دانشگاه صنعتی شریف

دانشکده مهندسی برق

گزارش کارآموزی کارشناسی

عنوان:

مدارهای مخابراتی PC-Based

نگارش:

امیر رضا مقیمی

استاد راهنما:

دکتر بنایی

شهریور 1384


بسم الله الرحمن الرحيم

دانشگاه صنعتی شریف

دانشکده مهندسی برق

گزارش کارآموزی کارشناسی

عنوان:

مدارهای مخابراتی PC-Based

نگارش:

امیر رضا مقیمی

استاد راهنما:

دکتر بنایی

شهریور 1384

چکیده:

شرکت خدمات انفورماتیک، مجری پروژه های نصب، گسترش و نگهداری شبکه VSATی است که بانکهای کشور را به هم متّصل می کند و خدماتی مثل عابر بانکها را ممکن می سازد. در گروه تحقیقات و پشتیبانی فنّی (محل کارآموزی من) تلاش بر این است که شرکت چه از جنبه تعمیر و نگهداری وسایل موجود و چه از لحاظ طرّاحی و ساخت تجهیزات جدید خودکفا شود، و من هم در کار مهندسی در هر دو جنبه مشغول فعّالیتهایی شدم. در بخش تعمیر و نگهداری، چندین دستگاه Hughes PES5000 VSAT terminal در شرکت موجود است که نقشه و طرز کار منبع تغذیه آنها برای تعمیر لازم است، که مهندسی معکوس یک نمونه از آنها به من واگذار شد. در بخش طرّاحی هم، driver یک کارت Inroute – Outroute Controller (IOC) که برای ISA bus در PC توسّط مهندسین شرکت طرّاحی شده است، جهت portability به Windows منتقل می شود.

پس این دوره کارآموزی، عملاً دو فاز دارد که هر کدام بخشی از مبحث مدارهای مخابرات دیجیتال است:

1- مهندسی معکوس یک Switching Power Supply:

در این قسمت، یک منبع تغذیه DC که از pulse width modulation برای تنظیم سطح ولتاژ خروجی خود استفاده می کند، بررسی می شود. از روی برد آن، نقشه مدار و دیدی کیفی نسبت به کارکرد آن استخراج می شود. کلّیات پروسه انجام کار و نتیجه های آن در طول این گزارش درج شده اند.

2- انتقال driver یک کارت ISA مخابراتی از محیط DOS به محیط Windows :

در کار طرّاحی مدارهای مخابراتی در قالب کارتهایی برای PC، طبیعتاً نوشتن driver در محیطهای مختلف اهمّیت زیادی دارد. در این قسمت، driver یک کارت Inroute – Outroute Controller که برای DOS نوشته شده بود، به محیط Windows منتقل می شود. آشنایی سطحی با کارت مورد بحث و درک نسبتاً خوبی از معماری سیستم Windows برای چنین کاری لازم است. همچنین آشنایی جزء به جزء با پروسه و استانداردهای driver در Windows ضروری است که کلّیات این مباحث به همراه نتیجه کار (کد نوشته شده) آمده است.

فهرست مطالب:

فصل اول: مهندسی معکوس منبع تغذیه سوئیچینگ (switching power supply)

1-1) مقدمه:

اولین فاز دوره کارآموزی، مهندسی معکوس یک switching power supply بود. بورد مورد نظر، منبع تغذیه یک دستگاه Hughes PES5000 VSAT terminal بوده که نقشه و طرز کار آن جهت اجرای تعمیرات و پشتیبانی فنی لازم است. این منبع، به علت ارائه 3 سطح ولتاژ DC (19.4V، 6.2V و 3.3V) در خروجی و برخورداری از مدارهای محافظت از overload و voltage spike، از لحاظ طراحی و شناخت نیز مورد جالبی است.

1-1-1) طرز کار کلی یک switching power supply:

در منابع تغذیه DC معمولی، برق شهر پس از گذشتن از یک ترانسفورماتور کاهنده، برای تنظیم سطح ولتاژ خروجی، توسط مجموعه یکسو کننده و فیلتر پایین گذر، DC خروجی را تولید می کند. این شیوه، ساده و ارزان بوده و اشکال اساسی آن، حجم زیاد ترانسفورماتور لازم است.

در مقابل، یک switching power supply ابتدا ولتاژ ورودی را DC می کند، سپس با استفاده از یک pulse width modulator (PWM)، این ولتاژ را on/off کرده، نتیجه را (که یک pulse-train با عرض پالس متناسب با ولتاژ خروجی است) از یک ترانسفورماتور عبور می دهد. این رویکرد با حذف سیم پیچ ولتاژ بالا، حجم ترانسفورماتور مورد نیاز را به شدت کاهش می دهد. در طرف ثانویه ترانس، ولتاژ دوباره یکسو شده و عمل PWM دوباره انجام می شود، ولی این بار فقط از یک یا چند مرحله فیلتر پایین گذر – و عموماً یک رگولاتور – می گذرد تا ولتاژ مورد نظر به دست آید.

1-1-2) وظایف من در این فاز:

در این دوره کارآموزی، استخراج نقشه مدار از روی بورد، با تمام جزئیات (مقدار، حد ولتاژ، شماره و یا datasheet هر المان)، بر عهده من بود.

1-2) پروسه انجام کار:

بورد این منبع تغذیه، تک لایه و تک رو با تعداد IC نسبتاً کم است؛ بنابراین اکثر اتصالات آن با چشم قابل تشخیص اند. پس عمده کار استخراج نقشه در بررسی عینی و رسم مشاهدات خلاصه می شود. البته به علت تراکم المانها و تنوع اندازه آنها، در برخی نقاط باز کردن قطعات (عمدتاً heat sink، خازنهای بزرگتر و ترانسفورماتور) جهت مشاهده و بعضاً تست و اندازه گیری لازم است. از طرفی هم به علت محدود بودن تعداد این بوردها، سعی بر آن است که از دستکاری بی مورد پرهیز شود تا در نهایت بتوان منبع را دوباره به کار گرفت.

روی این بورد، چند خازن SMD، چند سلف و یک ترانس هم وجود دارد که همگی باید با LCR متر اندازه گیری شوند. در مورد ترانسفورماتور، فقط مقاومت و اندوکتانس هر سیم پیچ به طور جداگانه اندازه گیری می شود؛ برای اهداف فعلی اندازه گیری نسبت دور سود خاصی ندارد (ولتاژهای ورودی و خروجی مشخص اند).

طبیعتاً هم پس از طی مراحل مذکور، نقشه حاصل به کامپیوتر وارد می شود. من برای قبیل کارها، از Orcad استفاده می کنم.

1-3) نتیجه کار – تشریح مدار:

این مدار، مشابه توضیحات قسمت 1-1-1، ابتدا ولتاژ را با استفاده از یک پل دیود ساده و خازن ولتاژ-بالا DC کرده، نتیجه را به کمک یک PWM controller، به قطار پالس تبدیل می کند. در همین طرف اولیه ترانس (ضمیمه الف – صفحه اول)، دو thyristor وجود دارد که در صورت on شدن، PWM controller را از کار می اندازند. Gate این دو توسط دو optocoupler کنترل می شود که به نوبه خود از مدارهای محافظتی (ضمیمه الف – صفحه سوم، توضیح در ادامه) فرمان می گیرند. ولتاژ PWM Vcc از طرف ثانویه ترانس تأمین می شود (گره P1) که به کمک مجموعه یکسو کننده و رگولاتور خیلی ساده ای در حدود 20V نگاه داشته می شود. قطار پالس خروجی به کمک ترکیب دارلینگتون-مانند یک BJT و یک MOS-FET تقویت شده به ورودی ترانس اعمال می شود.

در ثانویه (ضمیمه الف – صفحه دوم)، به غیر از تغذیه PWM مذکور، سه خروجی دیگر از ترانس گرفته می شود (به شکل و ساختار ترانس توجه شود). خروجی اول (بالای صفحه) با عبور از ترکیب ساده یکسوساز نیم موج و چند مرحله فیلتر پایین گذر، DC شده مستقیماً به خروجی 19.4V می رود. همین خروجی پس از دوباره رگوله شدن (در حدود 12V)، برای تغذیه fan خنک کننده - که در بدنه منبع تغذیه نصب شده است – مورد استفاده قرار می گیرد.

خروجی دوم ترانس (وسط صفحه) پس از DC شدن (گره SB1) برای تغذیه PWM controller دوم، فیدبکهای حول آن و opamp های محافظتی مصرف می شود. این PWM، برای تولید خروجی 3.3V (حساسترین سطح ولتاژ) به کار گرفته شده؛ بدین ترتیب که خروجی سوم ترانس پس از DC شدن (پایین صفحه) با کمک تقویت یک MOS-FET، توسط PWM با کنترل حلقه بسته مدوله شده با گذشتن از چند لایه فیلتر پایین گذر، 3.3V را به دست می دهد. سطح 6.2V هم از همین خروجی ترانس ساخته می شود؛ پس از DC شدن، یک رگولاتور ساده این خروجی را تثبیت می کند.

یک قسمت دیگر این مدارها، ولتاژ هر سه خروجی را به کمک زینر دیود چک می کند، و در صورت افت بیش از حد هر کدام، ترانزیستور Q702 را (که در حالت عادی وصل است) قطع می کند. این باعث می شود که SB8 و SB2 هر کدام افزایش سطح پیدا کنند. افزایش SB8 باعث قطع Q505 (ضمیمه الف – صفحه دوم) و در پی آن غیر فعال شدن PWM دوم می شود. از طرف دیگر افزایش SB2 منجر به فعال شدن Optocoupler دیگر (ISO1) می شود که PWM اول را قطع می کند.

یک مدار کوچک هم در صورت افت گره SB12 (ولتاژ خروجی اول ترانس)، از طریق کنترل ISO2، عملاً فیدبک لازم برای تنظیم ولتاژ PWM اول را تامین می کند.

4–1) روند ادامه کار :

با توجه به نقشه کامل مدار (ضمیمه الف) و تحلیل مختصری که در بالا ارائه شد، کاری که در رابطه با این منبع باقی می ماند به شرح زیر است :

از آنجایی که هدف اصلی این پروژه، تعمیر و پشتیبانی فنی تجهیزات فعال است، مهم ترین کاری که باقی می ماند، شناسایی نقاط حساس و اساسی مدار و به دست آوردن رفتار نرمال و سالم آنها است. اگر این اطلاعات حاصل شود (از طریق تحلیل، شبیه سازی و یا آزمایش) می توان در صورت بروز مشکل به سرعت محدوده معیوب را مشخص کرد و به تعویض قطعات پرداخت. مثلا به نظر می رسد که همان گره SB12 (که برای فیدبک دادن به PWM اول استفاده می شود) گزینه معقولی است. اگر میزان ولتاژ DC این گره کم یا زیاد باشد یا ترانس مشکل پیدا کرده و یا فیدبک درست عمل نمی کند (زمانبندی PWM اول خراب است) اگر ولتاژ آن ریپل زیادی داشته باشد می توان نتیجه گرفت که یکی از خازن های 470µF سوخته است. از طرفی اگر گره ای که در ضمیمه الف – صفحه 3، خروجی چهار Opamp را به هم وصل می کند، به طور پیوسته نزدیک GND باشد و خروجی ها در range معقولی باشند، نتیجه می گیریم که یکی از Opamp ها یا رگولاتورهای مربوط به سه خروجی (IC703، IC704 و IC705) دچار مشکل شده اند.

اگر هم دقت و جزئیات بیشتری مورد نظر باشد، می توان عمداً بعضی از قطعات کلیدی مدار را باز یا اتصال کوتاه کرده اثرات این کار را روی بقیه مدار مشاهده کرد. این اطلاعات هم برای عیب یابی و هم از دیدگاه تحلیل و طراحی (نقش هر المان در مدار) مفید است.

1-5) نتیجه گیری:

این فاز، که حدوداً دو هفته اول دوره کارآموزی را به خود اختصاص داد، اولین تجربه مهندسی معکوس من و نیز اولین برخوردم با یک switching power supply است؛ و در طول آن با طرز کار Thyristor و Optocoupler و اساس چنین منابعی آشنا شدم. همچنین پروسه مهندسی معکوس برای من شفاف شد و با تحلیل کیفی مدار های ناشناخته مأنوس تر شدم.

فصل دوم : انتقال یک DOS-Based Driver به محیط Windows

2-1) مقدمه :

در فاز دوم دوره کارآموزی، در واقع فاز اصلی آن، Driver یک کارت IOC که در شرکت طراحی شده بود از محیط اولیه آن (DOS) به Windows منتقل می شود. هدف این فاز حصول قابلیت نصب کارت روی کامپیوتر های عادی است که اکثراً با Windows کار می کنند.

2-1-1) معرفی کارت IOC :

کارت IOC (Inroute-Outroute Controller)، یک کارت مخابراتی است که طراحی آن با FPGA انجام شده است. مشخصات دقیق آن برای کار حاضر لازم نیست، فقط تا این حد کافی است که بدانیم:

- کارت IOC مذکور، یک ISA card است. تأثیر این حقیقت روی ساختار Driver در بخش بعد بررسی می شود.
- یک Interrupt Request دارد، پس Driver تنها یک ISR نیاز دارد. این موضوع (همچنان که خواهیم دید) برای یک Windows Driver، مقدار کد لازم را کاهش می دهد. البته این تک Interrupt، 16 منشأ دارد که با خواندن رجیستر های داخلی کارت درون خود ISR تشخیص داده می شود.
- I/O range آن، بین 800H تا 87FH است.
- حافظه داخلی ندارد و از DMA هم استفاده نمی کند.
- Driver آن از چند مورد متغیر Global استفاده می کند که در سیستم Protected و Context-Based مثل Windows، باید به آنها توجه خاص شود.
- عملکرد کلی آن این است که به طور سریال، Packet دریافت و ارسال می کند و Data را در یک سری Buffer که در حافظه اصلی است نگاه می دارد. در مقابل درخواست های I/O از نزم افزار مرتبه بالاتر هم محتویات این بافر ها را می نویسد و یا می خواند.

2-1-2) ابزار لازم برای کار:

برای نوشتن Driver در محیط Windows، یک بسته نرم افزاری از Microsoft به نام Driver Development Kit (DDK) مورد نیاز است. البته این مجموعه در بازار ایران به راحتی یافت نمی شود و زمان نسبتاً زیادی طول کشید تا آن را بدست آورم. این مجموعه خود یک سری ابزار build دارد که می توان با نصب یک appwizard ساده از داخل محیط Visual C++ 6 هم به آنها دسترسی داشت.

2-1-3) پیشنیاز ها و آمادگی:

من تا قبل از این تجربه برنامه نویسی در Windows به زبان C++ را نداشتم، پس مدتی وقت لازم بود که با محیط Visual Studio و طرز کار المان های Windows (محیطی کاملاً شیءگرا و مملؤ از استاندارد و قرارداد که در آن برای هر کاری رویه از پیش تعیین شده ای وجود دارد) آشنا شوم. طبیعتاً برای کارهای driver نویسی نیز ابتدا باید با معماری سیستم آشنا شوم، به همین سبب از کتاب Windows 95 System Programming Secrets نوشته Matt Pietrek مطالعاتی در مورد سیستم Windows و برنامه نویسی در آن انجام دادم. البته کمک زیادی هم از مجموعه MSDN گرفتم. البته نتیجه این مطالعات ارتباط مستقیم با بحث حاضر ندارد و فقط برای من بستری است که بحث های مربوط به driver و تقابل اجزای مربوطه در Windows را بهتر درک کنم؛ به همین علت از ذکر آنها در این گزارش خودداری کرده خواننده را به دو منبع مذکور ارجاع می دهم. در باقی گزارش فرض شده است که خواننده با مفاهیمی چون Kernel و User، Memory Management در Windows، Processها و Threadها، Dynamic Linking و Messaging آشنایی مختصر دارد.

2-2) I/O Model در Windows و ساختار یک Windows driver:

طبیعتاً در ادامه کار، روشهای I/O در Windows و برخورد این سیستم را با driver باید بررسی کرد. منبع اساسی تمامی مطالب مربوط به این بحث، همان MSDN است. در این مجموعه، یک قسمت عمده به نام DDK documentation وجود دارد که به تشریح همین موارد پرداخته، قسمتهایی از kernel را که عموماً فقط در I/O نقش دارند توضیح می دهد. از این پس هر چه ذکر می شود، حاصل مطالعه MSDN و آزمایش نتایج است:

2-2-1) بخشهایی از Windows که در I/O و driver support نقش دارند:

برای driverهایی که به صورت kernel-mode اجرا می شوند (driverهای user-mode بحثی متفاوت اند که به آن نمی پردازیم؛ به دلیل ISA بودن کارت در ابتدا قابل استفاده نیستند – توضیحات در بخشهای آتی)، تمام درخواستهای I/O که از برنامه های دیگر صادر می شوند نخست به I/O Manager فرستاده می شوند. یعنی عملاً هیچ برنامه ای از نوع و ساختار driver مورد نظر اطلاع ندارد و سیستم file access برای همه عملیات I/O استفاده می شود. I/O Manager این درخواستها را به صورت Input/Output Request Packet (IRP) به driver مربوطه تحویل می دهد. driver هم اگر درست نوشته شده باشد، مستقیماً با کد assembly عملیات را انجام نمی دهد، بلکه همه low-level I/O توسط Hardware Abstraction Layer (HAL) انجام می گردد و driver از HAL به عنوان واسطه استفاده می کند. Interruptها نیز مستقیماً driver را صدا نمی کنند، همه interruptها به I/O Manager می روند و I/O Manager آنها را به driver مربوطه می رساند.

برای deviceهایی که از Plug and Play (PnP) بهره می جویند، PnP Manager در ابتدای کار زمان تشخیص device) با I/O Manager ارتباط برقرار کرده driver را به آن می شناساند، ولی کارت ما به حکم ISA بودنش با PnP کاری ندارد. بخشهایی مانند Power Manager هم چنین اند.

از آنجایی که در Windows همه چیز یا object است و یا اینکه سیستم از طریق یک object آن را می شناسد، و از آنجا که تمامی objectها و classها در Windows از طریق Object Manager مشاهده و کنترل می شوند، این قسمت نیز برای driverها و deviceها (که هر کدام object مربوط به خود را دارند) حیاتی است. هر Windows driver یک Registry key نیز دارد که هنگام startup به سیستم می فهماند که باید فعالش کند. البته از Registry استفاده های دیگری هم می توان کرد، ولی فعلاً برای این کار ضرورت ندارد.

2-2-2) نکات اولیه و اساسی در مورد driverها:

- اولین موردی که هر driver نویس باید به آن توجه کند، انتخاب زبان برنامه نویسی مناسب است. از آنجا که Windows مختص یک نوع CPU یا مجموعه سخت افزار خاصی نیست، برنامه ای که برای Windows نوشته می شود نیز اگر بخواهد source-code compatibility خود را با Windows در هر حالتی حفظ کند، نباید شامل کد assembly (یا هیچ جزء وابسته به سخت افزار دیگر) شود. همان طور که ذکر شد، در Windows همه low-level I/O به واسطه HAL انجام می شود. از طرفی، چئن خود Microsoft مجموعه کد بسیار وسیع و ابزار

بسیار قوی در رابطه با driver نویسی در اختیار نویسندگان قرار داده که همگی به زبان C نوشته شده اند، زبان C عموماً بهترین انتخاب برای یک driver است.

- در یک driver، باید از استفاده از هر structure یا object که حجم و یا مشخصات آن روی platformهای مختلف یا احیاناً بین اجراهای مختلف متفاوت باشد (مثلاً stringهایی که بر حسب مورد از کاراکترهای ASCII یا UNICODE استفاده می کنند) پرهیز شود.
- برخی از کارهایی که در برنامه های عادی می توان با استفاده از موارد تعریف شده در headerهای <windows.h> و امثالهم انجام داد، یا system functionهای دیگر، تنها در user-mode قابل استفاده اند. البته اکثر این موارد معادلهای kernel-mode نیز دارند، ولی باید در به کار گیری آنها دقت کرد.
- <ntddk.h> که همراه DDK می آید، همه data structureها و توابع پایه Windows را تعریف می کند، نیازی به استفاده از <windows.h> و ... نیست.

2-2-3) معرفی ساختار یک kernel-mode Windows driver:

در نگاه خیلی ساده، یک Windows driver، یک Dynamic Link Library با پسوند sys. است که توابعی که export می کند، اولاً نقشها و وظایف مشخصی دارند، ثانیاً همه توسط اعضای kernel صدا می شوند و ثالثاً همه در kernel mode ولی با IRQLهای متفاوت اجرا می شوند. (IRQL، سطح حساسیت کد است. کدی که در یک IRQL اجرا می شود می تواند در هر زمانی ایست داده شود تا کد با IRQL بالاتر اجرا شود. هر کدام از توابع مشخص یک driver در IRQL مشخصی اجرا می شود.) همه driverهای فعال در یک سیستم در %system%\drivers کامپیوتر قرار دارند، مثلاً برای یک دستگاه خاص این path می تواند C:\WINNT\system32\drivers باشد. Windows در زمان startup، همه driverهایی را که در Registry در محل مشخصی key دارند، صدا می کند.

این صدا کردن، دو مرحله اصلی دارد: 1- ساختن یک structure تعریف شده از نوع DRIVER_OBJECT که معرف یک driver فعال در سیستم است. 2- صداکردن تابع DrverEntry (یکی از توابع export شده لازم). از موقعی که DriverEntry باز می گردد، در صورتی که driver، non-PnP باشد (که در این مورد، هست)، سیستم این driver و device را فعال فرض می کند، و I/O Manager همه interface خود با driver را از راه همان DRIVER_OBJECT انجام می دهد.

در kernel-mode، driverها چند لایه دارند. یک low-level driver یا bus driver، مستقیماً عملیات I/O انجام داده و interruptها را پاسخ می دهد. اگر device نیاز داشته باشد، می توان driverهای مرتبه بالاتری (مثل function driver یا filter driverها) بالاسر این bus driver گذاشت که

کاری به interrupt request و دیگر عملیات low-level ندارند و فقط IRPها را پردازش کرده، در صورت لزوم به driverهای مرتبه پایینتر ارجاع می دهند. این کارت IOC از آنجایی که یک ISA card است و اصولاً برای ISA bus، bus driver از پیش نوشته شده ای وجود ندارد، و از طرفی عملیات driver از پیش نوشته همگی low-level اند، پس در این مورد خاص تبدیل driver قدیمی به یک bus driver معقول است.

هر driver، موظف است یک structure تغریف شده DEVICE_OBJECT تولید کند (به کمک توابع I/O Manager) که در فضای kernel قرار می گیرد و به همراه DRIVER_OBJECT از پیش تولید شده، معرف driverها در مقابل بقیه kernel باشد. دلیل اصلی جدا کردن این دو، به نظر می رسد بیشتر مواقعی باشد که بخواهیم با یک driver (عملاً یک فایل sys.)، چند device را کنترل کنیم (این مفهوم در deviceهای PnP، مثلاً و به خصوص USB portها، مطرح می شود، که در آن صورت به یک driver این فرصت داده می شود که چندین DEVICE_OBJECT ثبت کند). از آنجایی که توابع مختلف driver در IRQLهای مختلف و contextهای مختلف (ر.ک. MSDN، مبحث Processes) صدا می شوند، این DEVICE_OBJECT، که می توان از طرق مختلف آدرس آن را در اختیار همه این توابع مختلف قرار داد، فضای global خوبی به نظر می رسد که می توان متغیرهایی مثل شمارنده ها و بافرها و system state را در آن قرار داد. به همین علت، I/O Manager این امکان را می دهد که حجم دلخواهی را به این object اضافه کنیم، و آن را device extension یا device context می نامد.

توابع export شده از یک driver، عمدتاً وظیفه پاسخ دادن به interruptها (در bus driverها) و IRPهاست. IRPها می توانند از applicationها، دیگر driverها، خود سیستم (معمولاً اجزای kernel مانند PnP Manager) و ... سرچشمه گیرند ولی همه در I/O Manager تولید و بین driverها توزیع می گردند. در بخش بعدی، توابع معمول در یک driver – و مباحث مربوط به هر کدام – توضیح داده می شود.

4-2-2) توابع معمول export شده توسط یکWindows kernel-mode bus driver:

1- تابع DriverEntry: همان طور که ذکر شد، در هنگام startup، Windows از داخل registry، فایل sys. همه driverها را پیدا کرده و تابع DriverEntry آنها را (که باید دقیقاً همین نام را داشته باشد – شبیه تابع main در برنامه های معمولی) صدا می کند. این تابع باید چند کار انجام دهد: ساختن device object و (در صورت چند لایه بودن) متصل نمودن آن به driver stack؛ ایجاد device extension در صورت نیاز؛ export کردن بقیه توابع مثل ISR و توابع Dispatch؛ و initialize

کردن متغیرهای global و registerهای خود device. در صورت نیاز، این تابع می تواند کارهای دیگری نیز انجام دهد که بستگی به نویسنده دارد.

2- تابع AddDevice: تابعی با اسم دلخواه که باید در DriverEntry، export شود. این تابع برای deviceهای PnP، به موقع تشخیص اولیه device جدا می شود. در deviceهای non-PnP، کارایی آن حذف می شود: قسمتی از آن به کلی از بین می رود و قسمتی هم باید به DriverEntry منتقل شود (چون در این صورت، فرض بر این است که device به هنگام startup، فعال است) که در بالا ذکر شد. در IOC driver، من تابعی خالی را برای AddDevice، export کردم ولی مطمئن نیستم که این کار لازم است.

3- توابع dispatch: گفتیم که I/O Manager هر درخواستی که از طرف دیگر اجزای سیستم یا برنامه های user-mode می آید را به یک IRP تبدیل کرده به driver می دهد. یک IRP در واقع یک structure تعریف شده است که برای معرفی نوع درخواست، دو DWORD دارد، یکی major function code و دیگری minor function code. مثلاً هر درخواستی که از PnP Manager می آید، دارای major code برابر IRP_MJ_PNP (همه مقادیر این کدها با macro مشخص می شوند) و بسته به نوع درخواست، minor code مشخصی است. از طرفی، در driver object یک آرایه از pointerها وجود دارد که هر کدام متناظر با یکی از major codeها است، و driver باید در تابع DriverEntry خود آدرس handler مربوط به هر کدی را که پردازش می کند در آن قرار دهد. به این توابع، dispatch routine گفته می شود. I/O Manager به هنگام تولید IRP، از داخل driver object، آدرس تابع مربوط به major code حاضر را پیدا کرده آن را صدا می کند. تشخیص minor codeها بر عهده driver است.

4- تابع StartIo: چون توابع dispatch به هنگام دریافت IRP صدا می شوند، ذاتاً ماهیتی آسنکرون دارند و حتی ممکن است چند تابع تقریباً در یک زمان صدا شوند. این بدین معنی است که هنگام دریافت IRP، ممکن است driver مشغول عملیات I/O دیگر یا حتی سرویس کردن یک interrupt باشد؛ به عبارت دیگر device در هنگام دریافت IRP ممکن است اشغال باشد (این مزیت عملکرد multitasking در Windows است که چند قطعه کد در خلال هم و ظاهراً همزمان اجرا می شوند). پس در توابع dispatch که در خلال آنها نیاز به انجام I/O وجود دارد، باید این عملیات را به زمان دیگری موکول کرد. I/O Manager این کار را انجام می دهد. می توان در انتهای یک dispatch routine به I/O Manager فهماند که هنوز عملیات I/O باقی مانده است، آن عملیات را مشخص کرد و سپس اجرای تابع را تمام کرد. I/O Manager هم در اولین فرصت که device بیکار شد، تابع StartIo را صدا می کند تا I/O را انجام دهد. طبیعتاً در انتهای StartIo باید به I/O Manager فهماند که کار تمام

شده و عملیات بعدی را می توان شروع کرد. این کار باید آخر عملیات I/O در سرویس کردن interruptها هم انجام شود.

5- تابع InterruptService: همان طور که گفته شد، در Windows همه interruptها توسط I/O Manager (در مرحله اول) پاسخ داده می شوند. این ارگان سپس تابع InterruptService در driver مربوطه را صدا می زند (طبیعتاً تابع DriverEntry باید شماره interrupt مربوط و آدرس ISR را به I/O Manager گزارش دهد). ولی ملاحظه ای مهم در این مرحله کار وجود دارد:

در بخشهای پیشین اشاره شد که هر تابعی در driver، در IRQL مشخصی اجرا می شود و بالاترین سطوح IRQL، سطوح DIRQL، مختص ISRها است، چون ISRها باید فوراً و بدون ایست اجرا شوند. پس عملاً یک ISR در هنگام اجرا، تمام وقت CPU را به خود اختصاص می دهد – مگر اینکه interrupt با حساسیت بیشتری در خلال آن داده شود – و این با روح multitasker سیستم منافات دارد؛ حتی در برخی موارد برای سیستم مرگبار است. بنابراین ISRها باید تا حد ممکن کوتاه باشند و عملیات I/O در آنها به حداقل ممکن برسد. به همین علت یک ISR معمولاً اطلاعاتی را که در مورد وضعیت سیستم و device نیاز دارد، می خواند و در یک context قابل دسترس در همه driver (مثلاً device context) ذخیره کرده، اجرای خود را متوقف می کند. البته قبل از خروج، یک درخواست DPC یا Deferred Procedure Call انجام می دهد که عملیات لازم در اولین فرصت ولی در IRQL پایینتر انجام شود. I/O Manager این درخواستها را به نوبت، با صدا کردن تابع DpcForIsr مربوطه، سرویس می کند.

6- تابع DpcForIsr: این تابع، که DriverEntry آن را در driver object به I/O Manager معرفی می کند، وظیفه پردازش interrupt را دارد. کلیه عملیات I/O (غیر از خواندن وضعیت اولیه) و کلیه عملیات دیگر باید در این تابع انجام شود. در آخر کار هم (مثل آخر StartIo) باید به I/O Manager فهماند که کار تمام شده و می تواند StartIo را با عملیات بعدی (در صورت وجود) صدا کند.

- نکته ای باید اینجا ذکر شود، آن هم اینکه در صورت استفاده device و driver از بیش از یک interrupt، پروسه بالا دیگر جواب نمی دود و در خیلی از کارها باید تسهیلات I/O Manager را دور زد و مستقیماً با توابع kernel وارد عمل شد، که حجم و پیچیدگی کد را به شدت افزایش می دهد. خوشبختانه در مورد کارت IOC حاضر، این کار لازم نیست.

7- توابع Close، Create و ...: این توابع و توابع دیگر در مقابل عملیات سیستم و کد لایه های بالاتر واکنش می دهند. جزئیات آنها به بحث حاضر ارتباطی ندارد اما، مثل AddDevice، گذاشتن آنها – هر چند به صورت dummy function – ضرری ندارد و گسترش کد را در آینده نیز ساده تر می کند.

توجه: برای بررسی جزئیات مباحث بالا و کسب اطلاعات بیشتر، رجوع کنید به مجموعه MSDN، بخش Kernel-Mode Driver Architecture: Windows DDK.

2-3) پروسه انتقال کد قدیمی به ساختار یک kernel-mode Windows driver:

اولین مرحله انجام کار، تصمیم گیری در مورد ساختار driver حاضر است؛ و اینکه چه بخشهایی را به چه شکل در driver قرار دهیم. به عنوان مثال:

- چون کارت، یک ISA card است، با PnP Manager و Power Manager و IRPهای آنها کاری نداریم و AddDevice هم یک dummy function است.
- چون کارت، ISA است و آدرسهای I/O مشخصی دارد، می توان در خود کد driver این آدرسها را قرار داد و مستقیماً از macroهای HAL استفاده کرد. (در مورد این driver فقط byte رد و بدل می شود، پس فقط دو macro مورد نیاز اند: READ_PORT_UCHAR و WRITE_PORT_UCHAR)
- ISR موجود نسبتاً طولانی است، پس حتماً باید به دو بخش (همچنان که توضیح داده شد) تقسیم شود.
- باید چند dispatch routine برای اعمالی چون read، write و control داشته باشد. طبیعتاً یک StartIo هم لازم است.
- driver قدیمی چندین متغیر و بافر global دارد. در Windows، به علت اینکه driver یک DLL است که توابع آن در context نامعلوم و بعضاً اتفاقی صدا می شوند، باید این globalها را در محل مشخصی قرار داد. بهترین جا، خود device extension است. برای این کار، یک structure در header تعریف شده که همه این globalها را در بر می گیرد.

این موارد که رعایت شد، باید بدنه اصلی driver (در واقع تعریف متغیرها و prototype توابع) نگاشته شود. پس از آن به سراغ DriverEntry می رویم:

این تابع، کاملاً از کد جدید تشکیل شده است و فقط در آخر آن، توابعی از driver قدیمی صدا زده می شوند که کارشان device initialization است. قبل از آن همه موارد مطرح شده در بخش گذشته، به علاوه initialize کردن محتویات device extension، انجام می شود. اینجا نیز مانند هر جای دیگری در کد، همه عملیات I/O موجود باید به macroهای HAL برگردانده شوند.

سپس کار به سراغ interrupt می رود. در DriverEntry، interrupt شماره 13 به این driver نسبت داده شده است و آدرس ISR و DPC نیز در اختیار I/O Manager قرار گرفته اند. با توجه به کد

قدیمی، دیده می شود که دو رجیستر StatusAddr و RDIVect در تعیین منشأ interrupt و نوع واکنش driver به آن نقش دارند، پس ISR این دو را خوانده و در محل مربوطه در device extension ذخیره می کند. سپس یک DPC صدا کرده خارج می گردد. همه بدنه ISR قدیمی DPC routine قرار می گیرد، با این تفاوت که اولین مورد خواندن StatusAddr و RDIVect از حافظه extension (به جای خود کارت) خوانده می شوند. البته آخر تابع هم باید به I/O Manager گزارش داده شود تا StartIo بعدی صدا شود.

متأسفانه در طول چند هفته ای که برای این فاز باقی مانده بود، بیشتر از آنچه تا اینجا درج شده موفق به پیشروی در انجام کار نشدم. چون قرار است که برنامه دیگری بالای این driver قرار گیرد و IRPهایی میان آن ئ driver رد و بدل شود، و من دسترسی به کد نمونه قدیمی آن نداشتم، قسمتهای dispatch و StartIo هنوز در driver خالی مانده و بخشهایی از کد قدیمی هنوز منتقل نشده است. گذشته از این، در طول پروسه انتقال احساس کردم که کد قدیمی هم اشکالاتی دارد یا اینکه همه آن در اختیارم نیست. در هر صورت باید توسط شخصی انجام شود که هم به کارت و عملکرد آن تسلط دارد، هم به driver قدیمی و به مباحث Windows که تا اینجا اشاره شد؛ و دانش من به مورد سوم محدود است. اگر زمان بیشتری در اختیارم بود می توانستم کار را ادامه دهم، ولی همین آشنایی با محیط سیستم عامل بیشتر وقت مرا در دوره کارآموزی به خود اشغال کرد و عملاً فقط در هفته آخر توانستم به کدنویسی بپردازم. کد نوشته شده هم خالی از اشکال نیست، تا جای ممکن سعی کرده ام که اشکالات زمان compile را رفع کنم ولی در مرحله linking نه، خود driver قدیمی به کدی اشاره دارد که موجود نیست. در بخش بعدی تلاش شده است که راه کارهایی ادامه کار پیشنهاد شود.

2-4) روند ادامه کار:

همان طور که اشاره شد، driver فعلی هنوز تا تمام شدن فاصله دارد. شخصی باید این کار را انجام دهد که غیر از تسلط روی Windows و DDK ، روی driver قدیمی و طرز کار آن نیز مسلط باشد، و وقت بیشتری در اختیار داشته باشد. مراحلی که باید طی شوند در ادامه تشریح شده اند:

1- نوشتن StartIo و Dispatchها: باید با توجه به کارایی نرم افزار لایه های بالاتر و این که چه تعاملی با driver دارند، IRPهایی که ممکن است به driver فرستاده شود را پردازش کرد. با توجه به کد driver قدیمی، این مرحله خیلی مشکل نخواهد بود .

2- تکمیل IOC.reg : این فایل که ویرایش کننده Registry است، در حال حاضر کمترین کار لازم را انجام می دهد، یعنی شناساندن IOC.sys به سیستم. در صورت نیاز، می توان آن را گسترش داد

(فرمت چنین فایلی در MSDN آمده است) یا اینکه keyهای دیگری به Registry اضافه کرد که اطلاعات بیشتری در مورد device در اختیار driver و سیستم قرار می دهد.

3- نوشتن نرم افزار لایه بالاتر: همانطور که قبلا اشاره شد ، بخشی از کارایی این driver در پردازش IRPهایی است که از برنامه های دیگر می آیند، و نسخه DOS-based آن نیز بر اساس وجود چنین نرم افزاری نوشته شده است. در محیط Windows، بسته به حساسیت کد و میزان تعامل آن با کاربر، این نرم افزار می تواند فرم یک function driver یا یک user-mode application به خود گیرد.

4- در صورت نیاز می توان یک نصب کننده هم برای driver نوشت. این نصب کننده فقط یک application کوچک است که فایل (یا فایل های) sys. را به همان %system%\drivers کپی کرده و تغییرات داخل فایل های reg. را در registry اعمال می کند. این نرم افزار ضروری نیست ولی کاربرپسندی را به شدت افزایش می دهد.

طبیعتاً در تمامی مراحل مذکور debugging نقش حیاتی دارد؛ و حیاتی تر و مشکل تر از همه خود bus driver است. از آنجایی که خطاهایی که ممکن است در زمان اجرا برای این driver پیش آید، احتمالاً در همان startup رخ می دهند یا در هر صورت به احتمال زیاد برای سیستم مشکلات اساسی به بار می آورند، شناسایی و رفع آنها کار مشکلی است. یک راه حل این است که دو سیستم را – مثلاً از طریق serial port – به هم وصل کرده خطاهای روی یکی را به دیگری بفرستیم و از آنجا کنترل کنیم.

در آخر، یک مورد مهم دیگر نیز باقی می ماند: error handling. در driver قدیمی، تنها اتفاقی که در صورت بروز خطا در سخت افزار رخ می داد، چاپ پیام مناسب روی صفحه بود. من هم در کدی که نوشتم همین شیوه را پیش گرفتم، ولی این برای یک سیستم Windows – مخصوصاً برای یک driver و مخصوصاً در مرحله debugging – اصلاً قابل قبول نیست. حداقل کاری که می توان انجام داد، error logging برای سیستم است که بتوان در Error Log (که از Control Panel فراخوانده می شود) ثبت شود. برای debugging که باید تمهیداتی دیده شود که این logها به کامپیوتر دیگری منتقل شود. برای همه این کارها، فرآیندهای از پیش تعیین شده وجود دارد؛ با این حال کارهای زمان بری هستند.

2-5) نتیجه گیری:

از آنجا که سیستم عامل های hardware-independent و فراگیر Windows همه جوانب یک دستگاه را کنترل کرده هیچ وقت (بر خلاف DOS) کنترل CPU را به یک قطعه کد واگذار نمی کنند، برای نوشتن هر برنامه ای در آنها باید روندی را که خود سیستم عامل در نظر گرفته دنبال کرد تا برنامه

ها هم درست اجرا شوند هم به سیستم لطمه ای نزنند. وقتی برنامه مورد نظر مستقیماً و از نزدیک با اجزای سیستم و سخت افزار درگیر است، سطح این حساسیت ها بسیار زیاد می شود. پس نوشتن یک driver در Windows، علاوه بر آشنایی با خود device، آشنایی زیادی با Windows نیز می طلبد. از همین جهت است که عمده زمان دوره کارآموزی من به مطالعه Windows و برنامه نویسی در آن گذشت (شخصاً تا قبل از این تجربه زیادی در برنامه نویسی سیستم نداشتم)؛ و این فاز کارآموزی دقیقاً «کارآموزی» بود نه تحویل کار تمام شده. ولی همین موضوع باعث میشود که این زمان برای من ارزش فوق العاده زیادی داشته باشد – نه تنها با اصول driver نویسی (که در این مملکت، افراد مسلط به این کار به سختی یافت می شوند) آشنا شدم، بلکه فنون دیگری از جمله برنامه نویسی در VC++ برای Windows، استفاده از MFC Libraries و اصول معماری Windows و ابزار معمول یک برنامه Windows را نیز آموختم که مختص driver نویسی نیستند و کاربرد آنها بسیار فراگیر است.

از دیدگاه تحویل کار، عملاً با توجه به قطعه کدی که در اختیار داشتم و بدون کسب اطلاعات زیادی در مورد معماری خود کارت IOC و هدف استفاده از آن (که خود پروسه زمان بری است) و صرف وقت زیادی برای debugging (که سخت ترین بخش نوشتن هر نرم افزاری است، مخصوصاً یک driver)، کار را خیلی بیش از این نمی توان به پیش برد؛ زمان محدود کارآموزی هم فرصت ورود به این مباحث را نمی دهد.

کاری که در حال حاضر ارائه می شود متشکل است از دو قسمت: یکی کاملاً تمام شده و عملاً نیاز به تغییرات ندارد؛ دیگری کاملاً خالی است، چهار چوبی که برای تکمیل driver باید (عمدتاً از روی نسخه قدیمی آن) تکمیل شود. قطعات و لایه های دیگری هم که باید به کار اضافه شوند تا به مرحله استفاده برسد نیز دز طول گزارش این گزارش معرفی شده اند.

ضمائم

ضمیمه الف – نقشه منبع تغذیه (نتیجه فاز اول):

در سه صفحه آتی، نقشه ای که نتیجه مهندسی معکوس بورد منبع بوده، درج شده است. در صفحه اول، مدارهای ورودی تا سیم پیچ اولیه ترانس؛ در صفحه دوم، مدارهای بعد از ثانویه ترانس تا خروجیهای DC و در صفحه سوم، مدارهای فیدبک و محافظت کننده ترسیم شده اند.

ضمیمه ب – کد نوشته شده برای driver:

کدی که برای driver نوشته ویا منتقل شده است، در سه فایل قرار دارد:

1) IOC.h که تعریف device extension با کلیه متغیرهای داخل آن، و prototype همه توابع اصلی در آن قرار دارد. النته لازم تیست این تعاریف در یک header file جدا بیاید، این کار برای وضوح بیشتر انجام شده است.
2) IOC.cpp که بدنه اصلی کد (implementation توابع اصلی و توابع کمکی) در آن است.
3) IOC.reg که حاوی تغییرات لازم در registry است. برای اعمال تغییرات کافی است در Windows Explorer روی آن double-click شود. البته در صورت تمایل به نوشتن یک installer، این فایل توسط خود installer اجرا شود.

در ادامه، عین کد آمده است:

```
// File: IOC.h

#pragma once

#include <ntddk.h>
#include <string.h>
#include <stdio.h>

// Object extension type defined for enhanced readability
// and use of integrated VC++ tips:
#define MAX_PATHLEN 100

typedef enum {StatusReg, RDIVect, STATUS_REG_COUNT}
INTERRUPT_STATE_INDEX;

typedef struct {
      PDEVICE_OBJECT pDeviceObject;
      WCHAR regPath[MAX_PATHLEN];
      PKINTERRUPT pInterrupt;
      UCHAR interruptStatus[STATUS_REG_COUNT];

      volatile int ioc_pktcnt;
      volatile int ioc_SFHNum;
      volatile int ioc_Invalids;
      volatile int ioc_WriteErrors;
      volatile int ioc_ReadErrors;
      volatile int ioc_UnLocks;
      unsigned short ioc_TOffset;
      unsigned char PktBuf[4000];
      int PktBufLen;
      int FrmNum;
      int SFHNum;
      int PktNum;
      unsigned long SymbolErrors;
      unsigned long FreqCounter;
      unsigned long FreqAdjIrqCntr;
      unsigned long SymbolIrqCntr;
      unsigned long InvalidFrqCntr;
      long NewGlobal;
      UCHAR Command;
      unsigned int ODLCaddresses[4];
      unsigned long  NumberOfWaitingPackets;          //81/04/04 Fallahi
      int whichbuffer;                                //81/04/04 Fallahi
      int outroutindex;
      unsigned int PacketLen[CB_WIDTH];
      unsigned char  RxPktBuf[CB_WIDTH][MAXPCKTSIZE];
      unsigned char far rxchar[bufferlen + 2];
      int inbufcnt;
      unsigned int crcerror;
      long int numberofpackets;
      long int numberoflanpackets;

      unsigned char feclentable[256];

} IOCEXTENSION, *PIOCEXTENSION;

////////////////////////////////////////////////////////////////////
// Device resource macros (interrupt vector and I/O range):

// Interrupt vector:
#define UDLCINT  5                              // 81/04/04 Fallahi
```

```
#define INTR      UDLCINT+8                    // 81/04/04 Fallahi

// Addresses:
#define  BASEADDRESS (unsigned char *) 0x800
#define  STATUSaddr     BASEADDRESS + 0x1D

#define  FIFOADD        BASEADDRESS + 0x0A      /* base address */

#define IFIDaddrL       BASEADDRESS+ 0x14       // 81/04/04 Fallahi
#define IFIDaddrM       BASEADDRESS+ 0x15       // 81/04/04 Fallahi
#define ResetFifoAddr   BASEADDRESS+ 0x18
#define RESETaddr       BASEADDRESS+ 0x1F
#define RELSEaddr       BASEADDRESS+ 0x1E
#define RDDATAaddr      BASEADDRESS+ 0x16
#define NUMOPaddr       BASEADDRESS+ 0x1C       //81/04/04 Fallahi
#define OverRunAddr     BASEADDRESS+ 0x1B       //81/04/04 Fallahi
#define nCONFIG         BASEADDRESS+ 0x40       //81/04/04 Fallahi
#define DCLK            BASEADDRESS+ 0x41       //81/04/04 Fallahi

#define  WRFIFO         FIFOADD     /* write */
#define  WRCMD          FIFOADD + 1 /* write */
#define  RESETFIFO      FIFOADD + 2 /* write */
#define  INTACK         FIFOADD + 3 /* write */
#define  TOFFLOW        FIFOADD + 4 /* write */
#define  TOFFHIGH       FIFOADD + 5 /* write */

#define  RDIVECT        FIFOADD     /* read */
#define  RDSTATUS       FIFOADD + 1 /* read */
#define  RDFRMNUM       FIFOADD + 2 /* read */

#define  ZEROS          1
#define  PZEROS         6

#define  SFHIRQ         0x0b

// Counter Addresses, Read Only
#define  FEC_RDCNT0     BASEADDRESS+0x38
#define  FEC_RDCNT1     BASEADDRESS+0x39
#define  FEC_RDCNT2     BASEADDRESS+0x3a
#define  FRQ_RDCNT0     BASEADDRESS+0x3b
#define  FRQ_RDCNT1     BASEADDRESS+0x3c
#define  FRQ_RDCNT2     BASEADDRESS+0x3d
#define  FRQ_RDCNT3     BASEADDRESS+0x3e

// Counter Clear Addresses, Write Only
#define  FEC_CLRCNT     BASEADDRESS+0x38
#define  FRQ_CLRCNT     BASEADDRESS+0x3b
#define  FRQ_REFADJ     BASEADDRESS+0x3f
#define  CLR_DEM_LOCK   BASEADDRESS+0x39   /* write only */

#define IFIDL           0x81              //81/04/04 Fallahi
#define IFIDM           0x1A              //81/04/04 Fallahi
#define RESETaddr       BASEADDRESS+ 0x1F //81/04/04 Fallahi
#define CNTRLaddr       BASEADDRESS+ 0x19 //81/04/04 Fallahi
#define CNTRLREGISTER   0x0               //81/04/04 Fallahi

// Other macros:
#define RBF_FileName    "ioc_fpga.cfg"    // 81/04/04 Fallahi
#define CB_WIDTH 160                      // 81/04/04 Fallahi
#define MAXPCKTSIZE  256                  // 81/04/04 Fallahi
#define bufferlen 10000                   // 81/04/04 Fallahi
```

```
#define FifoLen 0x8192                    // 81/04/04 Fallahi
#define IO_ERR_HARDWARE_INIT 0xF000FFFF   // 84/06/12 Moghimi - NTSTATUS

////////////////////////////////////////////////////////////////
// Prototypes of backbone functions, implemented and explained in IOC.cpp

NTSTATUS DriverEntry (IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING
RegistryPath);

NTSTATUS IocDispatchRead  (IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp);

NTSTATUS IocDispatchWrite (IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp);

NTSTATUS IocDispatchDeviceControl (IN PDEVICE_OBJECT DeviceObject, IN
PIRP Irp);

NTSTATUS IocDispatchCreate (IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp);

NTSTATUS IocDispatchClose  (IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp);

VOID IocStartIo (IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp);

BOOLEAN  IocInterruptService(IN struct _KINTERRUPT *Interrupt, IN PVOID
ServiceContext);

VOID IocDpcForIsr(IN PKDPC Dpc, IN struct _DEVICE_OBJECT *DeviceObject,
IN struct _IRP *Irp, IN PVOID Context);
```

```
// File: IOC.cpp

#include "ioc.h"


////////////////////////////////////////////////////////////////////////

// Dummy functions, may be used later to handle PnP, power and WMI IRPs:

NTSTATUS IocAddDevice (IN PDRIVER_OBJECT DriverObject, IN PDEVICE_OBJECT
PhysicalDeviceObject)
{ return STATUS_SUCCESS; }

VOID IocUnload (IN PDRIVER_OBJECT DriverObject)  { }

NTSTATUS IocDispatchSystemControl(IN PDEVICE_OBJECT  DeviceObject, IN
PIRP Irp)
{ return STATUS_SUCCESS; }

////////////////////////////////////////////////////////////////////////
// Helper functions:

void SendError(char* action, NTSTATUS errorType)
{
// This is temporary and weak error handling, and should be replaced
// with actual error logging (at the very minimum)
      printf("Error in IOC driver while %s, NTSTATUS code %8X (%d
decimal)!", action, errorType, errorType);
      printf("\nPress ENTER to continue...");
      getchar();
}

void CopyData(void *dest, void *src, unsigned int srcSize)
{
      dest = (unsigned char *)dest;
      src = (unsigned char *)src;
      for (unsigned int i = 0; i < srcSize; i++)
            dest[i] = src[i];
}

void InitContextInfo(PIOCEXTENSION iocExtension)
{
      iocExtension->ioc_pktcnt = 0;  // Initialize counters in extension
      iocExtension->ioc_SFHNum = 0;
      iocExtension->ioc_Invalids = 0;
      iocExtension->ioc_WriteErrors = 0;
      iocExtension->ioc_ReadErrors = 0;
      iocExtension->ioc_UnLocks = 0;
      iocExtension->ioc_TOffset = 0;
      iocExtension->PktBufLen = 0;
      iocExtension->FrmNum = 0;
      iocExtension->SFHNum = 0;
      iocExtension->PktNum = 0;
      iocExtension->SymbolErrors = 0;
      iocExtension->FreqCounter = 0;
      iocExtension->FreqAdjIrqCntr = 0;
      iocExtension->SymbolIrqCntr = 0;
      iocExtension->InvalidFrqCntr = 0;
      iocExtension->NewGlobal = 0;
      iocExtension->Command = 0;
```

```
	unsigned int tempODLC[4] = {0x0,0xffff,0x35f,0xffff};
	CopyData(iocExtension->ODLCaddresses, tempODLC, 4*sizeof(unsigned
int));
	iocExtension->NumberOfWaitingPackets = 0;
	iocExtension->outroutindex = 0;
	iocExtension->crcerror = 0;
	iocExtension->numberofpackets = 0;
	iocExtension->numberoflanpackets = 0;

	unsigned char tempfeclen[256] = {
	0x00, 0x39, 0x72, 0x4b, 0xe4, 0xdd, 0x96, 0xaf, 0xf1, 0xc8, 0x83,
0xba, 0x15, 0x2c, 0x67, 0x5e, 0xdb, 0xe2, 0xa9, 0x90, 0x3f, 0x06, 0x4d,
0x74, 0x2a, 0x13, 0x58, 0x61, 0xce, 0xf7, 0xbc, 0x85, 0x8f, 0xb6, 0xfd,
0xc4, 0x6b, 0x52 ,0x19, 0x20, 0x7e, 0x47, 0x0c, 0x35, 0x9a, 0xa3, 0xe8,
0xd1, 0x54, 0x6d, 0x26, 0x1f, 0xb0, 0x89, 0xc2, 0xfb, 0xa5, 0x9c, 0xd7,
0xee, 0x41, 0x78, 0x33, 0x0a, 0x27, 0x1e, 0x55, 0x6c, 0xc3, 0xfa, 0xb1,
0x88, 0xd6, 0xef, 0xa4, 0x9d, 0x32, 0x0b, 0x40, 0x79, 0xfc, 0xc5, 0x8e,
0xb7, 0x18, 0x21, 0x6a, 0x53, 0x0d, 0x34, 0x7f, 0x46, 0xe9, 0xd0, 0x9b,
0xa2, 0xa8, 0x91, 0xda, 0xe3, 0x4c, 0x75, 0x3e, 0x07, 0x59, 0x60, 0x2b,
0x12, 0xbd, 0x84, 0xcf, 0xf6, 0x73, 0x4a, 0x01, 0x38, 0x97, 0xae, 0xe5,
0xdc, 0x82, 0xbb, 0xf0, 0xc9, 0x66, 0x5f, 0x14, 0x2d, 0x4e, 0x77, 0x3c,
0x05, 0xaa, 0x93, 0xd8, 0xe1, 0xbf, 0x86, 0xcd, 0xf4, 0x5b, 0x62, 0x29,
0x10, 0x95, 0xac, 0xe7, 0xde, 0x71, 0x48, 0x03, 0x3a, 0x64, 0x5d, 0x16,
0x2f, 0x80, 0xb9, 0xf2, 0xcb, 0xc1, 0xf8, 0xb3, 0x8a, 0x25, 0x1c, 0x57,
0x6e, 0x30, 0x09, 0x42, 0x7b, 0xd4, 0xed, 0xa6, 0x9f, 0x1a, 0x23, 0x68,
0x51, 0xfe, 0xc7, 0x8c, 0xb5, 0xeb, 0xd2, 0x99 ,0xa0, 0x0f, 0x36, 0x7d,
0x44, 0x69, 0x50, 0x1b, 0x22, 0x8d, 0xb4, 0xff, 0xc6, 0x98, 0xa1, 0xea,
0xd3, 0x7c, 0x45, 0x0e, 0x37, 0xb2, 0x8b, 0xc0, 0xf9, 0x56, 0x6f, 0x24,
0x1d, 0x43, 0x7a, 0x31, 0x08, 0xa7, 0x9e, 0xd5, 0xec, 0xe6, 0xdf, 0x94,
0xad, 0x02, 0x3b, 0x70, 0x49, 0x17, 0x2e, 0x65, 0x5c, 0xf3, 0xca, 0x81,
0xb8, 0x3d, 0x04, 0x4f, 0x76, 0xd9, 0xe0, 0xab, 0x92, 0xcc, 0xf5, 0xbe,
0x87, 0x28, 0x11, 0x5a, 0x63};
	CopyData(iocExtension->feclentable, tempfeclen, 256*sizeof(unsigned
char));

}

NTSTATUS ProgramFPGA(void)
{
	long y;
	int i,k;
	FILE *in;
	unsigned char ch;
	char VersionStr[50];

	printf("\nProgramming the IOC FPGA");

	if ((in = fopen(RBF_FileName, "rbf")) == NULL)
	{
		printf("\nRBF File Not found.\n\n");
		return IO_ERR_HARDWARE_INIT;
	}

	fread( VersionStr, 1, 50, in );
	theDiuConfig.m_SetIOCVersionStr( VersionStr );
	WRITE_PORT_UCHAR(nCONFIG,1);
	delay(100);
	for(y=0;y<62283L;y++)
	{
		ch = fgetc(in);
		for(k=0; k<8; k++)
```

```
            {
                  WRITE_PORT_UCHAR(DCLK,ch);
            }
      }

      for(k=1; k<=10; k++)
      {
            for(i=1; i<=18; i++)
            {
                  WRITE_PORT_UCHAR(DCLK,1);
            }
       }

      fclose(in);
      delay(100);
      WRITE_PORT_UCHAR(RESETaddr,0);
      return STATUS_SUCCESS;
}

void IOCSetTOffset( short T_Offset, PIOCEXTENSION pContext)
{
// Reset T_Offset
   pContext->ioc_TOffset = T_Offset;
   WRITE_PORT_UCHAR( TOFFLOW, pContext->ioc_TOffset & 0xff );
   WRITE_PORT_UCHAR( TOFFHIGH, (pContext->ioc_TOffset >> 8) & 0xff );
}

void IOCHoldClock(PIOCEXTENSION pContext)
{
// Hold IOC Clock
   pContext->Command |= 0x0c;
   WRITE_PORT_UCHAR( WRCMD, pContext->Command );
// hold clock & disable fifo read by IOC control
}

void IOCReleaseClock(PIOCEXTENSION pContext)
{
// Release IOC Clock
   pContext->Command &= 0xF0;
   WRITE_PORT_UCHAR( WRCMD, 0 );    // release clock
}

void IOCReset( int SetNoPacket, PIOCEXTENSION pContext)
{
// Hold Clock, reset IOC Control & FIFO, Release Clock.
// it Writes cmd 08 into FIFO if SetNoPacket is true
   int i;
// Hold IOC Clock & Disable fifo read
   IOCHoldClock(pContext);
   WRITE_PORT_UCHAR( RESETFIFO, 0 );            // FIFO reset
   if( SetNoPacket )
   {
   // write no packet command in FIFO
       IOCWriteFIFO( 0 );
       IOCWriteFIFO( 0x08 );
   }
// clear interrupt vector
   WRITE_PORT_UCHAR( INTACK, 0 );
// start IOC clock
   IOCReleaseClock(pContext);
// IOC control reset
```

```
   pContext->Command |= 0x02;
   WRITE_PORT_UCHAR( WRCMD, pContext->Command );
   delay( 1 );
//   for( i = 0; i < 1000; i++ );
   pContext->Command &= 0xFD;
   WRITE_PORT_UCHAR( WRCMD, pContext->Command );
// clear reset

   WRITE_PORT_UCHAR( FEC_CLRCNT, 0 );  // clear symbol error rate counter
   WRITE_PORT_UCHAR( FRQ_CLRCNT, 0 );  // clear freq. adjust counter
}

void IOCInit( short T_Offset, PIOCEXTENSION pContext)
{
// Initiate IOC Control, set ISR, set T_Offset, reset FIFO & start clock
// cmd 08 will be writen into FIFO
   IOCHoldClock(pContext);        // Hold IOC Clock & Disable fifo read
   IOCSetTOffset( T_Offset, pContext);      // set T_Offset value
   IOCReset( 1, pContext);         // Restart IOC & write cmd 08 in FIFO
}

void InitializeUDLCC(PIOCEXTENSION pContext)          // 81/04/04 Fallahi
{
      int y;
      extern unsigned char cntrlreg;

      WRITE_PORT_UCHAR(IFIDaddrL,IFIDL); // for IFID
      WRITE_PORT_UCHAR(IFIDaddrM,IFIDM);
      WRITE_PORT_UCHAR(RESETaddr,0); // reset card and Hold it in RESET
      WRITE_PORT_UCHAR(CNTRLaddr,CNTRLREGISTER); // Control register
      pContext->ODLCaddresses[0] = theDiuConfig.m_GetCardAddress();
      for(y=0;y<4;y++)       // writing ODLC addresses in UDLCC
      {
            WRITE_PORT_UCHAR(BASEADDRESS+2*y  ,pContext-
            >ODLCaddresses[y]>>8);
            WRITE_PORT_UCHAR(BASEADDRESS+2*y+1,pContext-
            >ODLCaddresses[y]&0xFF);
      }

      IOCInit( theDiuConfig.m_GetTimingOffset() , pContext);
      WRITE_PORT_UCHAR(RELSEaddr,0); // release card
      WRITE_PORT_UCHAR(CNTRLaddr,cntrlreg);
}

NTSTATUS InitHardware(PIOCEXTENSION context)
{
      NTSTATUS returnStatus = ProgramFPGA();
      if (NT_SUCCESS(returnStatus)) InitializeUDLCC(pContext);
      return returnStatus;
}

int IOCGetFrmnum(void)
{// reads and returns current frame number
      return READ_PORT_UCHAR( RDFRMNUM );
}

void IOCWriteFIFO( char data )
{
// Write a byte in FIFO
   WRITE_PORT_UCHAR( WRFIFO, data );
}
```

```
void SymbolErrIRQ_Handler(PIOCEXTENSION pContext)
{
	long t = 0;
	t |= (long)READ_PORT_UCHAR( FEC_RDCNT0 );
	t |= ((long)READ_PORT_UCHAR( FEC_RDCNT1 ) << 8 );
	t |= ((long)READ_PORT_UCHAR( FEC_RDCNT2 ) << 16 );
	pContext->SymbolErrors = t;
	pContext->SymbolIrqCntr++;
	theDiuConfig.m_SetSER( (double)pContext->SymbolErrors/2097152.0);
// reset counter
	WRITE_PORT_UCHAR( FEC_CLRCNT, 0 );
}

void FreqAdjustIRQ_Handler(PIOCEXTENSION pContext)
{
	long t = 0, i, ii, iii, c;

	i = -1;
	ii = -1;
	c = 0;
	do
	{
		c++;
		iii = ii;
		ii = i;
		i = (long)READ_PORT_UCHAR( FRQ_RDCNT0 );
	}while( (i != ii || ii != iii) && c < 10 );
	if( c == 10 )
	{
		//InvalidFrqCntr++; Namazi& Zahedi 82/03/05
		WRITE_PORT_UCHAR( FRQ_CLRCNT, 0 );
		return;
	}
	t |= i;

	i = -1;
	ii = -1;
	c = 0;
	do
	{
		c++;
		iii = ii;
		ii = i;
		i = (long)READ_PORT_UCHAR( FRQ_RDCNT1 );
	}while( (i != ii || ii != iii) && c < 10 );
	if( c == 10 )
	{
		//InvalidFrqCntr++; Namazi& Zahedi 82/03/05
		WRITE_PORT_UCHAR( FRQ_CLRCNT, 0 );
		return;
	}
	t |= i << 8;

	i = -1;
	ii = -1;
	c = 0;
	do
	{
		c++;
		iii = ii;
```

```
            ii = i;
            i = (long)READ_PORT_UCHAR( FRQ_RDCNT2 );
      }while( (i != ii || ii != iii) && c < 10 );
      if( c == 10 )
      {
            //InvalidFrqCntr++; Namazi& Zahedi 82/03/05
            WRITE_PORT_UCHAR( FRQ_CLRCNT, 0 );
            return;
      }
      t |= i << 16;

      i = -1;
      ii = -1;
      c = 0;
      do
      {
            c++;
            iii = ii;
            ii = i;
            i = (long)READ_PORT_UCHAR( FRQ_RDCNT3 );
      }while( (i != ii || ii != iii) && c < 10 );
      if( c == 10 )
      {
            //InvalidFrqCntr++; Namazi& Zahedi 82/03/05
            WRITE_PORT_UCHAR( FRQ_CLRCNT, 0 );
            return;
      }
      t |= i << 24;

      /*
      t |= (long)READ_PORT_UCHAR( FRQ_RDCNT0 );
      t |= ((long)READ_PORT_UCHAR( FRQ_RDCNT1 ) << 8 );
      t |= ((long)READ_PORT_UCHAR( FRQ_RDCNT2 ) << 16 );
      t |= ((long)READ_PORT_UCHAR( FRQ_RDCNT3 ) << 24 );
      */
      pContext->FreqCounter = t;
      pContext->FreqAdjIrqCntr++;

// reset counter
      WRITE_PORT_UCHAR( FRQ_CLRCNT, 0 );
}

//////////////////////////////////////////////////////////////////////

// The DriverEntry function is called during system startup, initializes
// driver object, device and device object.  Also registers interrupt.

NTSTATUS DriverEntry (IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING
RegistryPath)
{
      NTSTATUS status;  // Used for status returns and error logging.

      PDEVICE_OBJECT iocObject;           // Create device object
      UNICODE_STRING deviceName;
// Initializing string name in most ridiculous way possible: giving
//straight unicode values.
      // Device Object name:  "\Device\IOC"
      WCHAR tempWideName[12] = {92, 68, 101, 118, 105, 99, 101, 92, 73,
79, 68, 0};
      RtlInitUnicodeString(&deviceName, tempWideName);
```

```
	status = IoCreateDevice(DriverObject, sizeof(IOCEXTENSION),
&deviceName, FILE_DEVICE_PHYSICAL_NETCARD, FILE_DEVICE_SECURE_OPEN,
FALSE, /*"D:P(A;;GA;;;SY)(A;;GRGWGX;;;BA)(A;;GR;;;WD)", {03739022-A57A-
496e-BB1F-1E2A320E2D29},*/ &iocObject);
	if (!NT_SUCCESS(status))
	{
		SendError("creating device object", status);
		return status;
	}

	PIOCEXTENSION iocExtension = (PIOCEXTENSION)(iocObject-
>DeviceExtension);	// Allocates device object extension, used for
//context-safe data storage.
	wcsncpy(iocExtension->regPath, RegistryPath->Buffer, MAX_PATHLEN);
// Saves registry path in the extension (see header file for extension
//definition).
	iocExtension->pDeviceObject = iocObject;  // Save address of Device
//Object in extension.
	InitContextInfo(iocExtension);	// Initialize global vars in
//extension.

	DriverObject->DriverExtension->AddDevice = IocAddDevice;
// Export entry points
	DriverObject->MajorFunction[IRP_MJ_READ] = IocDispatchRead;
	DriverObject->MajorFunction[IRP_MJ_WRITE] = IocDispatchWrite;
	DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] =
IocDispatchDeviceControl;
	DriverObject->MajorFunction[IRP_MJ_CREATE] = IocDispatchCreate;
	DriverObject->MajorFunction[IRP_MJ_CLOSE] = IocDispatchClose;
	DriverObject->MajorFunction[IRP_MJ_SYSTEM_CONTROL] =
IocDispatchSystemControl;
	DriverObject->MajorFunction[IRP_MJ_PNP] = NULL;
	DriverObject->MajorFunction[IRP_MJ_POWER] = NULL;
	DriverObject->DriverStartIo = IocStartIo;
	DriverObject->DriverUnload = IocUnload;

	// Connect interrupt and register DpcForIsr routine:
	status = IoConnectInterrupt(&(iocExtension->pInterrupt),
IocInterruptService, iocExtension, NULL, INTR, 1, 1, Latched, FALSE,
KeQueryActiveProcessors(), FALSE);
	if (!NT_SUCCESS(status))
	{
		SendError("connecting interrupt service routine", status);
		return status;
	}
	IoInitializeDpcRequest(iocObject, IocDpcForIsr);

	status = InitHardware(iocExtension);	// Initialize Hardware.
	if (!NT_SUCCESS(status))
	{
		SendError("initializing hardware", status);
		return status;
	}

	return STATUS_SUCCESS;
}

////////////////////////////////////////////////////////////////

// The following three functions respond to Read/Write or control
// IRPs, dispatch necessary operations for processing in IocStartIo
```

```
NTSTATUS IocDispatchRead  (IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
{
      return STATUS_SUCCESS;
}

NTSTATUS IocDispatchWrite (IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
{
      return STATUS_SUCCESS;
}

NTSTATUS IocDispatchDeviceControl (IN PDEVICE_OBJECT DeviceObject, IN
PIRP Irp)
{
      return STATUS_SUCCESS;
}

////////////////////////////////////////////////////////////////////

// IocStartIo performs bulk of IRP processing

VOID IocStartIo (IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
{
      return;
}

////////////////////////////////////////////////////////////////////

// Following are the ISR functions.  IocInterruptService initially
// handles interrupts, then dispatches DPC calls to IocDpcForIsr
// for extra processing functionality at lower IRQL.
// The NewSFH function is a helper function of the DPC routine, coded
// here instead of in the helper functions section for readability.

BOOLEAN  IocInterruptService(IN struct _KINTERRUPT *Interrupt, IN PVOID
ServiceContext)
{
      PIOCEXTENSION pContext = (PIOCEXTENSION)ServiceContext;
      pContext->interruptStatus[StatusReg] = READ_PORT_UCHAR(STATUSaddr);
      pContext->interruptStatus[RDIVect] = READ_PORT_UCHAR(RDIVECT);

      IoRequestDpc(pContext->pDeviceObject, Interrupt, pContext);
      // Queue DPC
      return TRUE;
}

int NewSFH(PIOCEXTENSION pContext)
{
// IOC Interrupt Handler for SFH & Errors
// returns 1 if any interrupt cause is detected or returns 0
   int i;
   bool flg = false;
   short ivct;
   char *pPkt;
   static int PktLen = 0;

   pContext->NewGlobal++;
   ivct = pContext->interruptStatus[RDIVect];   // read interrupt vector
   if( ivct & 0xff )
   {
      if( ivct & 2 )
```

```
        {   // invalid command detected. reset FIFO & CIU
            flg = true;
            pContext->ioc_Invalids++;
        }
        if( ivct & 4 )
        {   // FIFO write error
            flg = true;
            pContext->ioc_WriteErrors++;
        }
        if( ivct & 8 )
        {   // FIFO Read error
            flg = true;
            pContext->ioc_ReadErrors++;
          }

          if( ivct & 0x10 )
          {
                flg = true;
                pContext->ioc_UnLocks++;
          }

        if( ivct & 0x20 )
        {   // SymbolErrIRQ:
            SymbolErrIRQ_Handler(pContext);
        }
        if( ivct & 0x40 )
        {   // Freq. Adjust IRQ:
            FreqAdjustIRQ_Handler(pContext);
        }

        if( ivct & 1 )
        {   // new SFH or frame_clk detected
            if( !IOCGetFrmnum() )
                pContext->ioc_SFHNum++;
            pPkt = IOCWantPacket( &PktLen);
//          pPkt = &txbuf;
            if( pPkt )
                {   // copy packet to FIFO
                for( i = 0; i < PktLen; i++, pPkt++ )
              {
                    IOCWriteFIFO( *pPkt);
                    pContext->PktBuf[i] = *pPkt;
                }
                pContext->ioc_pktcnt++;
                pContext->FrmNum = IOCGetFrmnum();
                pContext->SFHNum = pContext->ioc_SFHNum;
                pContext->PktNum = pContext->ioc_pktcnt;
                pContext->PktBufLen = PktLen;
            }
            else
            {   // no packet command
                IOCWriteFIFO( 0 );
                IOCWriteFIFO( 0x08 );
//              for( i = 0; i < PZEROS; i++ )
//                 IOCWriteFIFO( 0 );
            }
        }

        if( flg )
            IOCError( ivct >> 1 );
```

```
      WRITE_PORT_UCHAR( INTACK, 0 );      // clear interrupt vector
      return 1;
   }
   return 0;
}

VOID IocDpcForIsr(IN PKDPC Dpc, IN struct _DEVICE_OBJECT *DeviceObject,
IN struct _IRP *Irp, IN PVOID pContext)
{
      pContext = (PIOCEXTENSION)pContext;
      int i,j;
      unsigned char Packetstatus;
      int   tempbufferpointer;
      unsigned char numofpacketsinfifo;
      //Packet status: MSB :  crcerr,q3,0,0,0,crcerr,crcerr,crcerr
      unsigned int odlcaddress;
      UCHAR ARQ_M;
      UCHAR adr1,adr2;
      UCHAR fec;
      UCHAR len;
      UCHAR CardStatus;
      WORD ReadCount;
      int Continue;

  // acknowledge the interrupt     (shall deactivate the interrupt pin)
  // Read Status and determine the number of Data in fifo.

      do
      {
      Continue = 0;

      if(pContext->interruptStatus[StatusReg] & 0x20)
      {
            Continue = 1;
            numofpacketsinfifo = READ_PORT_UCHAR(NUMOPaddr);
            pContext->numberofpackets += numofpacketsinfifo;

            for(i=0;i<numofpacketsinfifo;i++)
            {
                  if (!(READ_PORT_UCHAR(STATUSaddr)&0x40))
                  {
                        WRITE_PORT_UCHAR(RESETaddr,0);
                        WRITE_PORT_UCHAR(ResetFifoAddr,0);
                        WRITE_PORT_UCHAR(RELSEaddr,0);
                        goto emptyerror;
                  }
                  pContext->inbufcnt=0;
                  pContext->RxPktBuf[pContext->whichbuffer][pContext-
>inbufcnt++]=adr1=READ_PORT_UCHAR(RDDATAaddr);
                  if (!(READ_PORT_UCHAR(STATUSaddr)&0x40))
                  {
                        WRITE_PORT_UCHAR(RESETaddr,0);
                        WRITE_PORT_UCHAR(ResetFifoAddr,0);
                        WRITE_PORT_UCHAR(RELSEaddr,0);
                        goto emptyerror;
                  }

                  pContext->RxPktBuf[pContext->whichbuffer][pContext-
>inbufcnt++]=adr2=READ_PORT_UCHAR(RDDATAaddr);
                  if (!(READ_PORT_UCHAR(STATUSaddr)&0x40))
                  {
```

```
                        WRITE_PORT_UCHAR(RESETaddr,0);
                        WRITE_PORT_UCHAR(ResetFifoAddr,0);
                        WRITE_PORT_UCHAR(RELSEaddr,0);
                        goto emptyerror;
                  }
                  pContext->RxPktBuf[pContext->whichbuffer][pContext-
>inbufcnt++]=fec=READ_PORT_UCHAR(RDDATAaddr);  // FEC
                  if (!(READ_PORT_UCHAR(STATUSaddr)&0x40))
                  {
                        WRITE_PORT_UCHAR(RESETaddr,0);
                        WRITE_PORT_UCHAR(ResetFifoAddr,0);
                        WRITE_PORT_UCHAR(RELSEaddr,0);
                        goto emptyerror;
                  }
                  pContext->RxPktBuf[pContext->whichbuffer][pContext-
>inbufcnt++]=len=READ_PORT_UCHAR(RDDATAaddr);  // LENGTH
                  CardStatus=READ_PORT_UCHAR(STATUSaddr);
                  odlcaddress=adr1+adr2*256;
                  ReadCount = 0;
      // Discarding incomplete packets (all of them are 4 bytes)
                  while ( (CardStatus&0x40) && (fec!=pContext-
>feclentable[len]) && (ReadCount<FifoLen))  // FEC LEN ERROR
                  {
                        adr1=adr2; pContext->RxPktBuf[pContext-
>whichbuffer][0]=adr1;
                        adr2=fec;  pContext->RxPktBuf[pContext-
>whichbuffer][1]=adr2;
                        fec =len;  pContext->RxPktBuf[pContext-
>whichbuffer][2]=fec;  // FEC
                        len =READ_PORT_UCHAR(RDDATAaddr); pContext-
>RxPktBuf[pContext->whichbuffer][3]=len; // LENGTH
                        CardStatus=READ_PORT_UCHAR(STATUSaddr);
                        odlcaddress=adr1+adr2*256;
                        ReadCount++;
                  }

                  if( ((ReadCount>=FifoLen) ||(!(CardStatus&0x40))) )
                  {
                        WRITE_PORT_UCHAR(RESETaddr,0);
                        WRITE_PORT_UCHAR(ResetFifoAddr,0);
                        WRITE_PORT_UCHAR(RELSEaddr,0);
                        goto emptyerror;
                  }                    // EF==0 :> fifo empty

                  odlcaddress=adr1+(adr2*256);

//..................... VSAT packets received
.............................................
                  if (!(READ_PORT_UCHAR(STATUSaddr)&0x40))
                  {
                        WRITE_PORT_UCHAR(RESETaddr,0);
                        WRITE_PORT_UCHAR(ResetFifoAddr,0);
                        WRITE_PORT_UCHAR(RELSEaddr,0);
                        goto emptyerror;
                  }
                  pContext->RxPktBuf[pContext->whichbuffer][pContext-
>inbufcnt++]=READ_PORT_UCHAR(RDDATAaddr);
                  if (!(READ_PORT_UCHAR(STATUSaddr)&0x40))
                  {
                        WRITE_PORT_UCHAR(RESETaddr,0);
                        WRITE_PORT_UCHAR(ResetFifoAddr,0);
```

```
                        WRITE_PORT_UCHAR(RELSEaddr,0);
                        goto emptyerror;
                  }
                  pContext->RxPktBuf[pContext->whichbuffer][pContext-
>inbufcnt++]=READ_PORT_UCHAR(RDDATAaddr);
                  if (!(READ_PORT_UCHAR(STATUSaddr)&0x40))
                  {
                        WRITE_PORT_UCHAR(RESETaddr,0);
                        WRITE_PORT_UCHAR(ResetFifoAddr,0);
                        WRITE_PORT_UCHAR(RELSEaddr,0);
                        goto emptyerror;
                  }
                  pContext->RxPktBuf[pContext->whichbuffer][pContext-
>inbufcnt++]=READ_PORT_UCHAR(RDDATAaddr);
                  if (!(READ_PORT_UCHAR(STATUSaddr)&0x40))
                  {
                        WRITE_PORT_UCHAR(RESETaddr,0);
                        WRITE_PORT_UCHAR(ResetFifoAddr,0);
                        WRITE_PORT_UCHAR(RELSEaddr,0);
                        goto emptyerror;
                  }
                  pContext->RxPktBuf[pContext->whichbuffer][pContext-
>inbufcnt++]=ARQ_M=READ_PORT_UCHAR(RDDATAaddr);
                  tempbufferpointer=pContext->inbufcnt;
                  for(j=0;j<len-9;j++)
                  {
                        if (!(READ_PORT_UCHAR(STATUSaddr)&0x40))
                        {
                              pContext->inbufcnt=tempbufferpointer;
                              WRITE_PORT_UCHAR(RESETaddr,0);
                              WRITE_PORT_UCHAR(ResetFifoAddr,0);
                              WRITE_PORT_UCHAR(RELSEaddr,0);
                              goto emptyerror;
                        }
                        ARQ_M = pContext->RxPktBuf[pContext-
>whichbuffer][pContext->inbufcnt++]=READ_PORT_UCHAR(RDDATAaddr);
                        if (odlcaddress == 0x35f)
                        {
                              switch(j)
                              {
                                    case 4:
                                                NetID = ARQ_M;
                                                break;
                                    case 5:
                                                NetID += (ARQ_M << 8);
                              }
                        }
                  }

                  if (!(READ_PORT_UCHAR(STATUSaddr)&0x40))
                  {
                        pContext->inbufcnt=tempbufferpointer;
READ_PORT_UCHAR                     WRITE_PORT_UCHAR(RESETaddr,0);
                        WRITE_PORT_UCHAR(ResetFifoAddr,0);
                        WRITE_PORT_UCHAR(RELSEaddr,0);
                        goto emptyerror;
                  }
                  pContext->RxPktBuf[pContext->whichbuffer][pContext-
>inbufcnt++]=Packetstatus =READ_PORT_UCHAR(RDDATAaddr);
                  if((Packetstatus&0x80)!=0x80)
//          if((Packetstatus&0x87)!=0x87)   // Packet has no errors
```

```
                  {
                        pContext->PacketLen[pContext->whichbuffer]
=pContext->inbufcnt;
                        pContext->whichbuffer++;if (pContext-
>whichbuffer==CB_WIDTH) pContext->whichbuffer=0;
                        pContext->NumberOfWaitingPackets++;
                  }

                  else                          // CRC error occured
                  {
                     pContext->crcerror++;
                     pContext->whichbuffer++;if (pContext-
>whichbuffer==CB_WIDTH) pContext->whichbuffer=0;
                     pContext->NumberOfWaitingPackets++;
                  }
            }

emptyerror:
            if (READ_PORT_UCHAR(OverRunAddr))
            {
                  WRITE_PORT_UCHAR(ResetFifoAddr,0);
            }
      }

      Continue = NewSFH(pContext);
      }while( Continue );

// Interrupt servicing complete, start next packet and complete this one.
      IoStartNextPacket(DeviceObject, FALSE);
      IoCompleteRequest(Irp, IO_NO_INCREMENT);
      return;
}

//////////////////////////////////////////////////////////////////////

// The two dispatch functions below handle file object openings and
//closings by higher-level drivers (if any) and user-mode subsystems.

NTSTATUS IocDispatchCreate (IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
{
      return STATUS_SUCCESS;
}

NTSTATUS IocDispatchClose  (IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
{
      return STATUS_SUCCESS;
}

//////////////////////////////////////////////////////////////////////
```

```
// File: IOC.reg
// For details of registry editor file format, see MSDN

REGEDIT4
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Damn]
"Type"=dword:00000001
"ErrorControl"=dword:00000001
"Start"=dword:00000002
"DisplayName"="IOC"
"ImagePath"="system32\DRIVERS\IOC.sys"
```

مراجع و منابع:

1. Datasheet of the 2SA684 from Panasonic
2. Datasheet of the 2SK2648 from Fuji Electronics
3. Datasheet of the 2PA1015 from Discrete Semiconductors
4. Datasheet of the AS432 from Astec Semiconductor
5. Datasheets of the BT169, BYQ28X and LM339 from Phillips Semiconductors
6. Datasheet of the 2SC1815 from Toshiba
7. Datasheet of the CEP603AL from CEL
8. Datasheets of the CNX82A and KA431A from Fairchild Semiconductor
9. Datasheet of the KA3842A from WS
10. Datasheet of the STPS20H100CT from STMicroelectronics
11. Datasheet of the TL3834 from Texas Instruments.
12. Pietrek, Matt; *Windows 95 System Programming Secrets*; IDG Books Worldwide, Inc.; USA; Second Printing, 1996
13. *MSDN Library – April 2004* from Microsoft Corporation; particularly the sections *Platform SDK* and *DDK Documentation*.