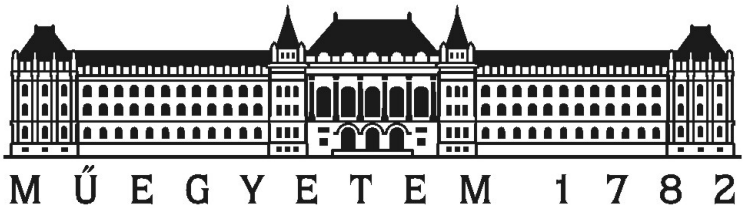


Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Szélessávú Hírközlés és Villamoságtan Tanszék

TDK dolgozat

Szoftverteljesítmény és optimalizáció vizsgálata kernel módú eszközkezelő programmal

Készítette:
Hajnal Erik
pit6dc

Konzulens:
dr. Csurgai-Horváth László
BME-HVT

Budapest

2012.

Tartalomjegyzék

1 Bevezetés

1.1 A dolgozat témájának, céljának ismertetése

A mai világunkban teljesen természetes, hogy a szoftverfejlesztők a szoftvereket magas szintű programozási nyelvek és vizuális fejlesztőeszközök segítségével készítik. Azonban ehhez a mögöttes technológia, a magas nyelvű programokat futtató alacsony szintű keretrendszerek magas fokú optimalizáltsága szükséges – vagy nagyon erős hardver. Rengeteg apróbb alkalmazásnál szinte észrevehetetlen a különbség egy alacsony és egy magas szintű nyelv segítségével készített program között a mai rendkívül gyors hardvereknek köszönhetően. Viszont a teljesítményigényes alkalmazásoknál, amiket az imént említett technológiák segítségével manapság nagyon gyorsan el lehet készíteni, sokszor tapasztalhatunk problémákat a sebességgel.

Ezen dolgozat alapvető kérdése az, hogy valóban nyugodt szívvel felejthetjük el a különböző optimalizációs lehetőségeket és bízhatunk mindent a fordítónkra vagy esetleg vannak-e bizonyos lehetőségeink, amelyek akár újak (például az új C++11 szabvány), akár régiek, ám mára kiszorultak a fókuszból (például a fordítási kapcsolók).

Azonban adja magát a kérdés: elég "okosak" a fordítóink? Valóban ki tudnak találni maguktól mindent? Vagy lehet hogy érdemes lenne bizonyos beállításokat manuálisan elvégezni, ami esetleg javíthatna az alkalmazásaink teljesítményén? Természetesen a szoftverfejlesztés átalakulásának folyamatát megállítani nem lehet, a célunk sokkal inkább pár olyan még / már kevésbé ismert lehetőség megvizsgálása, amelyekkel az alkalmazásaink teljesítményét lehetne javítani.

A szoftver eszközök minősítését számítógép-hardverrel támogatott mérésekkel kívánjuk elvégezni. Ehhez egy, az RTD-USA által készített DM7520 típusú, PCI buszra illeszkedő mérés-adatgyűjtő kártyát használunk fel, amelyhez saját illesztőprogram is készült a nagyobb flexibilitás érdekében. Ezáltal a szoftver futási teljesítményét egzakt módon, a hardver külső portjain kiadott impulzusok műszeres mérésével fogjuk elvégezni.

Egy ilyen témában nagyon nehéz teljes körű, minden platform - fejlesztőeszköz párosítást megvizsgálni, ezért a mi választásunk a Windows 7 (x64) platformra és a Visual Studio legújabb, 2012-es változatára esett. Emellett egy-egy rövid összehasonlítás kedvéért a GCC által előállított kódot is megvizsgáltuk, ám ezzel nem foglalkoztunk részleteiben.

A dolgozat eredményei segíthetnek mélyebben megérteni a programfejlesztési eszközök beállításainak hatásait, valamint az optimalizációs lehetőségek kihasználhatóságát.

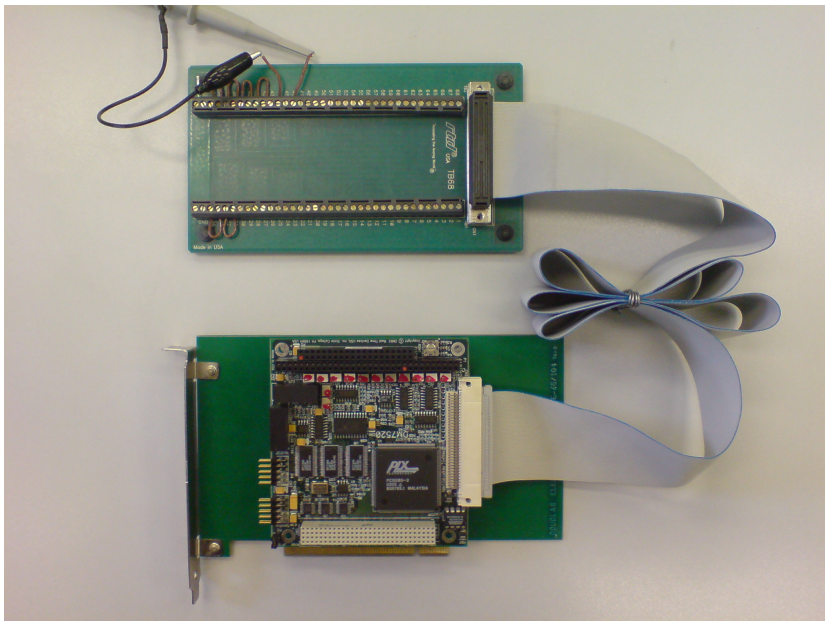
A dolgozat két nagyobb részre bontható: az első részben a mérésekhez vezető út bemutatása található. Egy áttekintés a kernel-módú programozásról, annak előnyeiről és hátrányairól, a driverfejlesztésről, a mérés menetéről.

A második rész különböző lehetőségeket próbál feltárni a performancia növelése érdekében. Górcső alá kerül a Windows prioritáskezelése, különböző kevésbé ismert, keveset használt fordítási kapcsoló, és a C++ világ legújabb áttörése, a C++11 szabvány nyújtotta optimalizációs lehetőségek is.

1.2 A mérési körülmények rövid ismertetése

Mivel a mérések során pont a nagyon apró, ám mégis jelen lévő sebességkülönbségeket szeretném kimutatni, a Windows által biztosított szoftveres időmérés pontossága nem elegendő. Ezért a mérésekhez egy data acquisition (mérés-adatgyűjtő) kártyát használunk, amelyhez egy - kifejezetten erre a célra - általunk fejlesztett illesztőprogramot használunk.

A kártya egy, a RealTime Devices Inc. által készített DM7520-as típusú mérés-adatgyűjtő kártya. Ez egy PC/104 formátumú eszköz, amelyet azért választottunk a mérések elvégzésére, mert mind a hardver, mind a szoftver dokumentációja rendelkezésünkre állt, ami nélkülözhetetlen volt a később ismertetésre kerülő saját kernel-módú driver megírásához. Mivel a kártyát asztali számítógépben használtuk, egy PCI-PC/104 konverter segítségével helyeztük be az asztali gépbe. A mérés-adatgyűjtő kártya ki/bemenetei egy I/O terminál kártyán keresztül érhetők el, amelyre már tetszőleges külső mérőeszköz csatlakoztatható. Az 1. ábrán látható a DM7520 kártya a PCI konvertermodulra szerelve, továbbá a hozzá csatlakoztatott I/O modul.



1. ábra A mérés-adatgyűjtő kártya PCI konverterrel és terminál-kártyával

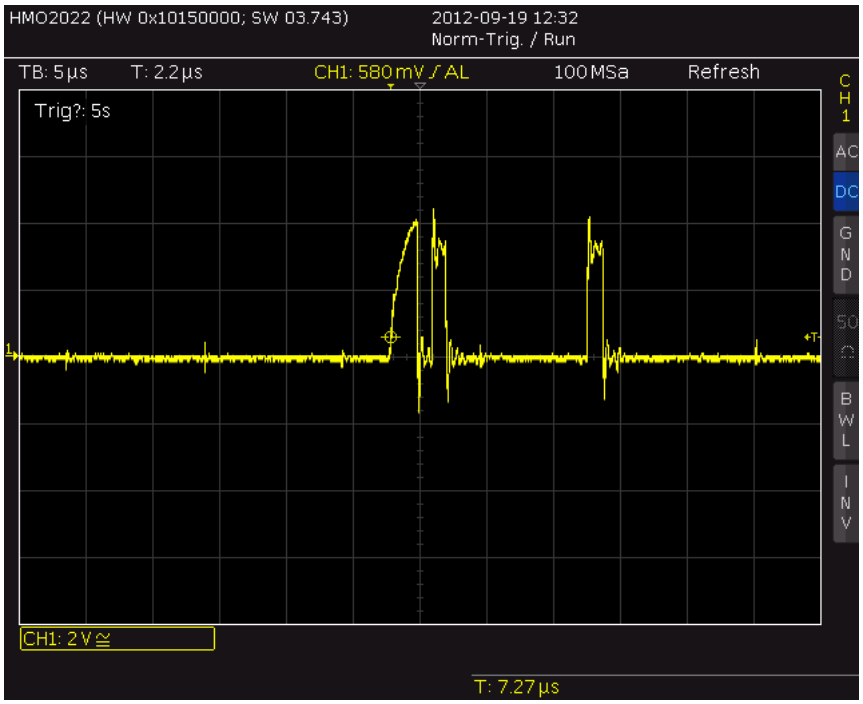
A DM7520 egy univerzális, PCI buszra illesztett mérés-adatgyűjtő eszköz, amely analóg ki- és bemenetekkel rendelkezik, univerzális időzítő áramköröket tartalmaz, digitális I/O portjai és még számos egyéb, általunk nem használt erőforrása van.

A dolgozatban kitűzött célok eléréséhez csupán a digitális I/O portok használatára volt szükség, amelyeknek a korlátlan hozzáférését biztosítja a későbbiekben ismertetésre kerülő eszközmeghajtó szoftver.

A kártya digitális I/O modulja két, egyenként nyolc bites csatornát biztosít számunkra, ami a méréseinkben kulcsszerepet tölt be. Ezek a portok a PCI

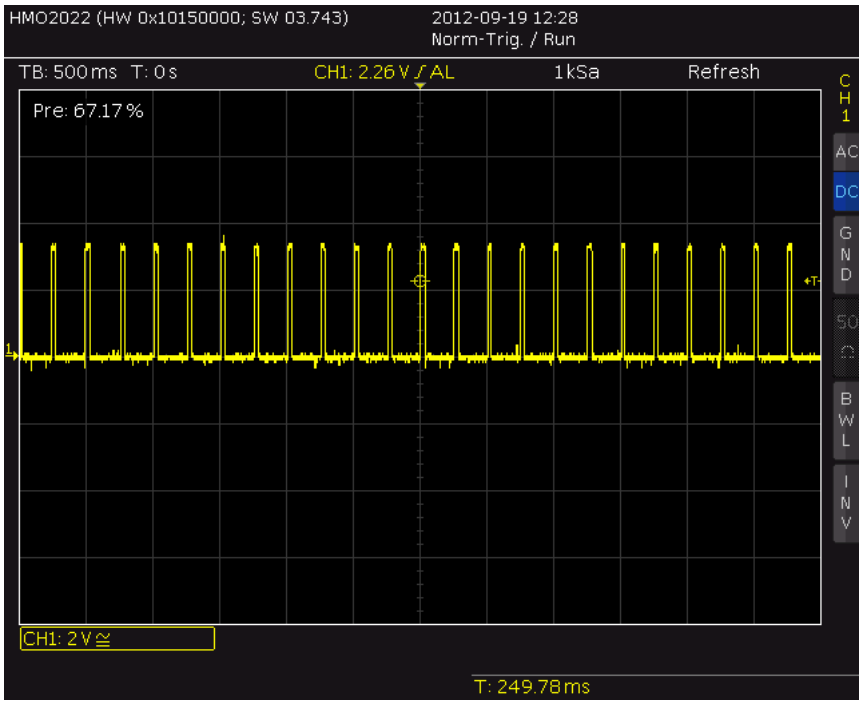
interfészen keresztül memóriába ágyazott eszközként érhetők el, és kimenetként programozva logikai 1 vagy 0 érték írható ki rájuk. A logikai jel porton történő megjelenését a kártya hardver felépítése csak minimális mértékben korlátozza, éppen a memóriába ágyazott működés miatt. A PCI busz arbitrációja ugyanakkor nagyobb mértékű, ráadásul előre nem kiszámítható késleltetést visz be a rendszerbe, de ennek az ideje is nagyságrendekkel kisebb, mint a tesztelni kívánt szoftverek futási idejei. Mindazonáltal a teszthardver az alaplapi vezérlőkön kívül semmilyen egyéb PCI eszközt nem tartalmazott, hogy az ebből adódó mérési hibákat kiküszöbölhessük.

A hardveres időmérés menete a következő: A kártyához egy I/O egységet kapcsolunk, amelynek az egyik kimenetét rákötjük egy oszcilloszkópra. A mérendő tartományba való belépéskor és annak elhagyásakor a kimeneten megjelenítünk egy impulzust, majd a két impulzus közötti távolságot leolvasva megkapjuk a mérni kívánt időt. Természetesen az impulzus kiadása és az aktuális algoritmus között még eltelik egy kevés idő amíg az irányítás visszakerül a driverből a mérőprogramba, azonban ezt az időt is le tudjuk mérni és ezáltal pontosítani a mérésünket.



2. ábra - Egy mérés kinézete

Az 1. ábrán egy konkrét mérésnek a megjelenését látjuk az oszcilloszkópon. Az első impulzus az előző mérés végét jelzi (a mérések ciklusban futnak, így könnyebben mérhetőek a kontextusváltásból származó késleltetések). A második impulzus jelzi az aktuális mérés kezdetét, az utolsó pedig a mérés végét. A ciklusban futó méréseket a 2. ábrán látható módon gyűjtjük össze.



3. ábra - A mérések összegyűjtése

Itt már nem látszik a mérések közötti nanoszekundumos szünet, azonban milliszekundumban mérhető értékeknél ez elhanyagolható mérési hiba. Az oszcilloszkóp időtartományának megfelelő mértékű növelésével elérjük, hogy 5-10 mérést lássunk egyszerre, majd a mért idők átlagolásával meghatározzuk a végső időt.

2 A mérések előkészítése

2.1 Kernel módú programozás - áttekintés

Mielőtt a kártyához megírt illesztőprogramot bemutatnánk, nyújtanánk egy rövid áttekintést a kernel módban való programozásról, annak előnyeiről és hátrányairól.

Kernel módban sokkal nagyobb a hatalma, és ezáltal a felelőssége is a programozónak. [1] Nem lehet a "trial-and-error" megközelítést alkalmazni, ugyanis itt a "Program működése leállt" képernyő helyett minden apró hiba STOP hibához (ismertebb nevein: kék halál, Blue Screen of Death) vezet. Lokális gépen dolgozva el kell felejteni a 21. századi debug eszközöket; még egy hibakódot sem lehet kiírni a képernyőre, hiszen a STOP hiba azonnal megjelenik.

A debug eszközökhöz hasonlóan búcsút kell inteni a CRT-nek (C RunTime), amelyet alapból minden felhasználói-módú program használ, ugyanis alapból a *main*, illetve *WinMain* függvényeinket a CRT hívja meg.

A CRT által biztosított, széles körben ismert és használt függvények és osztályok (printf, fopen, cout, vector, list, stb.) bár közvetlenül nem elérhetőek kernel-módból, rengetek CRT-beli függvény előbb-utóbb egy kernel-módú API hívásként végzi, így sokszor elegendő némi kutatás a megoldáshoz. Például a *printf* függvény kernel-módú megfelelője a *DbgPrint* függvény, még a paraméterei is ugyanazok (minimális különbségekkel).

A Windows programozásban jártas emberek által ismert Windows API sem elérhető, azonban ezen hívások két részre oszthatóak. Az első csoport olyan hívásokat tartalmaz, amelyek azonnal, vagy minimális ellenőrzéseket követően továbbítja a hívást a kernelbe. Ezeket a függvényeket a *printf*-hez hasonlóan lehet használni. Ebbe a csoportba tartoznak például a fájlokkal kapcsolatos függvények.

A második csoportban olyan függvények találhatóak, amelyek a kernel-módú fejlesztésnél irrelevánsak. Ilyen például a multimédiával vagy az ablakokkal kapcsolatos hívások. Ezen funkciókra kernel-módban egyszerűen nincs szükség, így nagy részüknek nincs is kernel-mód béli megfelelőjük.

A legnagyobb API, ami a rendelkezésünkre áll, az az úgynevezett Native API, ami a Windows API (röviden WINAPI) alatt helyezkedik el. A Native API-ban rengeteg alul-dokumentált vagy egyáltalán nem dokumentált függvény található, ami tovább nehezíti a kernel-módú fejlesztést.

2.2 A driverfejlesztés alapjai, legfontosabb lépései

A driverfejesztés sokrétű feladat. Meg kell valósítani a hardver és a Windows közötti kommunikációt, amihez a hardver részletesebb ismerete szükséges. Implementálni kell az eszköz működéséhez és kezeléséhez szükséges logikát, egyfajta belső függvénycsoportot. Létre kell hozni egy API-t, aminek segítségével a felhasználói-módú programok kommunikálhatnak a driverrel. Szükség van továbbá a drivernek a "leírására", egy .inf fájlra, amely segítségével a Windows tudja, hogy pontosan mit is tartalmaz az illesztőprogram. Ezen felül egy felhasználói módú API-t is érdemes készíteni, ugyanis felhasználói-módból a kernel-módú API-nkat csak a *DeviceIoControl*

nevű függvényen keresztül érhetjük el, ami semminemű kényelmet illetve validációt nem biztosít.

2.2.1 A driver szolgáltatás elkészítése

Az illesztőprogramok a Windows számára service-ekként, magyarul szolgáltatásokként jelennek meg. Az elkészült driver a legminimalisztikusabb esetben két fájlként jelenik meg. Lesz egy .inf fájlunk, ami leírja az illesztőprogramot, a másik pedig egy .sys fájl, ami a futtatható kódot tartalmazza. Ez a rész a .sys fájl elkészítésébe nyújt betekintést.

A Native API-ra épülő alkalmazások, így az illesztőprogramok belépési pontja is alapértelmezés szerint a *DriverEntry* függvény, melynek szignatúrája a következő:

```
1 NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObject, PUNICODE_STRING
      pRegistryPath)
```

Ehelyett azonban a legtöbb helyen ezt lehet látni:

```
1 extern "C" NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObject,
      PUNICODE_STRING pRegistryPath)
```

Ennek az oka az, hogy a drivereket külön fordítóprogramokkal kell lefordítani, amelyek viszont alapból C nyelven várják a forráskódot. Azonban a C++ nyújtotta rugalmasság (például a változók "akárhol" megengedett deklarációja) miatt rendszerint mégis C++ lesz a választott nyelv, ebben az esetben azonban gondoskodni kell róla, hogy a *DriverEntry* függvény C függvénynek megfelelő dekorációval kerüljön bele a .sys fájlba.

Az első paraméter a driverünk viselkedését szabályozó *DRIVER_OBJECT* struktúrára egy mutató, a második pedig a driverünk registry kulcsához az elérési út.

A *DriverEntry* függvény feladata összekapcsolni a Windows-t a driverrel. A *DriverEntry* futásakor az illesztőprogramunk még nem működésre kész, hanem pont ekkor készül az indulásra, ebben hivatott a *DriverEntry* segítő kezet nyújtani a Windows számára.

A visszatérési érték egy NTSTATUS, ami longként van definiálva, ezen keresztül tudjuk jelezni a műveleteink sikerességét.

A mi konkrét *DriverEntry*-nk a következőképpen néz ki:

```
1  extern "C" NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObject,
       PUNICODE_STRING pRegistryPath)
2  {
3    pDriverObject->DriverUnload                    = DM7520_DriverUnload;
4    pDriverObject->DriverExtension->AddDevice      = DM7520_AddDevice;
5    pDriverObject->MajorFunction[IRP_MJ_PNP]       = DM7520_PnP;
6    pDriverObject->MajorFunction[IRP_MJ_CLOSE]     = DM7520_Close;
7    pDriverObject->MajorFunction[IRP_MJ_CREATE]=   DM7520_Create;
8    pDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] =
         DM7520_DeviceControl;
9    return STATUS_SUCCESS;
10 }
```

Ahogy az a kódból is látható, a *PDRIVER_OBJECT* struktúra függvénymutatókon keresztül éri el az illesztőprogramunk funkcióit. A

DriverUnload a driver rendszerből való eltávolításakor kerül meghívásra, ennek felelőssége minden az egész drivernek (nem csak egy-egy konkrét funkciójának) lefoglalt erőforrás felszabadítása.

Az *AddDevice* hívás felel azért, hogy amikor egy új, a mi driverünk által kezelt eszközt csatlakoztatnak a számítógéphez, az eszköz megfelelően fel legyen telepítve.

A *MajorFunction* tömb felel a kártya főbb funkcióinak implementációjáért, ezekről később lesz szó.

A *STATUS_SUCCESS*, ami 0-ként van definiálva jelzi az általános sikert. Amikor a *DriverEntry* visszatér, a szolgáltatásunk természetesen nem áll le, hanem pont hogy ilyenkor áll készen a különböző hívások fogadására.

A következő lépés felkészíteni a drivert arra, hogy az eszközt behelyezik a rendszerbe, ezért felel az *AddDevice* függvény, amely a mi esetünkben a következőképpen néz ki:

```
1  NTSTATUS DM7520_AddDevice(PDRIVER_OBJECT pDriverObject,
       PDEVICE_OBJECT pdo) {
2    PDEVICE_OBJECT fdo;
3    NTSTATUS status = IoCreateDevice(pDriverObject, sizeof(
         DEVICE_EXTENSION), NULL, FILE_DEVICE_UNKNOWN,
         FILE_DEVICE_SECURE_OPEN, FALSE, &fdo);
4    DEVICE_EXTENSION* pdx = (DEVICE_EXTENSION*) fdo->DeviceExtension;
5    if(!NT_SUCCESS(status)) {
6      return status;
7    }
8    pdx->lowerDeviceObject = IoAttachDeviceToDeviceStack(fdo, pdo);
9    if(!pdx->lowerDeviceObject)
10   {
11     IoDeleteDevice(fdo);
12     return status;
13   }
14   fdo->Flags |= DO_POWER_PAGABLE;
15   fdo->Flags |= DO_BUFFERED_IO;
16   fdo->Flags &= ~DO_DEVICE_INITIALIZING;
17   status = IoRegisterDeviceInterface(pdo, &DM7520_GUID, NULL, &pdx
         ->ifname);
18   if(!NT_SUCCESS(status))
19   {
20     if(!pdx->ifname.Buffer)
21     {
22       RtlFreeUnicodeString(&pdx->ifname);
23     }
24     IoDetachDevice(pdx->lowerDeviceObject);
25     IoDeleteDevice(fdo);
26     return status;
27   }
28   status = IoSetDeviceInterfaceState(&pdx->ifname, TRUE);
29   if(!NT_SUCCESS(status))
30   {
31     if(!pdx->ifname.Buffer)
32     {
33       RtlFreeUnicodeString(&pdx->ifname);
34     }
35     IoDetachDevice(pdx->lowerDeviceObject);
36     IoDeleteDevice(fdo);
37     return status;
38   }
39   return STATUS_SUCCESS;
40 }
```

Ez a kicsit testesebb függvény egy nagyon kritikus pontja az illesztőprogramnak. A két kapott paraméter közül az első a már ismert *PDRIVER_OBJECT*, a második pedig egy *PDEVICE_OBJECT* struktúrára egy mutató. Ezen mutató mögött már az éppen behelyezett és konfigurálni kívánt eszköz található.

Érdemes még a konkrét implementáció előtt néhány szóban megemlíteni az ún. driver stacket, ami tulajdonképpen a driverek absztrakciós rétegei. Legalul helyezkednek el az ún. PDO-k, Physical Device Object, azaz maga a konkrét fizikai eszköz. Ez a réteg felel az eszközzel való legalapvetőbb kommunikációjáért, mint például az adatok küldésének és fogadásának implementációja, vagy akár az áramellátás szabályozása. Ezzel a réteggel nekünk nem sok dolgunk van, mivel a mi eszközünk egy teljesen szabványos PCI eszköz, a Windows alapértelmezett módszerei tökéletesen megfelelnek. Erre a szintre épül a Filter Device Object, ami tulajdonképpen egy "szűrő", ez a réteg nem kötelező egy illesztőprogramhoz; itt különböző cache-eléseket lehet például megvalósítani. Az erre épülő réteg az FDO, Function Device Object, ami tulajdonképpen a driver logikáját tartalmazza, itt van megvalósítva a kernel-módú API. Szükség szerint az FDO-ra épülhet még egy szűrőréteg, azonban nálunk erre nincs szükség.

A kódban megjelenik két rövidítés két változónév formájában. Az *fdo*-ra keresztelt *PDEVICE_OBJECT* a funkcionális illesztőprogramunkat reprezentálja, ezért teljes mértékben mi vagyunk a felelősek. A *pdo* nevű, szintén *PDEVICE_OBJECT* típusú változónk pedig a fizikai eszközillesztőnkre mutat, ezt javarészt a Windows maga menedzseli.

A függvénybe való belépéskor a PDO már létezik és készen áll arra, hogy ráépítsünk a driver stack többi részét. Első lépésként létrehozzuk az alap FDO-t *(4)*. Ehhez csak pár nagyon alapvető paraméterre van szükségünk, mint az eszköz típusa, az általunk igényelt extra tárterület, az ún. device extension mérete, valamint minimális biztonsággal kapcsolatos paraméterek.

A *DEVICE_EXTENSION* egy többé-kevésbé megszokott elnevezése annak az általunk definiált struktúrának, ami egy az eszközhöz csatolt adatokat tartalmazza, ennek a tartalma minden eseteben a konkrét eszközön múlik. A mi driverünkben ez a struktúra a következőképpen van definiálva:

```
1 struct DEVICE_EXTENSION
2 {
3   PDEVICE_OBJECT lowerDeviceObject;
4   UNICODE_STRING ifname;
5   volatile UCHAR* las0;
6   volatile UCHAR* las1;
7   volatile UCHAR* lcfg;
8 };
```

A *lowerDeviceObject* az FDO alatt elhelyezkedő rétegre mutat, ennek segítségével fogjuk tudni azt elérni. Az *ifname* (interface name) tartalma az eszközünk interfészének neve, amin keresztül majd a felhasználói módú programok elérik az eszközt. A másik három mező már a kártyánk konkrét felépítését tükrözi, ez a három mutató a kártya három memóriaterületére mutat, ezeken keresztül fogunk majd tudni I/O műveleteket végezni.

Az IoAttachDeviceToDeviceStack hívás *(10)* segítségével tudjuk az FDO-t

csatlakoztatni a driver stackre, majd a visszakapott *PDEVICE_OBJECT*-et, ami a közvetlenül alattunk elhelyezkedő rétegre mutat, elmentjük a *DEVICE_EXTENSION*-ünkbe.

Miután létrehoztuk és csatlakoztattuk az FDO-t, természetesen konfigurálni is kell azt. A legfontosabb lépés a különféle flagek beállítása *(16 - 18)*. Három olyan csoportja van ezeknek a flageknek, amelyekre mindenféleképpen oda kell figyelni. Az első az áramellátás. A *DO_POWER_PAGABLE* flag az ún. lapozható eszközillesztők jele. Ez annyit jelent, hogy szükség esetén a Windows kilapozhatja a driverünket a memóriából, ami csak a számítógép működéséhez kritikus eszközök esetében jelentene problémát. Senki nem szeretne arra várni, hogy a videokártya drivere visszakerüljön a memóriába, ám egy mérés-adatgyűjtő kártya megnyitásakor ez nem jelent gondot.

A második beállítandó flag az input/output módját szabályozza. Egy eszköznek három különféle lehetősége van az I/O műveleteinek lebonyolítására. Az első a *DO_BUFFERED_IO*, ami a pufferen keresztülő be- és kimenetet jelenti. Ebben az esetben az eszköz és a driver közé a Windows automatikusan létrehoz egy puffert, amit ő maga kezel számunkra. A második lehetőség a a *DO_DIRECT_IO*, ami a DMA használatával "közvetlen" utat biztosít a kártya és a driver közé az adatok számára. A harmadik beállítás, ami egyben az alapértelmezett is, az úgynevezett "Neither" (egyik sem) megoldás. Ebben az esetben a driver csak egy forrás és egy cél címet lát és neki kell megoldania a másolást. Azonban ez nagyon körülményes, lévén hogy a két cím teljesen más kontextusban van (jó eséllyel a cél egy kernel-módú cím a mi driverünkben, a forrás azonban szinte biztosan nem, hiszen mi egy FDO-ban vagyunk), pontosan emiatt ezt a megoldást nagyon ritkán használják FDO-kban. A választás azért esett a pufferelt megoldásra, mert sokkal kisebb az overhead az adatok mozgatásakor, mint a DMA-s megoldás esetén, az adataink alacsony mérete (1-1 bájtot küldünk egyszerre), miatt pedig a DMA amúgy sem nyújtana semmi előnyt.

Az utolsó flag, ami nélkülözhetetlen, az a *DO_DEVICE_INITIALIZING*. Ez a flag alapból be van állítva, és amíg ezt ki nem töröljük, az eszközt nem lehet elindítani, ugyanis ez a flag jelzi, hogy az eszköz még konfiguráció alatt áll. A pontosság kedvéért érdemes megjegyezni, hogy az, hogy ezt a flaget az *AddDevice* függvény közepén töröljük, nem jelenti azt, hogy az eszközt azonnal el lehetne indítani, mielőtt a függvény többi része lefutna; az eszköz elindítása csak az *AddDevice* függvény lefutása után lehetséges.

Következő lépésként létrehozunk az eszközünknek egy interfészt *(19)*, illetve ahhoz egy nevet. Ezen a néven keresztül lehet elérni az interfészt, amin keresztül pedig a kártyával való interakció lehetséges. Ezt az interfészt hozzá kell rendelnünk egy konkrét PDO-hoz, valamint szükség van egy GUID-ra is, ami a mi eszközünket azonosítja. A *DM7520_GUID* nálunk egy egyszerű konstans GUID, amit a Visual Studio beépített GUID generátorával készítettünk. Nem elég azonban regisztrálni egy interfészt, külön lépésként engedélyezni is kell azt *(30)*.

Amennyiben minden jól ment, nincs más dolgunk mint jelezni az eszköz hozzáadásának sikerét a *STATUS_SUCCESS* értékkel való visszatéréssel *(41)*.

A való életben természetesen messze nem megy minden problémák nélkül, így tekintsük át röviden a WDK által biztosított hibakezelési lehetőségeket.

A driverfejlesztéshez használt WDK függvények nagy többsége egy, már említett, *NTSTATUS*-t ad vissza, ezzel jelezve a művelet eredményét. A WDK-ban használt konvenció szerint a negatív értékek hibát, a nulla és a pozitív értékek pedig sikert jeleznek. Azonban a konkrét értékek jelentésének ismerete nélkül, az *NT_SUCCESS* makró segítségével egy boolként tekinthetünk az adott státuszra.

Minden egyes rendszerhívás után célszerű ellenőrizni, annak sikerességét. Hiba esetén minden félkész konfigurációt vissza kell vonni.

Ezek után biztosítanunk kell a kártyának a legalapvetőbb műveleteket, ilyen például a kártya megnyitása és bezárása. Ehhez azonban tudni kell, hogy hogyan működnek a kernelben az üzenetek. A különböző kéréseket üzenetek formájában kapják meg a driverek, ezeknek a pontos neve *IRP* (Interrupt Request Packet), ezek tartalmazzák a kérést, illetve a kéréssel kapcsolatos alapvető információkat.

Amikor egy driver egy adott kérést fogad, el kell döntenie mit kíván vele tenni. Egy egyszerűbb illesztőprogramban javarészt kétféle megoldásra van szükségünk. Az első a kérésnek a teljesítése, ebben az esetben a mi driverünk egy-az-egyben teljesíti a kérést: végrehajtja a kért műveletet, majd egy státuszkódot mellékelve jelzi, hogy az adott kérés teljesítve lett. A második gyakran használt lehetőség az úgynevezett "Forward & Forget" nevű módszer. Ebben az esetben mi nem kezeljük az adott kérést csupán továbbítjuk másnak. Utóbbira tipikus példák a különböző áramellátással kapcsolatos kérések, ahol egy FDO-nak rendszerint semmi dolga nincs (esetleg a kártya alaphelyzetbe állítása), azonban az alsóbb, fizikai eszközökkel foglalkozó rétegnek igen jelentős munkájuk van. A kérések teljesítése pedig a különböző I/O műveleteknél jön elő leggyakrabban, hiszen a Windows nem tudhatja, hogy a memóriából pontosan melyik címre is kéne kerülnie az adatoknak, ennek a megállapításáért már az eszközillesztő a felelős.

Anélkül, hogy minden függvényt részleteznénk, a két eljárást használó egy-egy függvényt bemutatunk:

```
1 NTSTATUS DM7520_PnP_StopDevice(PDEVICE_OBJECT fdo, PIRP pirp)
2 {
3   pirp->IoStatus.Status = STATUS_SUCCESS;
4   IoSkipCurrentIrpStackLocation(pirp); //Forward and Forget
5   DEVICE_EXTENSION* pdx = (DEVICE_EXTENSION*) fdo->DeviceExtension;
6   return IoCallDriver(pdx->lowerDeviceObject, pirp);
7 }
```

Az itt látható függvény a kártyánk megállításáért felel, ami például az eszköz szoftveres eltávolításakor jöhet elő. Kernel-módú üzenetkezelő függvényekre jól jellemző módon egy *NTSTATUS*-szal fogjuk jelezni a sikert. Paraméterként megkapjuk szokás szerint az FDO-t, valamint a már említett IRP-re egy mutatót.

A státuszt a már megszokott módon alapból jóhiszeműen *STATUS_SUCCESS*-re állítjuk. Ezután meghívjuk az *IoSkipCurrentIrpStackLocation* függvényt, amely annyit csinál, hogy az IRP-nket eggyel lejjebb küldi a driver stacken. Azonban az alattunk lévő drivert még meg is kell hívni, amihez először is meg kell azt találni *(5)*, majd az *IoCallDriver* hívással delegáljuk az - immáron jó helyen lévő - kérésünket az alattunk elhelyezkedő driver-réteg felé.

A következő kódrészlet pedig egy kérés teljesítését mutatja be:

```
1 NTSTATUS DM7520_Create(PDEVICE_OBJECT fdo, PIRP pirp)
2 {
3   IoCompleteRequest(pirp, IO_NO_INCREMENT);
4   pirp->IoStatus.Status = STATUS_SUCCESS;
5   return STATUS_SUCCESS;
6 }
```

Itt a kulcs lépés az *IoCompleteRequest* hívás, amellyel jelezhetjük, hogy az adott kérést mi teljes mértékben teljesítettük és az IRP-vel további munka nincs. Ennek meghívása előtt van természetesen maga a kérést teljesítő logika, amely jelen esetünkben üres, ugyanis itt a kártyát megnyitó kérést kezeljük, amelynek során azt kell biztosítanunk, hogy a hívó fél gond nélkül tudjon írni és olvasni akaró kéréseket küldeni. Mivel az eszközillesztőt saját célokra fejlesztettük, eltekintettünk a különféle biztonsági óvintézkedésektől amelyeket itt lehetne tenni (pl. blokkolni hogy egyszerre többen is írási engedéllyel rendelkezzenek). A függvény második paraméterével a kérést eredetileg feladó szálnak a prioritását lehetne növelni (cserébe azért, hogy eddig a kérést teljesítésére kellett várnia), azonban mi ezzel nem kívántunk élni. Mivel lényegében semmit nem csináltunk, ezért a státuszt gond nélkül állíthatjuk *STATUS_SUCCESS*-re, majd térhetünk vissza a függvényből.

Érdemes még külön szót ejteni a Plug and Play üzenetekről. A különféle Plug and Play üzeneteket (mint pl. a kártya behelyezése vagy eltávolítása) egyszerre, egy függvényben tudjuk kezelni (ld. a *DriverEntry* függvény).

```
1 NTSTATUS DM7520_PnP(PDEVICE_OBJECT fdo, PIRP pirp)
2 {
3   PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(pirp);
4   switch(stack->MinorFunction)
5   {
6     case IRP_MN_START_DEVICE:  return DM7520_PnP_StartDevice (fdo,
          pirp);
7     case IRP_MN_REMOVE_DEVICE: return DM7520_PnP_RemoveDevice(fdo,
          pirp);
8   }
9   //Default - "Forward and Forget"
10  IoSkipCurrentIrpStackLocation(pirp);
11  DEVICE_EXTENSION* pdx = (DEVICE_EXTENSION*) fdo->DeviceExtension;
12  return IoCallDriver(pdx->lowerDeviceObject, pirp);
13 }
```

Ez a függvény fogadja az összes Plug and Play üzenetet, majd szükség szerint delegálja az üzeneteket további függvényeknek. Az eddigi kezelő függvényeinkben mindig pontosan tudtuk, hogy milyen üzenettel van dolgunk, ezért sosem volt szükség az IRP tartalmát vizsgálni, azonban itt már csak annyit tudunk, hogy valamilyen Plug and Play esemény történt. Az IRP tartalmazza a pontos eseményt is, ehhez azonban hozzá kell férnünk az IRP-hez, amit az *IoGetCurrentIrpStackLocation* hívással tudunk megtenni. Ezek után már meg tudjuk vizsgálni az ún. Minor code-ját a kérésnek, amelyből már kiderül, hogy a PnP-en belül pontosan mi is történt.

A mi driverünkben a PnP események közül csak kettő érdekel, amikor elindítják a kártyát, illetve amikor eltávolítják azt a rendszerből, így ezeket egy-egy külön függvény kezeli. Minden egyéb PnP kérésre a már korábban bemutatott Forward & Forget megoldást alkalmazzuk.

Az eszköz elindításakor a *DM7520_PnP_StartDevice* függvényt hívjuk meg, ez felel a kártya felkészítéséért, hogy tudja fogadni a különböző szoftver-eredetű kéréseket (ilyen például az írás és az olvasás). A függvény a következőképpen néz ki:

```
1  NTSTATUS DM7520_PnP_StartDevice(PDEVICE_OBJECT fdo, PIRP pirp)
2  {
3    DEVICE_EXTENSION* pdx = (DEVICE_EXTENSION*) fdo->DeviceExtension;
4    IoSetDeviceInterfaceState(&pdx->ifname, TRUE);
5
6    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(pirp);
7    PCM_PARTIAL_RESOURCE_LIST translated = &stack->Parameters.
         StartDevice.AllocatedResourcesTranslated->List[0].
         PartialResourceList;
8
9    PCM_PARTIAL_RESOURCE_DESCRIPTOR resources = translated->
         PartialDescriptors;
10   ULONG count = translated->Count;
11   for(ULONG i = 0; i < count; ++i)
12   {
13     if(resources[i].Type == CmResourceTypeMemory)
14     {
15       switch(resources[i].u.Memory.Length)
16       {
17         case 512: // LAS0
18           pdx->las0 = (volatile UCHAR*) MmMapIoSpace(resources[i].u
                 .Memory.Start, 512, MmNonCached);
19           break;
20         case 256: // LCFG
21           pdx->lcfg = (volatile UCHAR*) MmMapIoSpace(resources[i].u
                 .Memory.Start, 256, MmNonCached);
22           break;
23         case 16:  // LAS1
24           pdx->las1 = (volatile UCHAR*) MmMapIoSpace(resources[i].u
                 .Memory.Start,  16, MmNonCached);
25           break;
26       }
27     }
28   }
29
30   pirp->IoStatus.Status = STATUS_SUCCESS;
31   IoCompleteRequest(pirp, IO_NO_INCREMENT);
32   return STATUS_SUCCESS;
33 }
```

Első lépésként aktiváljuk az *AddDevice* függvényben már regisztrált interfészünket *(4)*. Ahogyan ott, itt sem kell attól tartanunk, hogy az engedélyezés és a *StartDevice* függvény vége között valaki megpróbálja használni a félkész kártyánkat, ugyanis az eszköz csak a függvény visszatérése után számít működésre késznek. Ezek után a már látott módon hozzáférünk az IRP mögötti tartalomhoz *(6)*, amelyből megszerezzük az eszközünk erőforrásainak listáit *(7)*. Ezek után a felismert erőforrásokon végigiterálva bemappeljük a kártya memóriaterületeit a számítógép memóriájába. A mi kártyánkon három, különböző méretű memóriaterület található (LAS0, LAS1 és LCFG), így szerencsére a méretük ismeretében már egyértelműen tudjuk azonosítani őket. Az *MmMapIoSpace* függvény felel a mappelésért, amelynek egyszerűen át kell adni a mappelni kívánt memóriaterületre egy mutatót, annak hosszát, illetve hogy kívánunk-e a gyorsítótárazással élni. Mivel mi nanoszekundumos sebességgel kívánunk jeleket kiadni a kártyán, így nem

élünk ezzel a lehetőséggel. A függvény által visszaadott *void**-t, amely a számítógép memóriájára mutat, elmentjük a *DEVICE_EXTENSION*-ünk megfelelő részeibe, a bájtonkénti elérés és a gyorsítótárazás elkerülése érdekében *volatile UCHAR**-re átalakítva. Végezetül jelezzük a sikerünket és teljesítjük a kérést.

Az előbb bemutatásra került, hogy miként foglaljuk le az erőforrásokat a kártya használatához, azonban az is nagyon fontos, hogy ezeket a megfelelő időben visszaadjuk a rendszernek. Ezért felel az alábbi függvény:

```
1  NTSTATUS DM7520_PnP_RemoveDevice(PDEVICE_OBJECT fdo, PIRP pirp)
2  {
3    DEVICE_EXTENSION* pdx = (DEVICE_EXTENSION*) fdo->DeviceExtension;
4    IoSetDeviceInterfaceState(&pdx->ifname, FALSE);
5    RtlFreeUnicodeString(&pdx->ifname);
6    MmUnmapIoSpace((PVOID)pdx->las0, 512);
7    MmUnmapIoSpace((PVOID)pdx->las1,  16);
8    MmUnmapIoSpace((PVOID)pdx->lcfg, 256);
9
10   pirp->IoStatus.Status = STATUS_SUCCESS;
11   NTSTATUS status = IoCallDriver(pdx->lowerDeviceObject, pirp);
12   IoDetachDevice(pdx->lowerDeviceObject);
13   IoDeleteDevice(fdo);
14   return status;
15 }
```

Ez a függvény kerül meghívásra, amikor az eszközt eltávolítják a rendszerből, legyen az akár fizikai eltávolítás vagy szoftveres. Első lépésként megszüntetjük a kártya interfészének elérhetőségét, felszabadítjuk az interfész nevének fenntartott memóriát, valamint egyesével megszüntetjük a mappeléseket mindhárom memóriaterületre. Ezek után nem egyszerűen teljesítjük a kérést, hanem tovább küldjük az alsóbb rétegeknek, hiszen ha a kártyát fizikailag eltávolítják a rendszerből, akkor arról a fizikai drivernek is tudnia kell. A Forward & Forget módszerrel ellentétben azonban most nem felejtjük el a kérést, hanem megvárjuk míg az alattunk lévő rétegek elvégzik teendőiket, majd ezek után leválasztjuk a driverünket a driver stackről az *IoDetachDevice* hívással, valamint megsemmisítjük a funkcionális driverünket.

Ezek a függvények biztosítják a kártyánk alapvető működéséhez az "adminisztrációt". Bár az eszközünk még egyáltalán nem funkcionális, de már gond nélkül be-ki lehet kapcsolni, illetve hozzá lehet adni és el lehet venni a rendszerből. Magáért a funkcionális működésért a következő függvény felel:

```
1  NTSTATUS DM7520_DeviceControl(PDEVICE_OBJECT fdo, PIRP pirp)
2  {
3    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(pirp);
4    DEVICE_EXTENSION* pdx = (DEVICE_EXTENSION*) fdo->DeviceExtension;
5    NTSTATUS status = STATUS_INVALID_DEVICE_REQUEST;
6    pirp->IoStatus.Information = 0;
7
8    switch(stack->Parameters.DeviceIoControl.IoControlCode)
9    {
10     case DM7520_WRITE_BUFFERED:
11     {
12       // ...
13       break;
14     }
15     case DM7520_READ_BUFFERED:
16     {
17       // ...
18       break;
19     }
20   }
21   IoCompleteRequest(pirp, IO_NO_INCREMENT);
22   pirp->IoStatus.Status = status;
23   return status;
24 }
```

A driver API-jának bemutatásánál látni fogjuk, hogy az eszközzel való kommunikációhoz minden esetben a Windows API által biztosított *DeviceIoControl* függvényt fogjuk használni. Ezek a függvényhívások jutnak el kérések formájában a fent látható függvényünkhöz. Általánosságban ez a függvény felel azért, hogy a kártyára specifikus funkcionális kéréseket fogadja és végrehajtsa. A mi esetünkben csupán két funkciót definiáltunk a kártyához, az írást és az olvasást. Kezdésként megszerezzük az IRP-hez tartozó információkat és beállítjuk a visszaadandó értékek alapértelmezéseit. Az *information* jelentése mindig az adott kéréstől függ, így a legegyszerűbb ezt kezdetben lenullázni. A *status* kezdeti értékének a *STATUS_INVALID_DEVICE_REQUEST* értéket választottuk, ezzel garantálva a megfelelő hibajelzést amennyiben valaki egy érvénytelen kérést küld.

Az *IoControlCode* tartalmazza a pontos kérést (mint a Plug and Play esetében a *MinorCode*), amelynek lehetséges értékeit mi definiáltuk a következő módon:

```
1  // IOCTL codes
2  #define DM7520_WRITE_BUFFERED \
3    CTL_CODE(FILE_DEVICE_UNKNOWN, 0x800, METHOD_BUFFERED,
         FILE_WRITE_ACCESS)
4  #define DM7520_READ_BUFFERED \
5    CTL_CODE(FILE_DEVICE_UNKNOWN, 0x801, METHOD_BUFFERED,
         FILE_READ_ACCESS)
```

A *CTL_CODE* egy speciális makró, amely segítségével létre lehet hozni a négy részből álló IOCTL (I/O Control) kódokat. A kód első tartalmazza, hogy a kód milyen eszközre vonatkozik (például grafikus kártya, merevlemez). A mérés-adatgyűjtő kártyákhoz a Windows nem biztosít semmilyen különleges bánásmódot, ezért az "egyéb" kategóriát jelölő *FILE_DEVICE_UNKNOWN* értéket választottuk. A kód második része egy, általunk megadott, azonosítója

a kérésnek. Harmadikként kell megadni, hogy ehhez a kéréshez milyen típusú I/O-t kívánunk használni, ami a mi esetünkben mindig pufferelt. Végül pedig a kéréshez szükséges jogokat kell megadni, azaz ahhoz hogy ezt a kérést valaki elküldhesse, milyen hozzáférési joggal kell rendelkeznie a kártyához.

Ezek után végrehajtjuk az írást / olvasást, majd végül teljesítjük az IRP-t. Amennyiben a program a switch egyik ágába sem lép be, úgy a státusz marad *STATUS_INVALID_DEVICE_REQUEST*, ellenkező esetben pedig az adott ág felelős a státusz beállításáért.

Az I/O műveletekhez egy I/O műveletet leíró struktúrát használunk, amelynek definíciója a következő:

```
1 struct DM7520_IO_DESCRIPTOR
2 {
3   DM7520::MEMORY_LOCATION memloc;
4   DM7520::SIZE size;
5   UINT16 offset;
6   UINT32 data;
7 };
```

Az első tag a *memloc*, amely azt tartalmazza, hogy a művelet a kártyának melyik memóriatartományára vonatkoznak. A *size* mező mutatja meg, hogy hány bájtnyi adatot szeretnénk írni / olvasni. Ez a két tag egy-egy enumerációval adható meg, ezek a következőképpen néznek ki:

```
1 enum MEMORY_LOCATION : char
2   { LAS0 = 0x0, LAS1 = 0x1, LCFG = 0x2 };
3 enum SIZE : char
4   { BYTE = 0x1, WORD = 0x02, DWORD = 0x04 };
```

Az *offset* a művelet memóriarekeszen belüli pontos helyét mondja meg, a *data* pedig az adat amit írni kívánunk, illetve amit beolvastunk.

Az írási ág a következőképpen néz ki:

```
1  switch(stack->Parameters.DeviceIoControl.IoControlCode) {
2    case DM7520_WRITE_BUFFERED:
3    {
4      DM7520_IO_DESCRIPTOR* ioDesc = (DM7520_IO_DESCRIPTOR*) pirp->
           AssociatedIrp.SystemBuffer;
5      status = STATUS_SUCCESS;
6      pirp->IoStatus.Information = ioDesc->size;
7      volatile UCHAR* address;
8      switch(ioDesc->memloc) {
9        case DM7520::LAS0: address = pdx->las0; break;
10       case DM7520::LAS1: address = pdx->las1; break;
11       case DM7520::LCFG: address = pdx->lcfg; break;
12     }
13     address += ioDesc->offset;
14     switch(ioDesc->size)
15     {
16       case DM7520::BYTE:  WRITE_REGISTER_UCHAR ((UCHAR*) address, (
             UINT8) ioDesc->data); break;
17       case DM7520::WORD:  WRITE_REGISTER_USHORT((USHORT*)address, (
             UINT16)ioDesc->data); break;
18       case DM7520::DWORD: WRITE_REGISTER_ULONG ((ULONG*) address, (
             UINT32)ioDesc->data); break;
19     }
20     break;
21   }
22 }
```

Pufferelt I/O esetén a Windows szolgáltat nekünk egy puffert, ahol elérjük a felhasználói módból átküldött adat másolatát, ez található a *SystemBuffer* által meghatározott helyen, ahol nekünk normális esetben egy I/O műveletet leíró struktúra van. Írás esetén az *Information* az eszközre írt bájtok számát tartalmazza. Ezek után az *address* nevű változónkban beállítjuk a kezdőcímet az átadott struktúra függvényében, majd hozzáadjuk az offsetet. Ezek után a méret függvényében a Windows Driver Kit által biztosított makrókon keresztül végrehajtjuk az írást az eszközre.

Az olvasás sokban hasonlít az írásra, azonban van pár fontos különbség közöttük. A megvalósítás a következő:

```
1  //...
2  case DM7520_READ_BUFFERED:
3  {
4    DM7520_IO_DESCRIPTOR* ioDesc = (DM7520_IO_DESCRIPTOR*) pirp->
         AssociatedIrp.SystemBuffer;
5    status = STATUS_SUCCESS;
6    pirp->IoStatus.Information = sizeof(DM7520_IO_DESCRIPTOR);
7    volatile UCHAR* address;
8    switch(ioDesc->memloc)
9    {
10     case DM7520::LAS0: address = pdx->las0; break;
11     case DM7520::LAS1: address = pdx->las1; break;
12     case DM7520::LCFG: address = pdx->lcfg; break;
13   }
14   address += ioDesc->offset;
15   switch(ioDesc->size)
16   {
17     case DM7520::BYTE:  ioDesc->data = READ_REGISTER_UCHAR ((UCHAR
           *) address); break;
18     case DM7520::WORD:  ioDesc->data = READ_REGISTER_USHORT((USHORT
           *)address); break;
19     case DM7520::DWORD: ioDesc->data = READ_REGISTER_ULONG ((ULONG
           *) address); break;
20   }
21   break;
22 }
```

Az íráshoz hasonlóan itt is szükségünk van inputra egy I/O leíró struktúra formájában, amelyet a már említett *SystemBuffer* változón keresztül érünk el. Az *Information* értékének itt az olvasott bájtok számát állítjuk be, azonban fontos odafigyelni, hogy a "beolvasott bájtok száma" kifejezést felhasználói módból értelmezzük! A driver szempontjából csak a *size*-nak megfelelő mennyiségű bájtot olvasunk be, azonban a felhasználónak egy egész *DM7520_IO_DESCRIPTOR* struktúrát visszajuttatunk, ezért annak az értékét adjuk vissza információ gyanánt. Ezek után pontosan ugyanaz történik mint az írásnál, azzal a különbséggel természetesen, hogy most a *READ_REGISTER* kezdetű makrókat használjuk, majd az értéket eltároljuk a struktúránkban.

Az I/O műveleteknél nem kell amiatt aggódnunk, hogy van-e hozzáférésünk a megadott pufferekhez, vagy hogy azok módosulhatnak-e időközben. Mivel pufferelt I/O-ról beszélünk, ezért a Windows a *DeviceIoControl* hívásoknál az átadott paramétereket lemásolja nekünk és egy kernel-módú memóriaterületen lévő pufferbe teszi őket, amit mi szabadon módosíthatunk. Olvasás esetében a *SystemBuffer* első *Information* bájtja automatikusan visszamásolódik a Windows által a felhasználói módú pufferbe,

így nekünk ezzel nem kell foglalkoznunk, attól eltekintve, hogy rögzítve van, hogy mely memóriacímre kell írnunk a visszaadandó értékeket.

Érdemes még a hibamegelőzés céljából megemlíteni, hogy a WDK fordítóprogramjai automatikusan végrehajtanak egy statikus kódanalízist a driverünkön, amelynek munkáját nagyban megkönnyíti, ha a különféle függvényeinkről előre megmondjuk, hogy milyen célt kívánnak betölteni. Ezt a következőképpen lehet megtenni:

```
1 extern "C" DRIVER_INITIALIZE DriverEntry;
2 DRIVER_UNLOAD DM7520_DriverUnload;
3 DRIVER_ADD_DEVICE DM7520_AddDevice;
4 DRIVER_DISPATCH DM7520_Close;
5 DRIVER_DISPATCH DM7520_Create;
6 DRIVER_DISPATCH DM7520_DeviceControl;
7 DRIVER_DISPATCH DM7520_PnP;
```

Azáltal, hogy a kódanalizátor látja, hogy a *DM7520_AddDevice* függvény elvileg az eszköz hozzáadását kéne hogy kezelje, meg tudja vizsgálni, hogy megfelel-e pár minimumkövetelménynek, amelyeket az erre a célra írt függvényeknek teljesíteniük kell.

Ezzel lényegében el is készítettünk az illesztőprogramunk forráskódját, amit a WDK által biztosított speciális fordítókkal le tudunk fordítani egy .sys fájllá. Azonban egy .sys fájllal önmagában a rendszer még nem tud mit kezdeni, szükségünk van egy második fájlra, amely leírja a driverünket. Ezt egy .inf fájlban tudjuk megtenni. A következőekben ennek a fájlnak a rövid bemutatása következik.

2.2.2 A driver-leíró .inf fájlok

Egy driver telepítése a Windows rendszerekben mindig egy .inf fájlon keresztül indul. Ezek a fájlok határozzák meg, hogy melyik eszközhöz pontosan milyen szolgáltatások és driverek tartoznak, milyen lépéseket kell az operációs rendszernek végrehajtania az eszközök telepítéséhez. Tekintsük át a mi konkrét .inf fájlunkat, hogy pontosan milyen információkat is kell megadni:

```
1 [Version]
2 Signature = "$Windows NT$"
3 Class     = RTDDataModule
4 ClassGuid = {D695ED6A-630D-4D83-8D8B-F1F0AC107AD0}
5 Provider  = %DM7520_InfProvider%
6 DriverVer = 04/04/2012,1.1.2.2
```

A legtöbb .inf fájl a kötelező *Version* részleggel indul, ebben vannak a legalapvetőbb információk, amelyek még nem is a konkrét driverről vagy eszközről, hanem egy termékcsaládról szól, ugyanis egy .inf fájl több eszközt és több drivert is leírhat. A *Signature* azonosítja a rendszer típusát, amelyre mi telepíteni kívánjuk az eszközünket. Ennek két értéke lehet: *$Windows NT$* és *$Chicago$*. Mindkettő pontosan ugyanazt jelenti, méghozzá hogy Windows alapú rendszerről beszélünk. Jelenleg tehát lényegében jelentékeny ez az érték, azonban kihagyni nem lehet, különben érvénytelen lesz a fájlunk.

A *Class* bejegyzés azonosítja a mi termékosztályunkat (további példák: videokártyák, monitorok, perifériák). Mivel a mérés-adatgyűjtő kártyák nem

annyira elterjedtek, hogy saját kategóriájuk legyen alapból a Windows-ban, így létrehoztunk mi egyet. Az osztályokhoz minden esetben tartozik egy GUID, amelyet a *ClassGuid* bejegyzésben kell megadni; ezt az értéket szintén mi generáltuk. A *Provider*-nél az eszköz / illesztőprogram készítőjét szokás megadni. Azonban .inf fájlokban nem szokás közvetlenül a szövegeket megadni, helyette csak egy azonosítót, a szövegeket pedig egy külön erre a célra kialakított részlegben gyűjteni. Végezetül a driver verzióját, illetve dátumát szükséges megadni, ez alapján tudja a Windows, hogy melyik driver újabb a másiknál.

A következő részlegek a másolandó fájlok forrásait és céljait hivatottak beállítani:

```
1  [SourceDisksNames]
2  1 = %DM7520_DiskName%
3
4  [SourceDisksFiles.amd64]
5  DM7520.sys = 1,\x64
6
7  [SourceDisksFiles.x86]
8  DM7520.sys = 1,\x86
9
10 [DestinationDirs]
11 DefaultDestDir = 12 ; Windows\System32\Drivers
```

A *SourceDisksNames* részlegben kell megadni a telepítőlemezek elnevezéseit, a "Kérem helyezze be az XYZ telepítőlemezt..." típusú üzenetekhez. A mi esetünkben csak egyetlen egyet adtunk meg, aminek a nevét a string poolban adtuk meg. Ezek után jön a *SourceDisksFiles* részleg, amelyeket a *.amd64* és *.x86* kiegészítésekkel külön lehet választani 32 és 64-bites esetekre. Mindkét esetben a *DM7520.sys* fájl az egyetlen másolandó, ez maga a driver szolgáltatásunk. Mindkét esetben az első telepítőlemezen van a fájl, azon belül pedig az *x64* vagy *x86* mappában, az éppen futó rendszer paramétereitől függően. Ezután jön, hogy hová is kell a fájlokat másolni, ezt lehet a *DestinationDirs* részlegben megadni. A mi esetünkben elegendő egy alapértelmezett mappa, amelyre a Windows számos, számokkal jelölt lehetőséget biztosít számunkra. Mi a 12 jelű, Windows\System32\Drivers mappára mutatót választottuk, ahol a legtöbb driver elhelyezkedik.

Ezt követi a registry-vel kapcsolatos két bejegyzés:

```
1 [ClassInstall32]
2 AddReg = DM7520_Class_AddReg
3
4 [DM7520_Class_AddReg]
5 HKR,,,,%DM7520_ClassName%
```

A *ClassInstall32* részleg mondja meg, hogy az adott eszközosztály telepítésekor (tehát amikor a rendszer először találkozik egy olyan típusú eszközzel) miket kell csinálnia. Ebben az egyetlen kötelezően kitöltendő bejegyzés az *AddReg*, aminek egy másik részlegre kell mutatnia, amely leírja, hogy hol és milyen bejegyzéseket kell létrehozni a registry-ben. A mi esetünkben egyetlen egy bejegyzést kívánunk létrehozni azzal az egyetlen sorral a *DM7520_Class_AddReg* részlegben. Mivel nekünk semmit nem kell a registry-ben tárolni, a lehető legtöbb beállítást kihagytuk, így egy egyszerű

bejegyzést hoztunk létre *DM7520_ClassName* néven, amit a szövegeket tároló részben oldunk fel.

Ezután már a konkrét eszközöket azonosító részek következnek (a mi esetünkben csak egy eszköz van):

```
1 [Manufacturer]
2 %DM7520_Manufacturer% = DM7520,NTamd64,NTx86
3
4 [DM7520.ntamd64]
5 %DM7520_DeviceDescription% = DM7520_Install,PCI\VEN_1435&DEV_7520
6
7 [DM7520.ntx86]
8 %DM7520_DeviceDescription% = DM7520_Install,PCI\VEN_1435&DEV_7520
```

A *Manufacturer* részlegben lehet a gyártókat, az eszközöket és a platformokat összekapcsolni. A bejegyzések kulcsa mindig egy-egy gyártónak a neve (ami a string poolban mutat), értéke pedig egy adott eszközcsaládot leíró részleg, valamint utána vesszővel elválasztva a különböző támogatott platformok. Mi egy termékcsaládot és két platformot adtunk meg, amely két további (ám azonos) részleget eredményez. A részleg neve a megadott termékcsalád azonosítója és a platform azonosítója egy ponttal elválasztva. Ezekben a részlegekben lehet a termékcsaládon belül a termékeket megadni. Itt a kulcsok mindig az adott eszköznek a neve (string poolban), az értékek pedig az eszköznek a telepítéséről szóló részleg neve, valamint az eszköznek az egyedi azonosítója, ami tartalmazza a gyártó (vendor) azonosítóját, az eszközét, valamint az illesztés típusát is (PCI).

A telepítésről szóló részleg a következőképpen néz ki:

```
1 [DM7520_Install]
2 CopyFiles = @DM7520.sys
3
4 [DM7520_Install.Services]
5 AddService = DM7520_srvc,2,DM7520_ServiceInstall
```

Az első részlegben meg kell adni a fájlok neveit, amelyeket át kívánunk másolni. Ezek azonban nem magukra a fájlokra hivatkoznak, hanem a *SourceDisksFiles* részlegben megadott bejegyzésekre! A *DM7520_Install.Services* részleg a telepítésnek a szolgáltatásokról szóló része. Míg az előző részben csak fájlokat másoltunk, most új szolgáltatások létrehozására van lehetőség. Az *AddService* direktíva első paramétere a létrehozni kívánt szolgáltatás neve. Ezt követi a flag paraméter, amelynek a 2 értéke azt jelenti, hogy a megadott szolgáltatás egy Plug and Play function driver. Az utolsó paraméter pedig egy hivatkozás amely a szolgáltatás telepítésének konkrét paramétereit írja le. Ez a következő:

```
1 [DM7520_ServiceInstall]
2 DisplayName   = %DM7520_ServiceName%
3 ServiceType   = 1  ; SERVICE_KERNEL_DRIVER
4 StartType     = 3  ; SERVICE_DEMAND_START
5 ErrorControl  = 1  ; SERVICE_ERROR_NORMAL
6 ServiceBinary = %12%\DM7520.sys
```

Itt már a konkrét szolgáltatásunkat írjuk le, megadva először a szolgáltatás felhasználók számára megjelenő nevét (igény szerint egy rövid leírás is

adható). Ezek után meg kell adni a szolgáltatás típusát, ami a mi esetünkben egy kernel-módú driver. Szükség van továbbá arra is, hogy ez az adott szolgáltatás mikor induljon el. Itt olyan lehetőségek lennének még például, mint a "minden induláskor", a "minden induláskor késleltetve", vagy a "manuális indítás szükséges". Mi azt választottuk, hogy induljon el automatikusan, de csak akkor ha szükség van rá. Beállítandó továbbá, hogy a szolgáltatás hogyan kezelje a különböző fellépő hibákat. Egy merevlemez-illesztő esetében érthető lehet a hibáknak a szigorúbb kezelése, azonban mi megelégedtünk az alapbeállítással. Végezetül meg kell adni, hogy a rendszer hol találja magát a szoftverünket. Itt szintén előjön a már látott Windows\System32\Drivers mappát jelölő 12-es szám.

Az utolsó részleg pedig a string pool, ami a következőképpen néz ki:

```
1 [Strings]
2 DM7520_ClassName           = "RTD DataModule"
3 DM7520_DeviceDescription = "DM7520 Device"
4 DM7520_DiskName            = "DM7520 driver installation disc"
5 DM7520_InfProvider         = "RTD Embedded Technologies, Inc."
6 DM7520_Manufacturer        = "RTD Embedded Technologies, Inc."
7 DM7520_ServiceName         = "DM7520 device driver"
```

Itt egészen egyszerűen kulcsként azonosítókat adunk meg, amelyekre a fájl többi részén százalékjelek között tudunk hivatkozni, értékként pedig az azonosító helyére becserélni kíván szöveget lehet megadni. Ezzel a megoldással az összes szövegünket egy helyre tudjuk gyűjteni, ami sok szempontból is hasznos (pl. lokalizáció).

Az .inf és a szoftverünket tartalmazó .sys fájlok birtokában már behelyezhetjük az eszközt a rendszerbe, amelyet a Windows nem fog felismerni, így az eszközkezelőben manuális kell a driver helyét megadni. Ezek után az eszköz automatikusan feltelepül és használható.

2.2.3 Az eszközillesztők digitális aláírása

A 64-bites Windows 7 rendszerben alapból csak és kizárólag digitálisan aláírt eszközillesztőket lehet telepíteni a drivernek álcázott vírusok elleni védelem miatt. Természetesen a mi apró teszt driverünkhöz nem szereztünk be semmilyen tanúsítványt, azonban szerencsére a Windows ezen védelmi mechanizmusát sokféle módon meg lehet kerülni. Az egyik legegyszerűbb, ha az ember belép a boot menübe (F8-at lenyomva az operációs rendszer betöltődésének kezdetekor), majd ott a "Disable driver signature enforcement" opciót választja. Ennek hatására a Windows úgy indul el, hogy aláíratlan eszközkezelőket is lehet telepíteni. Ez a hatás a rendszer újraindításáig érvényes, utána deaktiválódik. Ebben az esetben a driverünk és az eszközünk telepítve marad, azonban nem lehet elindítani egy hibaüzenet miatt (A driver nincs aláírva).

2.3 A kártya API-jának bemutatása

Az eddigiek során megtudtuk hogyan tudjuk a saját driverünket elkészíteni. Azonban önmagában egy driver nehezen használható. Egy felső szintű driver

(egy funkcionális driver) rendszerint biztosít valamilyen API-t, annak érdekében, hogy a kártya működését belsőleg nem ismerő emberek is tudjanak rá programokat fejleszteni. Egy driver API elkészítése lényegében semmiben sem tér el egy bármilyen könyvtár megírásától. Lehetőségünk van akár magát a forráskódot is átadni (például .h és .cpp fájlokként), akár egy statikus könyvtárat (.h és .lib fájlokkal), vagy akár egy DLL-t is a céloktól függően. A mi esetünkben törekedtünk az egyszerűségre, így a felhasználói alkalmazásunk keretein belül fejlesztettük ki az API-t is. Ez ellent mond bizonyos programozási elveknek, azonban egy ekkora méretű projektnél kétség kívül sokkal egyszerűbb.

Először is meg kell fontoljuk, hogy mekkora hatalmat kívánunk a felhasználó (tehát a "külső" programozó) kezébe adni. Itt is fel lehet állítani a két extrém esetet:

```
1  // Case 1: Extreme trust
2  void Write(DM7520::MEMORY_LOCATION memloc, unsigned offset, void*
       data, unsigned dataLength);
3  void Read(DM7520::MEMORY_LOCATION memloc, unsigned offset, void*
       buffer, unsigned length);
4
5  // Case 2: Extreme distrust - return values always indicate success
6  bool ResetCard();
7  bool StartTimer();
8  bool ReadRegisterA();
9  bool WriteRegisterB(char data);
10 bool SetSomeBoolValue(bool value);
```

Az első esetben teljesen megbízunk a programozóban, minimalisztikus API-t alakítunk ki, ennek ellenére azonban mindent meg lehet vele csinálni. Természetesen rábízni a programozóra, hogy pontosan tudja, hogy melyik memóriacímre írhat és milyen értékeket lehet odaírni, ez húsz évvel ezelőtt volt elfogadható. A másik véglet pedig amikor minden egyes elvégezhető műveletre külön függvényt biztosítunk. A mai szoftverfejlesztés ebbe az irányba halad, azonban a mi munkánkhoz inkább az első megoldásra hasonlító API-t hoztunk létre:

```
1  DEFINE_GUID(DM7520_GUID,
2    0xf14af851, 0x9aeb, 0x41f3, 0xb3, 0xa9, 0x7d, 0x51, 0x19, 0x56, 0
         xce, 0x33);
3
4  // IOCTL codes
5  #define DM7520_WRITE_BUFFERED \
6    CTL_CODE(FILE_DEVICE_UNKNOWN, 0x800, METHOD_BUFFERED,
         FILE_WRITE_ACCESS)
7  #define DM7520_READ_BUFFERED \
8    CTL_CODE(FILE_DEVICE_UNKNOWN, 0x801, METHOD_BUFFERED,
         FILE_READ_ACCESS)
9
10 class DM7520
11 {
12 public:
13   enum MEMORY_LOCATION : char
14     { LAS0 = 0x0, LAS1 = 0x1, LCFG = 0x2 };
15   enum SIZE : char
16     { BYTE = 0x1, WORD = 0x02, DWORD = 0x04 };
17   ~DM7520();
18   static void CloseDevice(DM7520* device);
```

```
19    static DM7520* OpenDevice();
20    UINT32 Read(MEMORY_LOCATION memloc, UINT16 offset, SIZE size);
21    void Write(MEMORY_LOCATION memloc, UINT16 offset, SIZE size,
          UINT32 data);
22
23  private:
24    DM7520();
25    DM7520(const DM7520*);
26    DM7520(HANDLE handle);
27    HANDLE handle;
28  };
29
30  struct DM7520_IO_DESCRIPTOR
31  {
32    DM7520::MEMORY_LOCATION memloc;
33    DM7520::SIZE size;
34    UINT16 offset;
35    UINT32 data;
36  };
```

A GUID-ot, az IOCTL kódokat, valamint a *DM7520_IO_DESCRIPTOR* struktúrát csak a teljesség kedvéért szerepeltetjük, ezeket korábban már bemutattuk. A kártya alapvető manipulációjához a *DM7520* nevű osztályt hoztuk létre, amelynek publikus függvényeivel meg lehet nyitni és be lehet zárni a kártyát, valamint az írás és olvasás funkciókat is ezen az osztályon keresztül lehet elvégezni. Mivel szerettünk volna egy mindig konzisztens állapotból induló osztályt megvalósítani, a különböző konstruktorokat priváttá tettük, így az osztályt csak az *OpenDevice* függvényen keresztül lehet példányosítani. Erre azért volt szükség, mert a konstruktorból csak nagyon korlátozottan tudtuk volna a hibakezelést megoldani.

A kártyánkhoz minden esetben egy *HANDLE* típusú változón keresztül fogunk tudni hozzáférni, amelyet az *OpenDevice* függvényen keresztül tudunk biztosítani az osztályunknak:

```
1   DM7520* DM7520::OpenDevice()
2   {
3     HDEVINFO hdi = SetupDiGetClassDevs(&DM7520_GUID, 0, NULL,
          DIGCF_PRESENT | DIGCF_DEVICEINTERFACE);
4     if(hdi == INVALID_HANDLE_VALUE)
5     {
6       return nullptr;
7     }
8     SP_DEVICE_INTERFACE_DATA did;
9     did.cbSize = sizeof(SP_DEVICE_INTERFACE_DATA);
10    if(!SetupDiEnumDeviceInterfaces(hdi, 0, &DM7520_GUID, 0, &did))
11    {
12      SetupDiDestroyDeviceInfoList(hdi);
13      return nullptr;
14    }
15    ULONG bufferSize;
16    SetupDiGetDeviceInterfaceDetail(hdi, &did, NULL, 0, &bufferSize,
          0);
17    PSP_DEVICE_INTERFACE_DETAIL_DATA ifDetail = (
          PSP_DEVICE_INTERFACE_DETAIL_DATA) new char[bufferSize];
18    ifDetail->cbSize = sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);
19    if(!SetupDiGetDeviceInterfaceDetail(hdi, &did, ifDetail,
          bufferSize, &bufferSize, 0))
20    {
21      SetupDiDestroyDeviceInfoList(hdi);
22      delete[] ifDetail;
```

```
23      return nullptr;
24    }
25    SetupDiDestroyDeviceInfoList(hdi);
26    HANDLE handle = CreateFile(ifDetail->DevicePath, GENERIC_READ |
          GENERIC_WRITE, 0, 0, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0)
          ;
27    delete[] ifDetail;
28    if(handle == INVALID_HANDLE_VALUE)
29    {
30      return nullptr;
31    }
32    return new DM7520(handle);
33  }
```

Ez a függvény felel a kártya eléréséért, és siker esetén egy, a kártya manipulációjára való osztályra ad egy mutatót, ellenkező esetben nulla a visszatérési érték. Ez a függvény igen nagy mértékben épít a Windows API egyik "al-API-jára", az úgynevezett SetupAPI-ra, amellyel a hardverekről lehet információt kérni, illetve kezelni őket. A *SetupDiGetClassDevs* függvénnyel egy adott eszközosztályba tartozó eszközöket lehet lekérni. Az első paraméter határozza meg az osztályt, ahova mi a kártyánknak megfelelő *RTDDataModule* eszközosztály GUID-ját adtuk meg. Érdekes továbbá még a negyedik paraméter, amivel azt lehet szabályozni, hogy mely eszközöket kapjuk meg. A mi flag-kombinációnk azt jelenti, hogy csak az éppen aktuálisan jelenlévő és használható interfésszel rendelkező hardvereket szeretnénk látni. Ez a függvény egy *HDEVINFO* típust szolgáltat, amely csak egy nyers erőforrás, önmagában nem lehet belőle információkat kinyerni. Ebben nyújt segítséget a *SetupDiEnumDeviceInterfaces* metódus, amelynek segítségével végig iterálhatunk az eszközeinken. Mivel a mi tesztprogramunkat csak belső tesztelésre használjuk, tudjuk, hogy mindig maximum egyetlen egy ilyen kártya lesz a rendszerben, így különösebb vizsgálódás nélkül elfogadjuk az első találatot, amit elmentünk egy *SP_DEVICE_INTERFACE_DATA* struktúrába. Bár a program tesztkörnyezetben való futásra készült, azért az alapvető hibakezelést nem hagytuk ki, így minden egyes hiba esetén felszabadítjuk az eszközinformációs listánkat a speciálisan erre a célra való *SetupDiDestroyDeviceInfoList* hívással.

Miután kezünkben van a feltételezett eszköz, megpróbálunk hozzáférni az interfészéhez, ehhez a *SetupDiGetDeviceInterfaceDetail* nevű függvény lesz a segítségünkre, ennek használata azonban némiképp meg lett bonyolítva. Paraméterként át kell neki adni, hogy mely eszközről kívánunk információt megtudni, valamint egy puffert illetve annak méretét, ahová az adatokat másolni tudja. Azonban az interfészt leíró struktúra minden eszköznél más mérettel rendelkezik, ezért előbb a függvényt puffer átadása nélkül *(17)* kell meghívni, ami a *bufferSize* nevű változónkba beleírja a szükséges méretet. Ezek után már létre lehet hozni a *PSP_DEVICE_INTERFACE_DETAIL_DATA* struktúránkat, amelyhez - elsőre talán meglepő módon - egy *bufferSize* méretű karaktertömböt rendelünk hozzá. Ez azért van így, mert ez a struktúra lényegében csak szövegeket tartalmaz, és az API készítői ezzel a lépéssel kívántak az összes szövegnek elegendő memóriaterületet foglaltatni. Ezt a későbbiekben természetesen minden esetben felszabadítjuk. A megfelelő méret birtokában már meghívhatjuk másodjára is a függvényünket, amely ezúttal már a számunkra

szükséges információkat adja meg.

Ebben a pillanatban minden a kezünkben van ahhoz, hogy hozzáférjünk a kártyához, így az információs listánkat fel is szabadítjuk, majd a *CreateFile* hívással létrehozzuk a *HANDLE*-t az eszközünkhöz. Első paraméterként a megnyitandó "fájl" elérési útját kell megadni, amelyet az imént megszerzett interfész-leíró struktúránk tartalmaz. Fontos még, hogy milyen jogosultságokkal kívánjuk az eszközt megnyitni. Egy kereskedelmi-célú driverben célszerű lehet külön csak olvasható hozzáférést biztosító megnyitási módokat is támogatni, azonban a mi esetünkben ez nem volt fontos. Az *OPEN_EXISTING* flag a megnyitás módját szabályozza és azt jelenti, hogy csak akkor nyissa meg a fájlt, hogyha az már létezik (alapból egy nem létező fájl megnyitásakor létrejön egy új fájl). Az összes többi paraméter figyelmen kívül hagyható.

Végezetül egy dinamikus memóriafoglalást követően térünk vissza, amely természetesen potenciális memóriaszivárgásokhoz vezethet, azonban ha valaki betartja azt a szabályt, hogy egy megnyitott fájlt be is kell zárni, akkor nem lesz gond. Mint az látható, a kártyát manipuláló osztályt egy paraméterezett konstruktoron keresztül hozzuk létre, amely semmi mást nem csinál csak beállítja a *HANDLE* tagváltozó értékét az átadott paraméter értékére:

```
1 DM7520::DM7520(HANDLE handle)
2   : handle(handle) { }
```

Miután az eszközünkkel való műveleteket elvégeztük, a *CloseDevice* nevű statikus függvénnyel lehet azt biztonságosan lezárni. A függvény csak megsemmisíti az osztályt, ami cserébe meghívja az osztály destruktorát, ami bezárja a *HANDLE*-jét:

```
1  void DM7520::CloseDevice(DM7520* device)
2  {
3    delete device;
4  }
5
6  DM7520::~DM7520()
7  {
8    if(handle)
9    {
10     CloseHandle(handle);
11   }
12 }
```

Az írásért felelős metódus a következő:

```
1  void DM7520::Write(DM7520::MEMORY_LOCATION memloc, UINT16 offset,
       DM7520::SIZE size, UINT32 data)
2  {
3    ULONG bytesWritten;
4    DM7520_IO_DESCRIPTOR ioDesc;
5    ioDesc.memloc = memloc;
6    ioDesc.offset = offset;
7    ioDesc.size = size;
8    ioDesc.data = data;
9    DeviceIoControl(handle, DM7520_WRITE_BUFFERED, (LPVOID) &ioDesc,
         sizeof(ioDesc), NULL, 0, &bytesWritten, NULL);
10 }
```

Mint az látható, semminemű ellenőrzést nem végzünk el a kérés továbbítása előtt. Ez a driverünk szűk felhasználása miatt van, egy általános

célú illesztőprogram esetében célszerű lenne még a kernel-módba való átkapcsolás előtt megbizonyosodni róla, hogy érvényes-e egyáltalán a kérés. Itt már megjelenik a már említett *DeviceIoControl* nevű függvény. Ennek használatával lehet az eszközeinknek különféle kéréseket küldeni. A paraméterek között megjelenik maga az eszközünk (*handle*), a konkrét utasítás (*DM7520_WRITE_BUFFERED*), átadjuk a pufferünket, illetve annak méretét. Ezek után jönne a kimeneti puffer, azonban írásnál erre nincs szükség. A *bytesWritten* változónkba fog bekerülni az írt bájtok száma (a driverben ez volt az *Information*). Az utolsó paraméterre pedig csak aszinkron I/O esetén van szükség, amitől mi eltekintünk. A *DeviceIoControl* beküldi a kérésünket kernel-módba, a driver stack tetejére, ahol az előbb-utóbb feldolgozásra kerül. Ekkor az alsó réteg mindig jelzi a felette lévőnek, hogy a kérést végrehajtották, míg el nem jut a driver stack tetejére, ahonnan az irányítás előbb-utóbb visszakerül a *DeviceIoControl*-hoz, ami ezek után visszatér.

Az íráshoz nagyon hasonló módon történik az olvasás:

```
1  UINT32 DM7520::Read(DM7520::MEMORY_LOCATION memloc, UINT16 offset,
       DM7520::SIZE size)
2  {
3    DM7520_IO_DESCRIPTOR ioDesc;
4    ioDesc.memloc = memloc;
5    ioDesc.offset = offset;
6    ioDesc.size = size;
7    ULONG bytesReturned;
8    DeviceIoControl(handle, DM7520_READ_BUFFERED, (LPVOID) &ioDesc,
         sizeof(ioDesc), (LPVOID) &ioDesc, sizeof(ioDesc), &
         bytesReturned, NULL);
9    return ioDesc.data;
10 }
```

Az egyetlen lényeges különbség, hogy itt már kimeneti pufferünk is van, ami történetesen megegyezik a bemeneti pufferrel. Azonban mivel a driver mindig másolatokkal történik, így ebből nem lesz gond. Mivel tesztelési célra készült a driverünk, a visszatérési érték egy 32-bites jelöletlen egész, még akkor is, ha esetleg az olvasni kívánt mennyiség csak egyetlen bájt.

Ezzel lényegében teljessé vált az API-nk, abból a szempontból, hogy mindenre képes. Bár nem a mai, sok absztrakciós réteget alkalmazó fejlesztésnek megfelelően készült, a használata relatíve egyszerű:

```
1 DM7520* device = DM7520::OpenDevice();
2 device->Write(DM7520::LAS0, 0x100, DM7520::BYTE, 0x42);
3 UINT32 result = device->Read(DM7520::LAS1, 0x06C, DM7520::BYTE);
4 DM7520::CloseDevice(device);
```

3 A mérések bemutatása

A méréseket a következő hardverkonfiguráción végeztük el:

Operációs rendszer	Windows 7 x64
Processzor	Intel Core 2 Duo E8500 @ 3.16 GHz
Memória (RAM)	4GB
Videokártya	Intel G33 Express Integrated Video Adapter
Mérés-adatgyűjtő kártya	RTD DM7520
Oszcilloszkóp	HAMEG HMO2022

1. táblázat - A hardverkonfigurációnk

A méréseinket több különböző algoritmuson végeztük el, amelyek mind egy-egy konkrét típusú munkavégzésre voltak kiélezve. A következőkben a tesztprogramok, valamint a tesztelési keretprogram rövid ismertetése következik.

3.1 A tesztprogramok

3.1.1 A keretprogram

A keretprogramnak nevezett kódrészlet nem más, mint a kódnak az a része, amely az összes tesztesetben megegyezik: a kártya felkonfigurálása, valamint a mérési határok jelzése.

```
1  #include "DM7520.h"
2  #include <Windows.h>
3
4  // Global test case specific initialisations...
5
6  int main()
7  {
8    DM7520* device = DM7520::OpenDevice();
9    device->Write(DM7520::LAS0, 0x100, DM7520::BYTE, 0x42); // Reset
10   device->Write(DM7520::LAS0, 0x07C, DM7520::BYTE, 0x05); // Port1
         output
11
12   while(true)
13   {
14     // Per-iteration initialisations...
15
16     device->Write(DM7520::LAS0, 0x074, DM7520::BYTE, 0xFF); //
           Write data
17     Sleep(1); // The oscilloscope is slow
18     device->Write(DM7520::LAS0, 0x074, DM7520::BYTE, 0x00); //
           Write data
19
20     // The testing algorithm...
21
22     device->Write(DM7520::LAS0, 0x074, DM7520::BYTE, 0xFF); //
           Write data
23     device->Write(DM7520::LAS0, 0x074, DM7520::BYTE, 0x00); //
           Write data
24   }
25   DM7520::CloseDevice(device);
26   return 0;
27 }
```

A *DM7520.h* fájl a már korábban bemutatott API-t tartalmazza, míg a *Windows.h*-ra a *Sleep* függvény miatt van szükség. Ezek után a tesztesetek globális változói / deklarációi következnek. A *main* függvényben megnyitjuk a kártyát, betöltjük az alapbeállításokat *(9)*, majd beállítjuk az egyik portot kimenetnek.

Következőként egy végtelen ciklus jön, amelynek egy iterációjában az adott teszteset egyszeri lefutása történik. Először is vannak olyan inicializációs lépések, amelyeket nem tudunk globálisan megtenni, muszáj minden egyes iteráció elején beállítani. Ezt követően kiadunk egy impulzust a kártyán, ez jelenti az algoritmus futásának kezdetét. A jel visszavétele előtt azonban várunk egy milliszekundumot, ugyanis az oszcilloszkópunk nagyobb időtartamok vizsgálatánál sokszor nem vette észre a mikroszekundum nagyságú ideig jelen lévő jelet. Az impulzus kiadása után lefut maga a tesztelendő algoritmus, utána pedig kiadunk még egy impulzust, jelezvén az algoritmus lefutásának végét.

Végezetül a ciklusból kilépve bezárjuk az eszközünket és visszatérünk. Természetesen a kód jelenlegi formájában ez lehetetlen lenne, azonban ezt a tesztprogramot két különféle módon használtuk. Az egyik a kódban látható, végtelen ciklusos módon. Ebben az esetben felesleges a második impulzus kiadása (jobban mondva inkább elhanyagolható), továbbá az eszköz bezárásához sem jutunk el soha. Ebben a módban mindig "erőszakos módon", feladatkezelőből lett bezárva a programunk. Mivel a Windows egy program bezárásakor felszabadít minden hozzá tartozó erőforrást, ezért nem kell aggódnunk amiatt, hogy a kártyához esetleg maradnak bezáratlan hozzáférések.

A másik használati módja a szoftvernek az volt, hogy a ciklusfeltétel egy billentyű lenyomása volt. Ebben az esetben egy billentyű leütésekor lefutott egy iteráció az algoritmusból. Ilyenkor már a második impulzusra is szükség van az algoritmus befejeződésének érzékeléséhez. Volt természetesen egy billentyű, amelynek lenyomására a program kilépett.

3.1.2 ArrayCopy

Ez a tesztprogram a memórián belül adatmásolásra lett kialakítva. Célja a fordító különféle párhuzamosítási képességeinek vizsgálata volt.

```
1 // Global initialisations:
2 #define N 1024*1024*768
3 char a[N], b[N];
4
5 // The algorithm:
6 for(int i = 0; i < N; ++i)
7 {
8   a[i] = b[i];
9 }
```

Mint az látható, tömbök egyszerű másolásáról van szó, amelyek a mai nagy mennyiségű adattal dolgozó alkalmazásokban rengetegszer előfordulnak. A hardverünk korlátozott képességei miatt végül a 768 MB-os blokkok másolása mellett döntöttünk.

Mivel a ciklusnak az iteráció teljesen függetlenek egymástól, így nagy mértékben párhuzamosítható lenne (több szálon), valamint reményeink szerint a generált gépi kódban rengeteg vektorműveletttel is találkozni fogunk.

3.1.3 Parallel

Ez a teszt szintén a fordító párhuzamosítási képességeit próbálja mérni, azonban itt behoztuk a képbe a lebegőpontos utasításokat is.

```
1  // Global initialisations:
2  #define N 1000
3  double a[N];
4  int b[N];
5
6  // The algorithm:
7  for(int i = 0; i < N; ++i)
8  {
9    a[i] = rand() / (double) (rand() + 1) * (-37.0) * (double) rand()
         / (double) (rand() + 1);
10   b[i] = rand() / (rand() + 1) * 37 * rand() / (rand() + 1);
11 }
```

3.1.4 Abstraction

Ennél a tesztnél az absztrakciós réteget teljesítményre vetülő hatásait vizsgáltuk.

```
1  // Global initialisations
2  #define N 50
3  typedef int (*function)(int);
4  function functions[N];
5
6  int myFunction(int n = 0)
7  {
8    if(n == 0)
9    {
10     for(int i = 0; i < N; ++i)
11     {
12       functions[i] = myFunction;
13     }
14   }
15   return (n < N) ? functions[n](n + 1) : 0;
16 }
17
18 // The algorithm
19 for(int i = 0; i < N; ++i)
20 {
21   a[i] = b[i];
22 }
```

Ezzel az elsőre talán kicsit elbonyolítottan megírt rekurzióval az volt a célunk, hogy kikerüljük a fordítónak a rekurziót optimalizáló beállításait (pl. egyszerű ciklussá alakítás). Mivel itt a függvény-pointer tömbünk csak futáskor kerül feltöltésre, amely ráadásul nem is biztos hogy lefut, így a fordító kénytelen minden egyes kódban látható függvényhíváshoz kódot generálni, nem tudja kioptimalizálni azt. Az algoritmusnak egy iterációjában N^2 függvényhívás zajlik le.

3.1.5 Fibonacci

A Fibonacci nevű tesztprogramunk a Fibonacci számok kiszámolásáról szól, amelynek a lépései egymástól függenek, nehezítve ezzel a párhuzamos kód generálását.

```
// Per-iteration initialisations
int x1 = 1, x2 = 1, current = 1, tmp;

// The algorithm
#define N 1000000000L
for(__int64 i = 0; i < N; ++i)
{
  tmp = current;
  current = x1 + x2;
  x2 = x1;
  x1 = tmp;
}
```

3.1.6 StringCPP11

Ez az algoritmus a mai programjainkban rendkívül gyakran előforduló tömbmásolásának egy speciális esetét, a szövegmásolást veszi górcső alá. Ehhez a tesztprogramhoz írtunk egy rendkívül primitív szöveget kezelő osztályt, amely tartalmazza az alap konstruktorokat és a destruktort.

```
// Global initialisations
class MyString
{
  char* _str;
public:

  MyString() : _str(nullptr)
  {}

  MyString(const char* str) : _str(new char[strlen(str) + 1])
  {
    strcpy(_str, str);
  }

  MyString(const MyString& mystr) : _str(new char[strlen(mystr._str
      ) + 1])
  {
    strcpy(_str, mystr._str);
  }

  MyString& operator=(const char* str)
  {
    delete _str;
    _str = new char[strlen(str) + 1];
    strcpy(_str, str);
    return *this;
  }

  MyString& operator=(const MyString& mystr)
  {
    if(this == &mystr)
      return *this;
    return operator=(mystr._str);
  }

  ~MyString()
  {
    delete _str;
  }
};
```

```
40
41 extern const char* pi;
42
43 MyString GetMyString()
44 {
45   return MyString(pi);
46 }
```

A *pi* nevű külső változónk egy másik fájlban van definiálva, ami egy körülbelül 9100 karakterből álló, a π számjegyeit tartalmazó karakterlánc.

```
1 // The algorithm
2 MyString mystr;
3 for(int i = 0; i < 100000; ++i)
4 {
5   mystr = GetMyString();
6 }
```

3.2 Mérések az alapbeállítással

Alapbeállítás alatt a Visual Studio 2012 által alapból felkonfigurált *Release* módot értjük, 64-bites fordítással. Ez a következő fordítási kapcsolókat eredményezte [5] (meghagytuk a Visual Studio által alapból létrehozott sorrendet):

/GS Extra kód generálása a puffer túlcsordulások észrevétele érdekében.

/GL Teljes körű programoptimalizálás. Enélkül csak modulonként (fordítási egységenként) történik optimalizáció.

/W3 A figyelmeztetések szintjét szabályozó kapcsoló. Irreleváns az optimalizációban.

/Gy Az alapból külön-külön elhelyezkedő függvények ún. függvénycsomagokba való csomagolását engedélyezi.

/Zc:wchar_t Hatására a fordító a *wchar_t*-t beépített típusként kezeli. Irreleváns.

/Zi A debug információ generálást befolyásolja. Hatására az információt egy külön .pdb fájlba teszi. Irreleváns.

/Gm- Kikapcsolja az ún. minimális újrafordítást, tehát hiába módosul csak egyetlen fordítási egység, az összes újra lesz fordítva. Irreleváns.

/O2 Optimalizációs beállítás a maximális sebesség elérése érdekében.

/Fd"x64Testvc110.pdb" A .pdb fájl helyét határozza meg. Irreleváns.

/fp:precise A lebegőpontos műveletek sebesség-pontosság arányát határozza meg. Ez a legpontosabb beállítás, és egyben az alapbeállítás is.

/D "_MBCS" Különböző szimbólumokat *#define*-ol.

/errorReport:prompt A fordítóban lévő belső hibák kezelését manipulálja. Alapból ha egy ilyen hiba előjön, szól a felhasználónak, majd továbbítja a Microsoft felé. Irreleváns

/WX- Deaktiválja a "minden figyelmeztetés hiba" funkcionalitást. Irreleváns.

/Zc:forScope Segítségével a for ciklus inicializációs változói csak a ciklus végéig tartanak (ellenkező esetben a cikluson kívüli legközelebbi scope-ig). Irreleváns.

/Gd Az alapértelmezett hívási konvenció a *__cdecl*.

/Oi Engedélyezi a különböző függvényhívások speciális inline-függvényekké való optimalizációját.

/MD A runtime könyvtárak (CRT) többszálú, DLL változatát fogja használni a programunk.

/Fa"x64Test" A generált assembly kód helyét határozza meg. Irreleváns.

/EHsc A kivételkezelés módját határozza meg. Ez a beállítás engedélyezi a C++ kivételek használatát és feltételezi, hogy az *extern "C"*-vel ellátott függvények sosem dobnak kivételt.

/nologo Megakadályozza a fordító induláskor látható logójának mutatását. Irreleváns.

/Fo"x64Test" A generált tárgykód helyét határozza meg. Irreleváns.

/Fp"x64Testtest.pch" Az előfordított header fájl helyét szabja meg. Irreleváns.

Ezzel a beállítással a következő eredményeket értük el (Idő alatt a továbbiakban mindig az egy iterációhoz szükséges átlagos időt értjük):

Teszteset	**Idő**
ArrayCopy	328.55 ms
Parallel	889.07 ms
Fibonacci	532.27 ms
Abstraction	264.56 ms
StringCPP11	1560 ms

2. táblázat - Az alap mérési eredményeink

A továbbiakban egyesével megvizsgálunk pár lehetséges optimalizációs lépést, illetve annak hatásait a kódra.

3.3 A célplatform

Manapság már egyik nagy processzorokkal foglalkozó vállalat sem gyárt olyan általános felhasználású PC / Notebook processzort, amely ne támogatná az x86-64 utasításkészletet, azaz ne lenne képes 64-bites módban futni. Az alábbi, 2010-ben készült statisztika szerint a Windows futtató számítógép kevesebb mint fele volt csak 64-bites, addig a szoftveroptimalizáció egyik fontos területén, a játékpiacon a felhasználók 59,70 százaléka futtatott 64-bites Windows 7 operációs rendszert 2012 szeptemberében a Steam rendszer statisztikái szerint. [4]

Global 64-bit vs. 32-bit Windows Installed Base by OS

Operating System	% 64-bit Installed Base	% 32-bit Installed Base	Total
Windows 7	46%	54%	100%
Windows Vista	11%	89%	100%
Windows XP	<1%	>99%	100%

Source: Windows Update, June 2010, includes all PC's attached and all SKU's

3. táblázat - A Windows platformok megoszlása

Mindezek ellenére a mai napig rendkívül kevés szoftverből érhető el 64-bites változat, annak ellenére, hogy a mi méréseink alapján jelentős sebességkülönbségeket lehetne elérni ezekkel. Természetesen a 64-bites támogatás majdnem hogy megkétszerezi a terméktámogatáshoz szükséges munkát, hiszen két különböző terméket forgalmaznának egyszerre. Minden kódot kétszer kell lefordítani, és a binárisoknak nem szabad összekeveredniük. A későbbi patch-eket is mindig két külön változatban kell biztosítani. Ami a fordítást illeti, ha egy kódot jól írunk meg, akkor mindennemű változtatás nélkül át lehet adni a 64-bites fordítónak.

Egy további probléma ezen a téren, hogy ez már túl "fejlett" egy átlag számítógép-felhasználónak. A legtöbben nem tudják milyen rendszert futtatnak, és ha most elkezdenének a boltokban többféle változatot is forgalmazni, rengetegen vennének rossz változatot, amely további terméktámogatási és jogi problémákhoz vezethet. Ezen a problémán némiképp segíthet a mai letöltéses vásárlás, ahol automatikusan lehetne érzékelni a számítógép konfigurációját és ezáltal a megfelelő változatot telepíteni. Ezt a megoldást azonban még csak a nagyobb vállalatok megoldásaiban lehet felfedezni. Például a Visual Studio webes telepítője telepítés közben vizsgálja meg a rendszert, és tölti le a szükséges komponenseket.

A teszteredményeink egyértelműen kimondták, hogy 64-bites rendszeren látványos a különbség a 32- illetve a 64-bites szoftverek futási sebességei közt:

Teszteset	Idő (x64)	Idő (x86)	Sebességnövekedés
ArrayCopy	328.55 ms	546.19 ms	39.85%
Parallel	889.07 ms	995.56 ms	10.70%
Fibonacci	532.27 ms	1273 ms	58.19%
Abstraction	264.56 ms	304.11 ms	13.01%
StringCPP11	1560 ms	1790 ms	12.85%

4. táblázat - x86 kontra x64 teszteredmények

Mint látható, a legrosszabb esetben is több, mint 10 százalékos teljesítménynövekedést értünk el. Azonban a nagy számokkal dolgozó

alkalmazások, mint például a mi Fibonacci tesztünk is, kihasználhatják a 64-bites operandusú műveletek előnyeit, és így akár másfélszeres sebességnövekedést is elérhetünk.

Ennek a tesztnek az eredménye az lett, hogy egyértelműen beláttuk, hogy jelentős előnyük van a 64-bites programoknak a 64-bites rendszereinken, így azt javasoljuk, hogy amennyiben erre lehetőségünk van, biztosítsunk a szoftverünkből 64-bites változatot is.

3.4 Pufferes biztonsági ellenőrzések kikapcsolása

Az alapértelmezettként beállított kapcsolók között találkozhattunk a */GS* fordítási kapcsolóval, amelynek hatására a fordítónk automatikusan kódot generál a függvényvisszatérésekhez, tömbök indexeléséhez, illetve egyéb olyan utasításokhoz, amelyeknél egy puffer túlcsordulás befolyásolhatná az eredményt. Egy példa erre a következő: egy rosszindulatú hacker egy olyan adat megadásával amely túlcsordítja a puffert, át tudja írni a stack-en a visszatérési címet, potenciálisan rossz indulatú kódnak átadván ezzel az irányítást.

Azonban ha belegondolunk ennek a kapcsolónak a gyakorlati jelentőségébe, hogy ez miért is jöhetett létre, mi a következőre jutottunk: régen, amikor még kevés volt a tárterület és a processzorok feldolgozóképessége, kritikus volt minél tömörebb és gyorsabb kódot írni. Viszont amikor elkezdtek az átlag számítógép paraméterei javulni, a szoftverfejlesztés sokkal biztonságosabbá vált. Legalábbis elkezdődtek a mozgalmak efelé, ezért a Microsoft úgy döntött, hogy a fordítójuk automatikusan generálni fogja ezeket az apró ellenőrzéseket, hiszen sosem fog mindenki biztonságos kódot írni. Természetesen ezek az ellenőrzések nem teljes körűek, így nem hanyagolhatjuk el a kódunkban a biztonsági ellenőrzéseket, azonban egy minimális védelmet nyújtanak. Ugyanakkor egy biztonságosan megírt kód esetében tulajdonképpen feleslegesen kidobott teljesítménynek is nevezhetjük ezt a funkcionalitást.

A tesztjeinkben mi kikapcsoltuk a pufferellenőrző kód generálását, amelyet a */GS-* kapcsolóval értünk el. Ez a következő eredményekhez vezetett:

Teszteset	**Idő (/GS)**	**Idő (/GS-)**	**Sebességnövekedés**
ArrayCopy	328.55 ms	306.51 ms	6.71%
Parallel	889.07 ms	854.66 ms	3.87%
Fibonacci	532.27 ms	520.72 ms	2.17%
Abstraction	264.56 ms	264.46 ms	0.04%
StringCPP11	1560 ms	1490 ms	4.49%

5. táblázat - A /GS kapcsoló hatása

Mint az látható, az alkalmazásaink sebessége konzisztensen javult, azonban meglehetősen csekély mértékben. Felmerül a kérdés, hogy érdemes lehet-e lemondani egy biztonsági rétegről pár százaléknyi sebességnövekedés elérése érdekeben. Mi úgy gondoljuk hogy mivel az új operációs rendszerek mindig egyre több és több biztonságot nyújtanak nekünk, amelyek megvédik az alkalmazásainkat a külső behatolások ellen, ezért megérheti. Amennyiben a kódunkban ügyelünk az alapvető biztonságra, mint például a puffereinkbe

kerülés előtt az adatok hosszának ellenőrzése, úgy nem fog csökkenni a programunk védettsége. Ebben az esetben lényegében "ingyen", mindennemű hátulütő nélkül kapjuk a pár százaléknyi sebességjavulást, amelyet ráadásul egyetlen egyszerű fordítási kapcsoló megadásával el is tudunk érni.

3.5 Az architektúra specifikálása

A Visual Studio 2012 optimalizációs beállításai között találhatunk egy "minimális architektúra" nevű beállítást. A következő lehetőségek közül választhatunk:

/arch:IA32 Csak az x87 FPU utasításkészletét használja.

/arch:SSE Engedélyezi a Streaming SIMD Extensions utasításkészletet.

/arch:SSE2 Engedélyezi az SSE2 utasításkészletet.

/arch:AVX Engedélyezi a 2011-ben debütált Advanced Vector Instructions utasításkészletet.

Egy adott utasításkészlet engedélyezése automatikusan engedélyezi a listában felette szereplőket is.

Sajnos a tesztszámítógépünk processzora nem támogatja az AVX utasításokat, ezért csak az SSE2 utasításokat tudtuk bekapcsolni. Azonban érdemes megjegyezni, hogy a generált kód az AVX utasításkészletet specifikálva eltért a csak SSE2-t használó kódtól, ezért elképzelhető, hogy annak használatával nagyobb sebességnövekedést lehetne elérni.

Az x87 FPU lebegőpontos segédprocesszor megjelenése óta rengeteg idő telt el, rengeteg új technológia jelent meg és a processzorokkal szembeni elvárások is teljesen átalakultak. Amíg régen a processzor órajelét mindig csak növelve nem volt jelentős a párhuzamosítás gondolata, addig manapság szinte egyáltalán nem emelkednek a processzorok órajelei, azonban a párhuzamos művelet végrehajtás mértéke egyre csak nő. Ebben a feladatkörben segítenek az elsősorban párhuzamos adatfeldolgozásra és lebegőpontos számításokra kialakított SSE és AVX utasításkészletek. Ezek nélkül a processzorban csak "mesterséges" párhuzamos végrehajtás jön létre, például az utasítások pipeline-olásával. Az SSE utasításkészlet viszont rengeteg olyan új utasítást tartalmazott, amelyekkel egy adott műveletet egyszerre lehetett több adaton végezni, ez a SIMD (Single Instruction Multiple Data - Egy utasítás, több adat) utasítások lényege.

A minimális processzorarchitektúra megemelésével szinte biztosan javul az alkalmazásunk teljesítménye, amennyiben az tartalmaz bárminemű SIMD utasítássá alakítható részeket (ide tartoznak például a szövegekkel való műveletvégzések is) vagy lebegőpontos műveleteket. Viszont a régebbi processzorok, amelyek nem támogatják az újabb utasításkészletet nem fogják tudni futtatni a programot, az ugyanis "érvénytelen utasítás" hibaüzenettel le fog állni. Az optimális minimum megállapításához tekintsük az alábbi statisztikát:

Technológia	Elterjedtség
SSE2	99.78%
SSE3	99.18%
SSE4.1	57.26%

6. táblázat - Az SSE technológiák elterjedtsége

A táblázatban található értékek a Steam Hardver Kérdőívéből származnak, amelynek profilja a számítógépes játékok. Emiatt esélyes, hogy az átlagnál némiképp magasabb értékeket látunk. Ettől függetlenül azonban kijelenthetjük, hogy az SSE2-es utasításkészletet nyugodtan engedélyezhetjük, hiszen az SSE3-at is csak a felhasználók egy nagyon szűk rétegének nem támogatja a processzora. Ezzel szemben az SSE4, amely az AVX előtt négy évvel, 2007-ben került a processzorokba, alig több mint a felhasználók felének gépében található meg.

Ezek alapján azt mondhatjuk, hogy már csak a korszerűség miatt is érdemes minden programot az SSE2 utasításkészlettel fordítani (A Visual Studio nem biztosít lehetőség külön az SSE3, SSSE3 és az SSE4 utasításkészletek használatára), ha az alkalmazásunk az adott rendszeren gond nélkül fut (tehát támogatva van az utasításkészlet), akkor semminemű hátrány nem ér minket, maximum gyorsabban fog futni a szoftverünk. Alacsony elterjedtsége miatt azonban sem az SSE4-et és ebből kifolyólag az AVX-et sem érdemes még minimálisan támogatott processzorarchitektúraként elvárni.

A mi tesztjeinket tehát */arch:SSE2* fordítási kapcsolóval fordítottuk, amely a következő eredményeket hozta:

Teszteset	Idő	Idő (/arch:SSE2)	Sebességnövekedés
ArrayCopy	328.55 ms	320.88 ms	2.33%
Parallel	889.07 ms	840.12 ms	5.51%
Fibonacci	532.27 ms	516.52 ms	2.96%
Abstraction	264.56 ms	264.51 ms	0.02%
StringCPP11	1560 ms	1390 ms	10.90%

7. táblázat - A /arch kapcsoló hatása

A tesztjeink alapján kijelenthetjük, hogy a teszteseteinkben nem mutatkozott jelentősebb mértékű sebességnövekedés, az utolsó tesztet leszámítva. Ennek ellenére azonban javasoljuk a minimálisan elvárt architektúra megemelését előbb-utóbb, hiszen az egy fejlettebb technológiát használ, amelyből hátrányunk biztosan nem, de előnyünk esetleg lehet.

3.6 Párhuzamos kódgenerálás

Mióta a processzorok órajeleinek frekvenciái abbahagyták a trendszerű növekedésüket a párhuzamosításra csúszott át a fókusz. Az informatika ezen ágazata jelenleg a többmagos, többprocesszoros, több számítógépes rendszerekről szól. Egy szoftverfejlesztő számára a párhuzamosítás több szinten bír jelentéssel. A fejlesztő akár az egész párhuzamosítást a kezébe veheti, ő maga irányíthatja a szálakat. Ebben az esetben azonban rengeteg idő megy el arra, hogy triviálisan párhuzamosítható feladatokat menedzseljen

kézzel, ami ráadásul a kód olvashatóságán is ronthat. Egy nagyon egyszerű példa erre mondjuk egy tömb másolása. Ha a fordító semmilyen párhuzamosítást nem végez el, hanem ehelyett mindent a programozónak kell megcsinálnia, akkor a kód jelentős része a különböző szálak irányításáról fog szólni.

A másik véglet az, amikor mindent a fordítóra bízunk. Azonban sok esetben a fordítóknak esélyük sincs kitalálni, hogy mely részek párhuzamosíthatóak és melyek nem anélkül, hogy a programozó erre bárminemű utalást tenne. Tegyük hozzá, hogy a vektorutasításokat jelen esetben nem tekintjük párhuzamosításnak, hiszen most a többszálú alkalmazásokról van szó. Mivel a szálak kezelése jelenleg még nem egy nagyon széles körben elterjedt dolog, ezért a Visual Studio ezt a hatalmat nem veszi ki a kezünkből, és önmagától soha nem hoz létre külön szálakat az alkalmazásainkban.

A két véglet között helyezkedik el a fordítók dolgának megkönnyítésén alapuló, direktívákon keresztüli párhuzamos kódgenerálás. Ebben az esetben különböző direktívákkal tudunk a fordítónknak "tippeket" adni, hogy a következő kódrészletet esetleg megpróbálhatná párhuzamosítani. Ehhez természetesen átadhatunk extra információkat a fordítónak, megkönnyítve ezzel a dolgát. Ilyen extra információ lehet például, hogy garantáljuk, hogy egy adott változót csak a következő kódrészlet használja, tehát egy másik szál azzal soha nem foglalkozik. A módszer hátránya, hogy nem rendelkezünk akkora irányítással a szálak felet, mint az első esetben. Előnye viszont, hogy mivel csak direktívákról beszélünk, így tulajdonképpen nem része a tényleges forráskódunknak, így ki tudjuk kapcsolni a párhuzamos kód generálását (például egy direktívával, amellyel be-ki kapcsolhatjuk a párhuzamosításról szóló direktívák figyelembe vételét), hogy a kódba bele kellene nyúlnunk, esetleges hibákat előidézve ezáltal. Erre a megoldásra alapul a Visual Studio által alapból is támogatott OpenMP nevű API.

Azonban nem szükséges új API-kat használni, ugyanis a Visual Studiónak van egy saját beállítása, amelynek segítségével a fordító automatikusan tudja a ciklusainkat párhuzamosítani. Persze a ciklusok csak egy nagyon apró részét teszik ki a kódunknak, azonban jelenlétük messze nem elhanyagolható, és a beállítás nagyon egyszerű használata miatt egy próbát mindenféleképpen megérhet mindenkinek. Ez a bizonyos kapcsoló a */Qpar* amely az általunk megjelölt ciklusokat megvizsgálja és amennyiben lehetőséget lát rá, párhuzamosítja őket.

A ciklusaink megjelölése nem tesz mást, mint egy egyszerű direktíva használatát. Ez a direktíva a *#pragma loop(hint_parallel(n))*. Az *n* egy nemnegatív egész paraméter, amellyel megmondhatjuk, hogy a ciklusunkat hány szálra ossza szét a fordító. Ha a paramétert 1-re állítjuk, azaz arra kérjük a fordítót, hogy egy szálon próbálja párhuzamosítani a ciklusunkat, az olyan mint ha semmit sem írtunk volna. Viszont a paraméter nulla értéke esetén a fordító megpróbálja a maximalizálni a szálak számát. A tesztprogramunkat ez a következőképpen befolyásolta:

```
1 // ArrayCopy: algorithm
2 #pragma loop(hint_parallel(0))
3 for(int i = 0; i < N; ++i) {
4   a[i] = b[i];
5 }
```

A többi tesztesetben hasonlóképpen változott a kódunk. Sajnos a hardverünk korlátoltsága miatt a mi esetünkben az egyszerre futó szálak száma kettőben maximalizálódott. Ennek ellenére az eredményeinken így is segített ez az apró trükk:

Teszteset	Idő	Idő (/Qpar)	Sebességnövekedés
ArrayCopy	328.55 ms	300.21 ms	8.63%
Parallel	889.07 ms	824.12 ms	7.31%
Fibonacci	532.27 ms	502.18 ms	5.65%
StringCPP11	1560 ms	1430 ms	8.33%

8. táblázat - A /Qpar kapcsoló hatása

Az eredményeink alapján elmondhatjuk, hogy a szoftvereink teljesítménye egyértelműen megnőtt, átlagosan 7.5%-kal. Bár tény, hogy itt már nem volt elég egy egyszerű fordítási kapcsolót beállítani, a forráskódba is bele kellett írni, azonban ennek a mértéke minimális, továbbá akár minden ciklus elé is beilleszthetjük a *#pragma loop(hint_parallel(0))* sort, hiszen amennyiben a ciklus nem párhuzamosítható, akkor semmilyen hibaüzenetet nem fogunk kapni, egészen egyszerűen olyan lesz, mint ha a direktívánkat oda sem írtuk volna. Összegezve tehát, amennyiben semmilyen más párhuzamosítási módszert nem használunk (saját kezű szálkezelés, OpenMP, stb.), és szeretnénk minimális erőfeszítéssel javítani alkalmazásaink teljesítményein, akkor a Visual Studio saját direktívákon keresztüli megoldása jó választás lehet.

3.7 A C++11 és a mozgató konstruktorok

A C++ legújabb, sokak által C++0x-ként emlegetett, majd végül C++11 névre keresztelt szabványa tavaly érte el hivatalosan a kész állapotát. Bár a Visual Studio a mai napig csak egy részét támogatja a szabványnak, van pár olyan, a jövőben talán kulcsfontosságúvá váló szerkezet, amelyet már a Visual Studio 2010-es verziója is támogatott. Optimalizációs szempontból ezek közül talán a mozgató konstruktor és a mozgató-hozzárendelő operátor a legjelentősebb. Mivel erről a témáról még relatíve kevés szakirodalom létezik, röviden ismertetjük a problémát, amelyre ezek az új konstrukciók megoldást nyújtanak.

Tekintsük a StringCPP11 tesztestünkben létrehozott szövegkezelő osztályunkat:

```
1   class MyString
2 {
3   char* _str;
4 public:
5
6   MyString() : _str(nullptr)
7   {}
8
9   MyString(const char* str) : _str(new char[strlen(str) + 1])
10  {
11    strcpy(_str, str);
12  }
```

```
13
14    MyString(const MyString& mystr) : _str(new char[strlen(mystr._str
          ) + 1])
15    {
16      strcpy(_str, mystr._str);
17    }
18
19    MyString& operator=(const char* str)
20    {
21      delete _str;
22      _str = new char[strlen(str) + 1];
23      strcpy(_str, str);
24      return *this;
25    }
26
27    MyString& operator=(const MyString& mystr)
28    {
29      if(this == &mystr)
30        return *this;
31      return operator=(mystr._str);
32    }
33
34    ~MyString()
35    {
36      delete _str;
37    }
38  };
39
40  extern const char* pi;
41
42  MyString GetMyString()
43  {
44    return MyString(pi);
45  }
```

Ezek után mi például a következő módon meghívjuk a *GetMyString* függvényünket:

```
1  MyString str;
2  // ...
3  str = GetMyString();
```

Ekkor a következő metódusok fognak lefutni:

1. *GetMyString()*
2. Ez létrehoz egy lokális példányt az osztályunkból: *MyString(const char*)*
3. Ehhez át kell másolnunk a szöveget az osztályunkba: *strcpy(...)*
4. Miután visszatértünk a *GetMyString()* függvényből, átmásoljuk a visszakapott osztályunkat: *MyString::operator=(const MyString&)*
5. *MyString::operator=(const char*)*
6. Átmásoljuk az ideiglenes példányunkból a véglegesbe a szöveget: *strcpy(...)*
7. Végül megsemmisül az ideiglenes példányunk: *MyString()*

Mint azt észrevehetjük, az ideiglenes *MyString* példányunk tulajdonképpen teljesen feleslegesen jött létre, hiszen ha lenne rá lehetőségünk, akkor sokkal jobban járnánk vele, hogyha rögtön a célobjektumunkba tudnánk másolni a szöveget, de legalább ne kellene két teljes értékű példányt létrehoznunk, megduplázva ezáltal a másolandó adat mennyiségét. A példánkban ez 9 KB helyett 18 KB adatot jelent, ami talán nem tűnhet soknak, de ez a probléma rendkívül széleskörű, előjön vektorok, listák és további tárolók esetén is, amelyek komplexebb alkalmazások esetén rengeteg adatot tartalmazhatnak, amelyek jó eséllyel szintén kétszer annyi másolást fognak eredményezni.

Erre a problémára nyújt megoldást a C++11 egyik újdonsága, az ún. mozgatási szemantika (move semantics). [3] Ez a bal-értékek (lvalue) és jobb-értékek (rvalue) megkülönböztetésén alapszik. A bal-értékek olyan kifejezések, amelyeknek a programozó a forráskód írása során tudja venni a címét az & (address-of) operátorral. Régebben a bal-érték definíciója az volt, hogy egy olyan bármi, amelyhez hozzá lehet rendelni, azonban ez a *const* kulcsszó megjelenésével megváltozott. Ezzel szemben a jobb-értékek olyan értékek, amelyeknek a programozó nem tudja előre venni a címét, azonban a fordító igen. Erre egy egyszerű példa a *(3 + 4)* kifejezés, amely elé nem tehetünk ki egy address-of operátort, azonban a program futása során mégis lesz címe, amelyhez a fordító hozzá is tud férni, és ezáltal különböző optimalizációs lehetőségek nyílnak meg előttünk. A jobb-értékekre gyakran ideiglenes "változókként" szoktak tekinteni, ugyanis a legtöbb esetben a jobb-értékek szinte azonnal megsemmisülnek. Erre példa az előbb bemutatott ideiglenes *MyString* példányunk, amelyre csak azért volt szükség, hogy a meghívott függvény és a célobjektum közé egyfajta közvetítői szerepet betöltsön.

Ezekre a jobb-értékekre hivatkozni is lehet, méghozzá a && használatával, amely tulajdonképpen egy referenciát nyújt nekünk egy jobb-értékre. Emiatt a "referencia" szó kétértelművé válik, ezért megkülönböztetünk bal-referenciát és jobb-referenciát. A szemléltetés kedvéért tekintsük az alábbi példát, amely szeretné a π számjegyeit feldolgozni valamilyen formában:

```
1 void f(const MyString& pi)
2 {
3   // ...
4   strcpy(local_buffer, pi)
5   // ...
6   // Modify the string
7 }
```

Ez a klasszikus változat. Ebben az esetben kénytelenek vagyunk lemásolni az egész szöveget, annak ellenére, hogy lehet, hogy paraméterként egy olyan változót kaptunk, ami csak ennek a függvényhívásnak a kedvéért jött létre.

```
1 void f(MyString&& pi)
2 {
3   // Modify the string through the parameter
4 }
```

Ha viszont megírjuk ezt a függvényt is, akkor biztosak lehetünk benne, hogy ha ebben a függvényben tartózkodunk, akkor a megkapott paraméterre már senkinek nem lesz szüksége, ezért anélkül hogy azt lemásolnánk, nyugodt szívvel manipulálhatjuk a paramétert, hiszen a függvényünk lefutása után úgyis meg

fog semmisülni. Az *f* függvény meghívásakor tehát a paraméter típusa sokkal nagyobb szerepet kaphat, mint eddig. Amennyiben egy lokális változót adunk át neki, akkor a régi, jól bevált módszer marad, tehát a felső változatot fogjuk meghívni. Azonban ha például *f(GetMyString())*-ként hívjuk meg, akkor már egy ideiglenes változót adunk át paraméterként, így az irányítás a második változatnál fog kikötni.

Ezt a módszert a konstruktorok esetében is lehet használni, hiszen sokszor egy ideiglenes változón keresztül hozunk létre egy újat, ami szintén megduplázza a szükséges másolást. Mozgató konstruktort a következőképpen lehet létrehozni:

```
1 MyString(MyString&& mystr) : _str(mystr._str)
2 {
3   mystr._str = nullptr;
4 }
```

Itt az látható, hogy először egyszerűen átvesszük a paramétertől a *char** változóját, amely az eddigi C++ világban meglehetősen szokatlan gyakorlat lett volna. Ezek után ráadásul a paraméter *_str* változóját még ki is nullázzuk. Ezt a műveletet, amikor a másik osztálytól egész egyszerűen "elvesszük" a változóit, lopásnak hívjuk. Azonban nézzük meg, miért is jó a lopás. Mivel tudjuk, hogy a paraméterként kapott osztály egy jobb-érték, ezért tudjuk, hogy hamarosan meg fog semmisülni, senkinek sincs már szüksége rá. Ezért neki sincs szüksége a változóira! Ha már úgyis rendelkezik egy mutatóval, ami mögött ott van az egész szöveg, miért másoljuk azt át? Egész egyszerűen csak tegyük rá mi a kezünket, hiszen pillanatokon belül meghívásra kerül a paraméter destruktora és csak "kidobnánk" egy teljesen céljainknak megfelelő szöveget. Ezért tehát a konstruktor inicializáló listában átmásoljuk a mutató értékét, majd a konstruktor testében kinullázzuk a másik osztály változóját. Erre azért van szükség, mert ha ott meghagynánk az értéket, akkor a destruktorának meghívásakor arra a mutatóra meghívna egy *delete*-et, amely tönkretenné a szövegünket. Így viszont a *delete* egy nullpointerre lesz meghívva, amely teljesen ártalmatlan.

Az alap gondolat tehát az, hogy mielőtt nekiállnánk átmásolni a dolgokat, először próbáljunk meg minden használható adatot átmozgatni a saját irányításunk alá, majd az ideiglenes változót olyan állapotba kell hoznunk, hogy annak megsemmisülése senki számára ne okozhasson gondot.

Természetesen ha van mozgató konstruktor, akkor lennie kell egy új hozzárendelési operátornak is. Azonban mivel ezt az elnevezést már használjuk, ezért itt is bevezetünk két új elnevezést: a másoló-hozzárendelő operátort (copy-assignment operator) és a mozgató-hozzárendelő operátort (move-assignment operator). Ennek kinézete a következő:

```
1 MyString& operator=(MyString&& mystr)
2 {
3   delete _str;
4   _str = mystr._str;
5   mystr._str = nullptr;
6   return *this;
7 }
```

Az első dolog amit érdemes megjegyezni, hogy itt nincs szükség megvizsgálni, hogy önmagunkhoz rendelünk-e hozzá. Ennek az a magyarázata,

hogy a kapott paraméter egy jobb-érték, tehát nem lehet hozzárendelni semmit. Azonban mi egy mozgató-hozzárendelő operátorban vagyunk, ez azt jelenti, hogy a **this* egy bal-érték. Mivel természetesen semmi nem lehet egyszerre bal-érték és jobb-érték is, ezért kizárt, hogy saját magunkat kapjuk meg paraméterként. Ezt leszámítva minden a már megszokott módon megy. Felszabadítjuk az eddig használt erőforrásainkat, ellopjuk a paraméter használható változóit, biztonságosan megsemmisíthetővé tesszük a paramétert, majd visszatérünk.

Mivel sokszor hasznos, ha a hozzárendelő operátoraink és másoló/mozgató konstruktoraink közül csak a konstruktort írjuk meg, az operátorhoz pedig egyszerűen felhasználjuk a már megírt konstruktort, ezért érdemes megjegyezni a jobb-referenciák egy érdekes tulajdonságát. Mivel egy jobb-referenciának van neve, és függvényparaméterként is megjelenik, így természetesen lehet hozzárendelni és a címét is venni. Ez azt jelenti, hogy a jobb-referencia önmagában egy bal-érték. Emiatt a következő naiv próbálkozás nem fog működni:

```
1 MyString& operator=(MyString&& mystr)
2 {
3   delete _str;
4   *this = MyString(mystr);
5   return *this;
6 }
```

Ebben az esetben ugyanis a *mystr* egy bal-érték, ezért a *MyString(const MyString&)*, vagyis a másoló konstruktorunk lesz meghívva. Ez teljesen logikus is, hiszen nekünk ebben a függvényben még szükségünk lehet a *mystr* változó tartalmára, azonban ha a mozgató konstruktorunknak jobb-referenciaként lenne átadva, akkor az megsemmisítené a változónkat, amely megakadályoz minket a további használatában. Emiatt mindig explicite jeleznünk kell, hogy mi jobb-referenciaként szeretnénk átadni egy változót. Erre a legegyszerűbb megoldás a castolás:

```
1 MyString& operator=(MyString&& mystr)
2 {
3   delete _str;
4   *this = MyString(static_cast<MyString&&>(mystr));
5   return *this;
6 }
```

Azonban a C++11 standard könyvtárait kiegészítették rengeteg új függvénnyel, amelyek közül az egyik pontosan a változók jobb-referenciává tételéért, más néven mozgatásáért felel:

```
1 #include <utility>
2
3 MyString& operator=(MyString&& mystr)
4 {
5   delete _str;
6   *this = MyString(std::move(mystr));
7   return *this;
8 }
```

Ezeket a függvényeket hozzáadva az osztályunkhoz a következőképpen változik a meghívott függvények listája:

1. *GetMyString()*
2. Ez létrehoz egy lokális példányt az osztályunkból: *MyString(const char*)*
3. Ehhez át kell másolnunk a szöveget az osztályunkba: *strcpy(...)*
4. Visszatértünk a *GetMyString()* függvényből, azonban egy ideiglenes változó van a kezünkben, ezért a mozgató-hozzárendelő operátort hívjuk meg: *MyString(MyString&&)*
5. Végül megsemmisül az ideiglenes példányunk: *MyString()*

Tehát megspóroljuk az adatmásolás felét, amely komplex alkalmazások esetén óriási előnyt jelenthet.

Amint az látható, a mozgató konstruktorok és mozgató-hozzárendelési operátorok megírása tovább tart, mint egy fordítási kapcsoló átállítása, azonban a hatásai is jóval nagyobbak. A tesztjeink közül egyedül a StringCPP11 tartalmazott olyan kódot, amelyre alkalmazható volt ez a fajta optimalizáció, ennek az eredménye a következő lett:

Teszteset	**Idő**	**Idő (C++11)**	**Sebességnövekedés**
StringCPP11	1560 ms	589.70 ms	62.20%

9. táblázat - A mozgató szemantika hatása

A több mint 60 százalékos teljesítménynövekedés magáért beszél, mindenféleképpen érdemes implementálni a mozgatási szemantikát az alkalmazásainkban. Amennyiben ezt nem tudjuk megtenni, de esetleg használunk a standard könyvtárakból osztályokat, már akkor is érdemes lehet újrafordítani az alkalmazásainkat, ugyanis a legtöbb fordító már a C++11-re aktualizált könyvtárakat tartalmazza. Tehát ha legutoljára 2008-ban fordítottuk le a rengeteg vektort használó alkalmazásunkat, akkor már egy egyszerű újrafordítástól is jelentősen nőhet a teljesítménye a szoftverünknek, amennyiben rendelkezünk egy újabb fordítóval, amely tartalmazza az új könyvtárakat.

3.8 A GCC fordítóval elért eredmények

Noha a mi vizsgálatunk a Visual Studióról szólt, tettünk egy rövid összehasonlítást a GCC fordítóval is. Itt nem tettünk próbát a különböző kapcsolók egyenkénti hatásainak megfigyelésére, hanem csak ezt az egy, saját kezűleg összeállított kombinációt fordítottuk le: [6]

-fabi-version=6 A C++ ABI (Application Binary Interface) verzióját határozza meg. A hatos érték a legújabb, C++11-et is támogató változat.

-fno-rtti Kikapcsolja az RTTI (RunTime Type Information) generálását.

-fstrict-enums Optimalizációs lehetőséget biztosít a fordító számára azáltal, hogy feltételezi, hogy az egészből enumerációba való konverzió mindig érvényes (tehát mindig létező értéket konvertálunk).

-O3 Az optimalizáció mértékét a lehető legmagasabbra állítja.

-Ofast Engedélyezi a fordítónak, hogy eltérjen a szabványoktól amennyiben az gyorsabb kódot eredményez.

-funsafe-loop-optimizations Engedélyezi a fordítónak, hogy még jobban optimalizálja a ciklusokat azáltal, hogy felteszi, hogy a ciklusfeltételek egy idő után mindig hamisak lesznek (nincs végtelen ciklus), és hogy a ciklusváltozók sosem csordulnak túl.

-fipa-pta Hatására a fordító sokkal nagyobb mértékben vizsgálja át a programunkat optimalizációs lehetőség után kutatva. Ez nagy mértékben megnövelheti a fordítás erőforrásigényét.

-std=c++11 A használt C++ szabványt lehet fele beállítani.

-march=core2 Specifikálhatjuk, hogy mely processzorcsaládra szeretnénk lefordítani a kódot, amely számos egyéb optimalizációs lehetőséget nyit meg. A mi esetünkben nem érzékeltünk semmi hatást.

Sajnos a fordítás során nem volt lehetőségünk 64-bites kód generálására, így a 32-bites változatokat hasonlítjuk össze. Az elért eredményeink a következőek:

Teszteset	Idő (Visual Studio)	Idő (GCC)	Sebességnövekedés
ArrayCopy	546.19 ms	515.60 ms	5.60%
Parallel	995.56 ms	864.40 ms	13.17%
Fibonacci	1273 ms	1185 ms	6.91%
StringCPP11	1790 ms	1020 ms	43.02%

10. táblázat - A GCC fordítóval elért eredményeink

Mint az látható, a Visual Studio alapbeállításai minden esetben alul maradtak az általunk összeállított GCC beállításokkal szemben. A GCC sokkal nagyobb szabadságot ad nekünk az optimalizációs beállítások kezelése ügyében. Bár tényleges sebességnövekedésről beszélhetünk, hasonló eredményeket a Visual Studióval is el tudtunk érni, miután saját kezűleg belenyúltunk a fordítási beállításokba. Az egyetlen jelentős eltérést a StringCPP11 esetén láthatjuk, ahol a GCC sokkal gyorsabbnak bizonyult. Ennek hátterében feltehetőleg az áll, hogy a GCC RVO (Return Value Optimization) technikája már alapból is alkalmazta a C++11-es lehetőségeket, míg a Visual Studio ezt magától nem teszi meg.

4 Összefoglalás

A dolgozat célja az volt, hogy megvizsgáljuk, lehetne-e még fokozni a fordítóink optimalizálási képességeit azáltal, hogy a manuálisan beállítható fordítási feltételeket kielemezzük és megfelelően alkalmazzuk. Ehhez építettünk egy tesztrendszert, amely a pontosság kedvéért kernel-módból vezérelve, hardveresen mérte a programok futási idejét.

Bár a tesztjeink csupán a valós alkalmazásokban előkerülő szűk keresztmetszeteknek csak egy részét fedték le, szinte minden esetben tudtunk pozitív eredményeket produkálni, még ha azok csak pár százalékkal is voltak jobbak az alapbeállításoknál. Összegzésképp kijelenthetjük, hogy egy kis odafigyeléssel és minimális extra munkával 5-10 százalékos teljesítménynövekedést érhetünk el a fordítónk különböző beállításainak állítgatásával.

Megvizsgáltuk a C++ legújabb szabványát is, amely jelentős újításokat hozott be a nyelvbe. Ezek egyike a mozgatási szemantika, amelyekkel több mint másfélszeresére sikerült növelnünk az alkalmazásunk teljesítményét. Noha ehhez már nem elég egy-két beállítást átállítani, ehhez már új dolgok tanulására és gyakorlására van szükség, ami idő- és erőforrás-igényes lehet, azonban a hatása szerintünk magáért beszél.

Mivel a 21. századi életnek a számítógépek kritikus pontjai, nagyon fontos, hogy minél hatékonyabban tudjunk velük dolgozni. Ennek ellenére azonban a mai napig rengetegszer kell a számítógépre várni, ami sokszor blokkolja a munkánkat. Úgy gondoljuk, hogy a vizsgálatunk eredménye ezen tud javítani, amely komolyabb alkalmazások esetén akár sok másodperces megtakarítást is jelenthet.

Irodalomjegyzék

[1] Walter Oney, *Programming the Microsoft Windows Driver Model*, 2nd Edition, 2003.

[2] Real Time Devices USA, Inc., *PCI4520/DM7520/DM7530 User's Manual* Version 2.0, 2003.

[3] Scott Meyers, *Presentation Materials: Overview of the New C++ (C++11)*, 2012.

[4] *Steam Hardware Survey* `http://store.steampowered.com/hwsurvey/`, 2012 szeptember

[5] *Compiler Switches for Visual Studio 2012 (C++)*, `http://msdn.microsoft.com/en-us/library/9s7c9wdw(v=vs.110).aspx`, Microsoft Developer Network, 2012.

[6] *Compiler Switches for GCC*, `http://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html`, 2012.