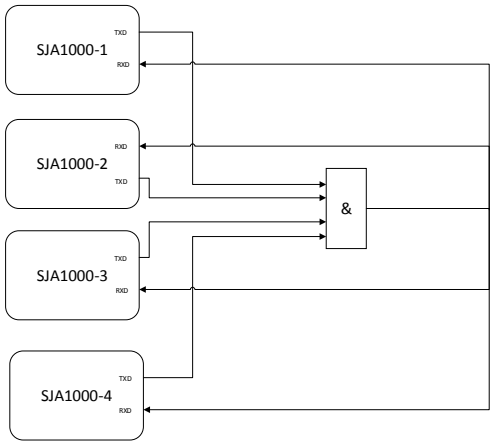
Požadavek zajistí zápis zprávy do dané fronty daného SJA a následné odeslání.

7 Testování

Nedílnou součástí vývoje softwaru je otestování jeho funkčnosti. To platí zejména u ovladačů, které jsou využívány dalšími vrstvami softwaru, jako jsou aplikace. Zde je spolehlivost klíčová, protože případné chyby jsou velmi těžko odhalitelné. Proto byla v rámci této práce vytvořena jednoduchá konzolová aplikace, která testuje zejména funkčnost frontování, příjem a odesílání zpráv.

7.1 Testovací aplikace

Aby bylo možno testovat zasílání zpráv na sběrnici CAN v rámci karty, byla do hradlového pole nahrána modifikovaná verze firmwaru. Tato verze propojuje vstupy řadičů SJA1000 s výstupy, které jsou logicky vynásobeny. Tuto situaci popisuje obrázek 12.



Obrázek 12 - Propojení TXD a RXD

Aplikace umožňuje testovat dva aspekty frontování. V první řadě ověřuje funkčnost odesílání zpráv. V tomto případě je jeden řadič SJA1000 zvolen jako referenční, který pouze přijímá zprávy od ostatních řadičů. Zbývající tři řadiče mají aktivovány tři odesílací fronty, do kterých jsou náhodně ukládány zprávy s náhodným identifikátorem. Je zde zabezpečeno, aby časová značka v jednotlivých frontách měla rostoucí charakter. Po přijetí zprávy referenčním řadičem je kontrolována shoda identifikátoru a dat zprávy.

Druhý způsob ověření funkčnosti spočívá v testování front přijímacích. Zde jsou zprávy odesílány pouze z jednoho řadiče a přijímány zbylou trojicí řadičů. Kromě kontroly shody identifikátoru musí také souhlasit čas přijetí zprávy u všech řadičů.

7.2 Výsledky testování

Ovladač byl otestován výše zmíněným způsobem. Po správném přijetí byla inkrementována proměnná reprezentující počet bezchybných přijetí zpráv. Oba testy byly spuštěny po dobu jedné hodiny. Během této doby nebyla aplikací zaregistrována jediná chyba. Lze tedy konstatovat funkční implementaci výše zmiňovaných požadavků.

8 Závěr

Cílem této práce bylo navrhnout a implementovat ovladač pro modifikovanou verzi PCI karty testeru sběrnice CAN pro operační systém Microsoft Windows 7. V rámci práce byl vytvořen ovladač jádra operačního systému podle modelu WDM, který zajišťuje podporu čtveřice obvodů SJA1000, které tato nová verze karty obsahuje. Podpora spočívá zejména v implementaci frontování zpráv. Služby ovladače jsou přístupné prostřednictvím knihovny DLL, která exportuje sadu funkcí.

Funkčnost byla ověřena pomocí testovací aplikace, jejíž vytvoření bylo dílčím cílem práce. V rámci aplikace byl úspěšně otestován příjem a odesílání zpráv z více front s ohledem na časovou značku. Testování probíhalo na platformě třiceti dvoubitové verze Microsoft Windows 7.

Dalším krokem pro vylepšení stávajícího řešení je implementovat ovladač ve frameworku Windows Driver Foundation, což by zaručilo kompatibilitu zdrojového kódu ovladače s nadcházejícím verzemi systému Windows.

9 Literatura

[1] DRÁP, Martin. *Jádro systému windows*. Brno: ComputerPress, a.s., 2011. ISBN 979-80-251-2731-5.

[2] RUSSINOVICH, Mark E, David A SOLOMON a Alex IONESCU. *Windows internals*. 6th ed. Redmond: Microsoft Press, 2012, xxii, 726 s. ISBN 978-0-7356-4873-9.

[3] ORWICK, Penny, David A SOLOMON a Alex IONESCU. *Developing drivers with the Windows driver foundations*. 6th ed. Remond: Microsoft Press, c2007, xxviii, 896 s. ISBN 978-0-7356-2374-3.

[4] ONEY, Walter, David A SOLOMON a Alex IONESCU. *Programming the Microsoft Windows driver model*. 2nd ed. Washington: Microsoft Press, 2003, xxviii, 846 s. ISBN 07-356-1803-8.

[5] Writing WDM Drivers. *Writing WDM Drivers* [online]. Dostupné z: http://msdn.microsoft.com/en-us/library/windows/hardware/ff566412(v=vs.85).aspx

[6] *Dokumentace Microsoft Windows Driver Kit*

[7] *Programátorský model CanTest 3.0*

[8] *Datasheet SJA1000*