بنام خداوند جان آفرین حکیم سخن در زبان آفرین

Crack

و

تکنیک‌های نفوذ به نرم‌افزار

تألیف:

مهندس امید غلامی

مهندس سید بهزاد لاجوردی

انتشارات ناقوس

غلامی، امید

Crack [کرک] و تکنیکهای نفوذ به نرم افزار/ تألیف امید غلامی، بهزاد لاجوردی.— تهران: ناقوس، ۱۳۸۴.

۷۵۲ ص.

ISBN : 964-377-164-4 : ۷۰۰۰۰ ریال

فهرستنویسی بر اساس اطلاعات فیپا.

۱. کامپیوترها -- ایمنی اطلاعات. ۲. شبکه های کامپیوتری -- قدامات تأمینی. ۳. حفاظت اطلاعات. الف. لاجوردی، بهزاد. ب. عنوان.

QA۷۶/۹/الف۹غ۸ ۰۰۵/۸

کتابخانهٔ ملی ایران ۸۴-۲۱۹۳۹م

انتشارات ناقوس

www.naghoospress.ir

نام کتاب	:	Crack و تکنیکهای نفوذ به نرم افزار
ناشر	:	انتشارات ناقوس
تألیف	:	امید غلامی- بهزاد لاجوردی
چاپ اول	:	۱۳۸۴
تیراژ	:	۲۰۰۰ جلد
لیتوگرافی	:	شاد
چاپ	:	سعدی
صحافی	:	پژواک اندیشه
طرح روی جلد	:	میترا نصری
قیمت	:	۷۰۰۰۰ ریال
شابک	:	۹۶۴-۳۷۷-۱۶۴-۴
ISBN	:	964-377-164-4

چاپ اول

مرکز پخش: انتشارات ناقوس

تهران: خیابان انقلاب- خیابان ۱۲ فروردین- کوچهٔ نوروز- پلاک ۲۷

تلفن: ۶۶۹۶۶۷۴۹- ۶۶۴۰۶۸۳۴- ۶۶۴۱۱۷۱۵

به نام خدا

حمد وسپاس خدایی را که از تلألو لطف و رحمتش، قلوب آدمیان شایسته تابش انوار حکمت گردید تا در اقیانوس کبیر عالم وجود، در جهت به چنگ آوردن گوهرهای علوم و کمالات شناور گردد و با نوشیدن از آب حیات معرفت، از گم‌گشتگی در وادی حیرت برحذر باشد.

این مجموعه را به پدر و مادر بزرگوارم و همه عزیزانی که جز با مساعدت و همکاری آنان این اثر پدید نمی‌آمد، تقدیم می‌کنم و امید آن دارم که این برگ سبز، مورد قبول همه دانش پژوهان، اساتید و دانشجویان رشته مهندسی کامپیوتر قرارگیرد.

زندگی صحنه زیبای هنرمندی ماست

هرکسی نغمه خود خواند و از صحنه رود

صحنه پیوسته بجاست

خرم آن نغمه که مردم بسپارند به یاد

سید بهزاد لاجوردی

تابستان ۸٤

به نام خدا

بدون تردید آنچه در دنیای امروز می‌گذرد، مدیون سعی و تلاش شرکت‌ها، گروه‌ها و افرادی است که بی‌وقفه در جهت اعتلای دانش انفورماتیک و اطلاعات گام برمی‌دارند. هدف اصلی از تألیف این کتاب، آشنایی این قشر با تکنیک‌ها و روش‌هایی است که توسط افراد و گروه‌هایی خاص برعلیه این جنبش نرم‌افزاری به کار گرفته شده و لطمات جبران‌ناپذیری را بر آن وارد کرده و می‌کنند. امیدوارم که توانسته باشم قدمی هرچند کوتاه در راستای این هدف بردارم.

در این میان جا دارد از پدر و مادر عزیزم که تنها حامی من در دوران تألیف این کتاب بودند، کمال تشکر و قدردانی را داشته باشم.

امید غلامی

تابستان ۱۳۸٤

فهرست مطالب

مقدمه

امروزه سیستم‌های کامپیوتری و نرم‌افزارهای مربوطه به‌طور وسیعی در رشته‌های گوناگون مورد استفاده قرار می‌گیرند و طیف وسیعی از شرکت‌ها و اشخاص در زمینه طراحی و تولید این نرم‌افزارها و سیستم‌های کامپیوتری به فعالیت مشغول هستند. در حال حاضر به دلیل عدم آگاهی اکثر تولید کنندگان نرم‌افزار از تکنیک‌های مهندسی معکوس، تولیدات و حقوق مالکیت آنها بر نرم‌افزار و داده‌های ارزشمندشان مورد تعرض قرار می‌گیرد که این امر لطمه‌های جبران‌ناپذیری را به آنها وارد کرده و می‌کند.

غیرقانونی بودن یادگیری، آموزش و به کارگیری تکنیک‌های مهندسی معکوس در اکثر کشورها، سبب شده است که این علم از گسترش خوبی برخوردار نبوده، و سازماندهی خاصی نیز نداشته باشد و تنها در سطح اشخاص و گروه‌های خاص و با تکیه به تجارب شخصی آنها مورد استفاده قرار بگیرد.

کتاب حاضر حاصل ٦سال تجربه، تحقیق و بررسی پیرامون برنامه‌نویسی سیستمی، سیستم‌های امنیتی و تکنیک‌های مهندسی معکوس می‌باشد. با توجه به ساختار خاص در نظر گرفته شده، این کتاب برای جوابگویی به نیازهای مخاطبین در دو سطح متوسط و پیشرفته تدوین شده است . در تدوین این کتاب سعی شده است مباحث پیشرفته به کلی از مباحث ابتدایی جدا گردد که این امر از ایجاد سردرگمی برای افراد مبتدی جلوگیری خواهد کرد.

بدیهی است که یادگیری و تسلط کامل بر تکنیک‌ها و روش‌های ارائه شده در این کتاب نیازمند تمرین زیاد و صرف وقت کافی است. نگارندگان، این کتاب را تنها شروعی برای مهندسی معکوس می‌دانند و با توجه به نیاز جامعه، امیدوار هستند که متخصصین بتوانند آن را به خوبی ادامه دهند.

مخاطبین

مخاطبین اصلی این کتاب افرادی هستند که آشنایی کاملی با عملکرد و برنامه‌نویسی درمحیط ویندوز دارند ولی با توجه به ساختار خاص درنظر گرفته شده این کتاب می‌تواند در دو سطح متوسط و پیشرفته مورد استفاده قرار گرفته و مفید واقع شود.

- **سطح متوسط :**

افرادی که آشنایی نسبی و کلی با سیستم عامل ویندوز و برنامه‌نویسی با آن را داشته و تنها قصد استفاده از ابزارهای مهندسی معکوس را دارند.

- **سطح پیشرفته :**

افرادی که تسلط کاملی بر ساختار و نحوه عملکرد سیستم عامل ویندوز داشته و از تجربه نسبی در زمینه برنامه‌نویسی سیستمی برخوردار هستند و قصد استفاده و یا تولید ابزارهای مهندسی معکوس و یا نرم‌افزارهای سیستمی را دارند.

سیستم مورد نیاز

اکثر مثال‌های این کتاب برروی هر دو خانواده ویندوزهای 9X و NT قابل انجام هستند. ولی در برخی از موارد روش‌های ذکر شده تنها برروی خانواده NT قابل پیاده‌سازی هستند. لذا توصیه می‌شود که تمرین‌های خود را برروی ویندوزهای 2000 , XP , 2003 ویا بالاتر انجام دهید.

نحوه سازمان‌دهی این کتاب

به افرادی که قصد یادگیری اصولی و به کار بستن این تکنیک‌ها را دارند توصیه می‌شود که پیش از شروع، مروری بر قسمت برنامه‌نویسی به زبان اسمبلی در ویندوز داشته باشند زیرا در حقیقت این بخش پیش زمینه‌های لازم را برای یادگیری اصولی ساختارهای داخلی سیستم عامل ، توابع API و درک کدهای اسمبلی فراهم می‌کند.

به منظور آشنایی بهتر با ساختار و نحوه سازماندهی این کتاب، نگاهی کلی به بخش‌ها و خلاصه مطالب هر یک از آنها می‌اندازیم.

بخش ۱ : « مهندسی معکوس چیست ؟ »

حاوی مقدمه‌ای راجع به مفاهیم اولیه مهندسی معکوس، موارد کاربرد و شرایط و امکانات لازم برای انجام این عملیات است.

بخش ۲ : « جمع آوری اطلاعات اولیه »

این بخش تکنیک‌ها و مراحل جمع آوری اطلاعات درباره نحوه عملکرد و اجزاء نرم‌افزارها را مورد بررسی قرار می‌دهد و ابزارهایی را به منظور تسهیل این عملیات معرفی کرده است. در این بخش نحوه عملکرد و کار با هر یک از آنها با استفاده از مثال‌هایی توضیح داده می‌شود.

بخش ۳ : « بررسی کدها »

این بخش مقدمات اولیه دسترسی به کدهای نرم‌افزارها را معرفی کرده و ابزارهایی را به منظور تحلیل و بررسی این کدها معرفی می‌کند.

بخش ۴ : « Decompiler ها »

این بخش ابزارهایی را به منظور Decompile کردن کدهای اجرایی تولید شده توسط کامپایلرهای گوناگون معرفی می‌کند.

بخش ۵ : « Debugger ها »

این بخش تکنیک‌ها و جزئیات عملیات اجرای مرحله به مرحله و کنترل روند اجرایی برنامه‌ها را توسط Debugger ها مورد بررسی قرار می‌دهد.

بخش ۶ : « ایجاد تغییرات در فایل‌های اجرایی »

بررسی‌های کاملی را بر روی تکنیک‌های مورد استفاده به‌منظور ایجاد تغییرات برروی ساختار، کدها و ظاهر نرم‌افزارها وفایل‌های اجرایی انجام می‌دهد و مثال‌هایی را نیز به‌منظور روشن‌تر شدن بهتر مطالب ارائه می‌کند.

بخش ۷ : « درک کدهای اسمبلی »

این بخش ساختارهای مورد استفاده در زبان‌های سطح بالا برای عملیات کامپایل را به‌طور دقیق مورد بررسی قرار می‌دهد.

بخش ۸ : « برنامه‌نویسی به زبان اسمبلی در ویندوز »

همان‌طور که ذکر شد، در این بخش نحوه ایجاد برنامه‌ها به وسیله زبان اسمبلی توضیح داده شده و در حین آموزش، ساختارها، نحوه عملکرد و نحوه استفاده از توابع API نیز مورد بررسی قرار می‌گیرند. در حقیقت این بخش پیش زمینه‌های لازم را به منظور شروع یادگیری تکنیک‌های مهندسی معکوس فراهم می‌کند.

بخش ۹ : « برنامه‌نویسی و ایجاد درایورها در ویندوزهای خانواده NT »

حاوی اطلاعاتی مفید به‌منظور معرفی ساختار درایورها و نحوه عملکرد و ایجاد آنها می‌باشد. به‌منظور روشن‌تر شدن بهتر مطالب، در این فصل چند نمونه درایور ساده طراحی و پیاده‌سازی شده است.

ضمیمه « دستورالعمل‌های اسمبلی X86 »

به دلیل نیازها و کاربردهای فراوان در این ضمیمه تمامی‌دستورالعمل‌های پردازنده‌های خانواده 80x86 به همراه توضیحات کوتاهی راجع به هر یک از آنها آورده شده است که می‌تواند به عنوان یک مرجع سریع مورد استفاده خوانندگان قرار بگیرد.

ضمیمه CD

این CD حاوی کلیه ابزارها و کدهای مورد استفاده در این کتاب می‌باشد. در بخش‌های مختلف کتاب به آدرس ابزارهای مورد نیاز در CD ضمیمه ارجاع شده است.

ارتباط با نویسندگان

نظرات، انتقادات و پیشنهادهای خود در مورد این کتاب را می‌توانید از طریق آدرس پست الکترونیکی زیر به نویسندگان این کتاب ارسال کنید.

omidgl@gmail.com
behzad.lajevardi@gmail.com

فصل اول

مهندسی معکوس چیست؟

فصل اول

مهندسی معکوس چیست؟

در مفهوم کلی در مراحل تولید یک محصول هرگونه تلاش در جهت رسیدن از یک مرحله بالاتر به مرحله پایین‌تر را مهندسی معکوس می‌نامند. به عنوان مثال رسیدن از محصول تولید شده نهایی به ایده‌ها و روش‌های تولید آن ، نمونه‌ای از مهندسی معکوس محسوب می‌شود . معمولاً این تلاش‌ها به دلایل زیر صورت می گیرند:

۱- شناسایی اجزاء سازنده یک محصول و روابط بین این اجزا.

۲- مستندسازی یک سیستم به نحوی که قابلیت درک بهتری را ایجاد کند.

۳- ایجاد تغییرات در سیستم به نحوی که با خواسته‌های جدید مطابقت داشته باشد (مهندسی مجدد).

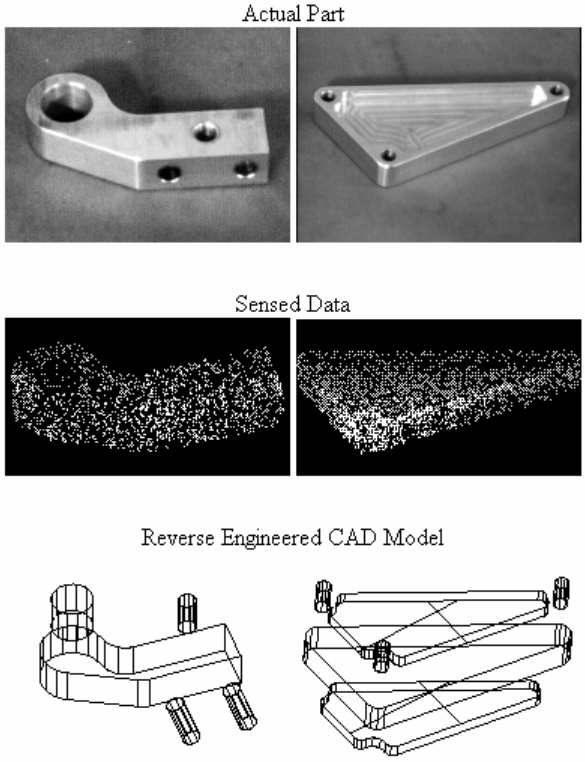
٤- بکارگیری محصول یا اجزای سازنده آن به منظور ساخت یک محصول جدید.

٥- بررسی و ردیاب مشکلات در یک سیستم موجود.

به منظور آشنایی بیشتر با مفاهیم ذکر شده نمونه‌هایی از کاربردهای مهندسی معکوس را در رشته‌های مختلف مورد بررسی قرار می‌دهیم.

مکانیک

کاربردهای مهندسی معکوس در این رشته بسیار مورد توجه قرار گرفته است. به عنوان مثال با گرفتن تصاویر و یا بررسی لیزری از جهات مختلف از یک جزء مکانیکی و پردازش این تصاویر به کمک نرم‌افزارهای کامپیوتری، طرح اولیه جزء مورد نظر با دقت نسبتاً خوبی در کامپیوتر ایجاد می‌شود. با استفاده از نرم‌افزارهای CAD می‌توان آن را مورد بررسی قرار داده و یا در صورت لزوم تغییراتی را در مدل موجود ایجاد کرد. طرح ایجاد شده می‌تواند به منظور کپی‌برداری و یا ایجاد محصول جدید به کار گرفته شود.



شکل (۱-۱)

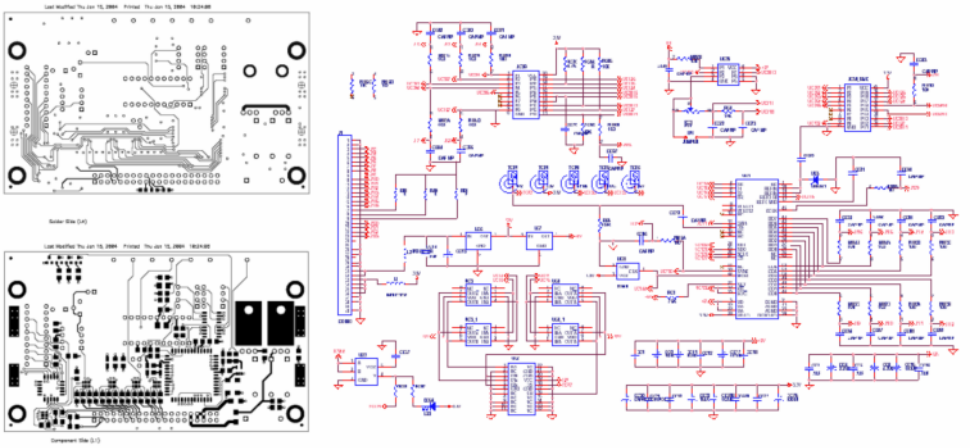
زیست شناسی و نانوتکنولوژی

مهندسی معکوس در رشته نوپای نانوتکنولوژی به کمک مهندسان و طراحان این رشته آمده است. نانوتکنولوژی رشته جدیدی است که برروی طراحی و ساخت اجزاء و سیستم‌های بسیار کوچک در حدود چندین ملکول و یا اتم بحث می‌کند. طراحان این سیستم‌ها از مدل‌های موجود در جهان طبیعی الگوهای بسیاری گرفته‌اند. به عنوان مثال با بررسی دقیق باکتری‌ها، ویروس‌ها و سایر سیستم‌های موجود در طبیعت و نحوه کارکرد آنها توانسته‌اند اهرم‌ها، بازوها و اجزاء متحرکی را با اندازه‌هایی در حدود چند اتم ایجاد کنند.

الکترونیک و سخت‌افزار

در رشته الکترونیک و طراحی دیجیتال مهندسی معکوس محبوبیت بسیاری پیدا کرده است و ابزارهای متنوعی نیز برای تسهیل آن ایجاد شده و مورد استفاده قرار می‌گیرند. به عنوان مثال انواع گوناگونی از اسکنرها برای اسکن و تحلیل مدارهای مجتمع چاپی و ایجاد نت لیست‌ها که در حقیقت طرح اولیه مدار هستند ارائه شده‌اند.

این اسکنرها به کمک یک تحلیل‌گر نرم‌افزاری با پردازش تصاویر گرفته شده از مدار، اجزاء آن را به طور خودکار شناسایی کرده و روابط بین آنها را نیز تشخیص می‌دهند و در نهایت طرح اولیه مدار را در کامپیوتر ایجاد می‌کنند. نمونه‌های دیگری نیز برای پردازش ورودی‌ها و خروجی‌های یک IC و ایجاد جدول منطقی آن وجود دارند که کاربردهای بسیاری دارند.



شکل (۱-۲)

شرایط استفاده و امکانات لازم

با توضیحات داده شده احتمالاً دید کلی نسبت به مهندسی معکوس پیدا کرده‌اید. حال ببینیم که در چه شرایطی از این تکنیک‌ها استفاده می‌شود.

باتوجه به اینکه به کارگیری تکنیک‌های مهندسی معکوس در هر رشته مشکلات خود را داشته و امکانات خاص خود را نیاز دارد، در مورد استفاده از این روش‌ها باید بررسی‌های کاملی صورت گیرد. در برخی از موارد به کارگیری این تکنیک‌ها برای کپی‌برداری و یا ایده گرفتن از یک سیستم موجود از انجام آن بصورت معمولی مشکل‌تر است و نیاز به امکانات بیشتری نیز دارد. در مواردی نیز ممکن است به کارگیری این روش‌ها موفقیت‌آمیز نباشد و با وجود صرف وقت و امکانات فراوان نتیجه مطلوب ایجاد نگردد.

باتوجه به مطالب گفته شده لزوم بررسی دقیق قبل از به کارگیری این روش‌ها از اهمیت ویژه‌ای برخوردار است.

نمونه‌هایی از کاربردها در رشته نرم‌افزار

باتوجه به اینکه در این کتاب مباحث بر روی تکنیک‌ها و روش‌های به کارگیری مهندسی معکوس در رشته نرم‌افزار متمرکز شده‌اند، در این بخش قصد داریم به برخی از کاربردهای آن در این زمینه نگاهی کوتاه بیاندازیم.

کشف کدگذاری‌ها و رمزها

در این زمینه مهندسی معکوس نقش اصلی را برعهده دارد. بدون استفاده از این تکنیک‌ها علم رمزنگاری و کشف رمز (Cryptology) در رشته نرم افزار کاری بسیار طاقت‌فرسا و وقت‌گیر است که بیشتر بر پایه روش‌های آزمون و خطا استوار می‌باشد.

استفاده در مراجع قانونی

در اکثر شکایات از نرم‌افزارهای کاربردی و یا سیستم‌های عامل در مراجع قانونی باتوجه به اینکه معمولاً سورس‌های این نرم‌افزارها در اختیار استفاده کننده گان قرار نمی‌گیرد، تکنیک‌های مهندسی معکوس نقش کلیدی را ایفا می‌کنند. با استفاده از این تکنیک‌ها کارکرد برنامه مورد آزمایش قرار گرفته و در صورت نیاز به بررسی دقیق‌تر، نرم‌افزار مربوطه Decompile شده و بررسی روی کدهای برنامه صورت خواهد گرفت. توجه داشته باشد که در برخی از کشورها به علت ضعف سیستم قضایی این دلایل از طرف مراجع قانونی مورد پذیرش قرار نمی‌گیرند.

مبارزه با ویروس‌ها و کشف نرم‌افزارهای مخرب

مبارزه و کشف این نرم‌افزارها بدون استفاده از روش‌های مهندسی معکوس غیرممکن است زیرا در صورتی که اطلاعات دقیق و کاملی در مورد نحوه عملکرد و روش‌های مورد استفاده در این نرم‌افزارها در اختیار نداشته باشیم، مبارزه با آنها کاری بی‌نتیجه خواهد بود.

در اکثر نرم‌افزارهای ضد ویروس نشانه‌های مختص هر ویروس و یا نرم‌افزار مخرب وجود دارد و باتوجه به اینکه ویروس‌ها معمولاً در روند اجرایی برنامه‌ها تغییر ایجاد می‌کنند، بازگرداندن تغییرات ایجاد شده و درمان فایل‌های آلوده نیازمند در اختیار داشتن جزییات کافی در مورد نحوه عملکرد هر یک از آنها می‌باشد.

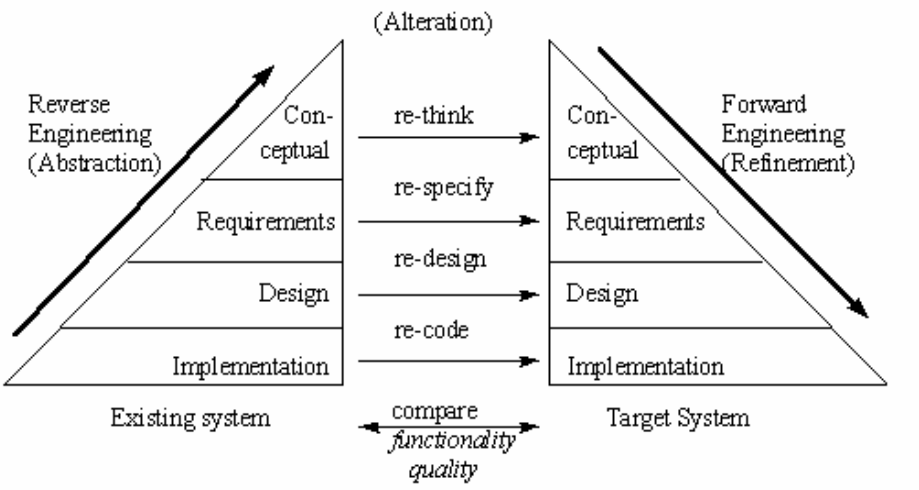
تعییر در روند اجرایی و یا ظاهر یک نرم‌افزار

موارد زیادی ممکن است بوجود بیاید که در آنها نیاز به ایجاد تغییرات و یا اضافه کردن قابلیت‌های جدید به نرم‌افزاری داشته باشیم که سورس کد آن را در اختیار نداریم و یا اینکه شرکت پشتیبان دیگر از آن محصول پشتیبانی نمی‌کند. یکی از این مشکلات که در آغاز هزاره جدید بوجود آمد، مشکل سال ۲۰۰۰ (Y2K) بود که احتمالاً با آن آشنا هستید. در این زمان اکثر شرکت‌ها و کاربران در حال استفاده از نرم‌افزارهایی بودند که شاید سالها مشکلات آنها را حل کرده و اطلاعات آنها را ذخیره و بایگانی کرده بودند. پس از گذشت سال‌ها ، شرکت‌های تولیدکنندهٔ این نرم‌افزارها تغییرات زیادی کرده بودند و معمولاً دیگر از نرم‌افزارهای قدیمی پشتیبانی نمی‌کردند و یا اینکه کاملاً منحل شده بودند. باتوجه به اینکه طراحی نرم‌افزارهای جدید مشکلات بیشتری را در برداشت، شرکت‌ها و کاربران این نرم‌افزارها بار مالی زیادی را برای تغییر این نرم‌افزارها بوسیله تکنیک‌های مهندسی معکوس متحمل شدند.

در موارد زیادی هم نیاز به ایجاد تغییرات در ظاهر نرم‌افزارهای موجود به خاطر پشتیبانی از یک زبان جدید و یا ایجاد تغییرات در لوگوها و یا سایر موارد داریم که با استفاده از روش‌های معمول قابل انجام نیستند. در اینگونه موارد نیز تکنیک‌های مهندسی معکوس به آسانی تغییرات را ایجاد کرده و اعمال می‌کنند.

طراحی مجدد یک نرم‌افزار

یکی از زمینه‌های کاری بسیار مفید مهندسی معکوس طراحی مجدد یک نرم‌افزار براساس یک نمونه موجود است. معمولاً این کار به دلایل زیادی از جمله تغییر سیستم عامل صورت می‌گیرد و کاری بسیار پیچیده و دقیق است که معمولاً به‌صورت تیمی و توسط شرکت‌های بزرگ صورت می‌گیرد. مراحل این کار که به دو قسمت مهندسی معکوس و مهندسی مستقیم (Forwarde...) تقسیم می‌شود به این صورت است که ابتدا کلیه ایده‌ها و روش‌های پیاده‌سازی نرم‌افزار مورد نظر تشخیص داده شده و سپس باتوجه به اطلاعات بدست آمده نرم‌افزار جدیدی طراحی و ایجاد می‌شود. در شکل۱-۳ طرح کلی مراحل انجام چنین پروژه‌هایی را مشاهده می‌کنید.



شکل (۱-۳)

مستندسازی یک نرم‌افزار موجود

یکی از موارد کاربرد مهندسی معکوس در مستندسازی برنامه‌ها و یا توابع کتابخانه‌ای است که بدون سورس کد عرضه می‌شوند و مستنداتی هم برای آنها موجود نیست. یک نمونه از این موارد توابع محلی ویندوز NT (Native API) است که شامل چندین هزار تابع است که برای آنها جز چند صفحه اطلاعات ابتدایی چیزی ارائه نشده و مستنداتی برای هیچ کدام از آنها از سوی شرکت سازنده یعنی مایکروسافت موجود نمی‌باشد.

مواردی که ذکر شده بخشی از کاربردهای مهندسی معکوس در زمینه نرم‌افزار بود. بدیهی است که کاربردهای این تکنیک‌ها تنها به موارد مذکور منحصر نمی‌شود و در شرایط و زمان‌های مختلف می‌توانند کاربردهای متعددی داشته باشند.

در فصل‌های بعدی در مورد روش‌ها و تکنیک‌های به کارگیری مهندسی معکوس توضیحات بیشتری ارائه خواهد شد.

فصل دوم

جمع آوری اطلاعات اولیه

فصل دوم

جمع‌آوری اطلاعات اولیه

می‌توان گفت اولین و مهم‌ترین قدم در مهندسی معکوس جمع‌آوری اطلاعات راجع به سیستم و یا نرم‌افزار مورد نظر است. بدون داشتن این اطلاعات قدم‌های بعدی تقریباً بی‌نتیجه و صرفاً مبتنی بر آزمایش و خطا خواهد بود که کاری بسیار وقت‌گیر و طاقت فرسا است. بدیهی است که هرچه اطلاعات شما در مورد اجزاء سازنده ، روابط بین آنها و سایر موارد بیشتر باشد، قدم‌های بعدی ، شما را با سرعت بسیار بیشتری به هدف می‌رسانند.

در مرحله جمع‌آوری اطلاعات ، آزمایش‌های مختلفی بر روی نرم‌افزار مورد نظر صورت می‌گیرد که در این فصل به آنها خواهیم پرداخت.

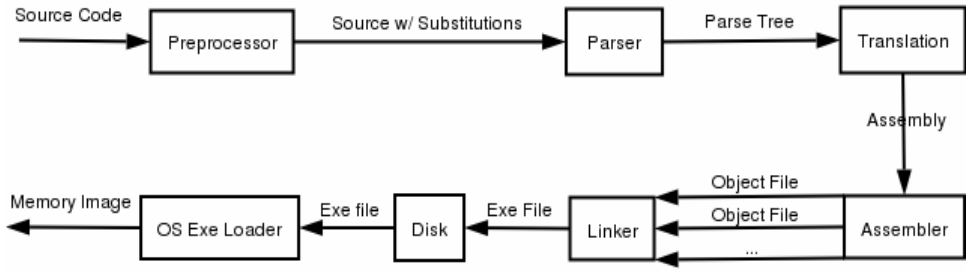
بررسی فایل‌های اجرایی به‌صورت ایستا

مرحله اول از جمع‌آوری اطلاعات، نیازی به اجرا شدن فایل مورد نظر نداشته و در حقیقت آن را به‌صورت ایستا بررسی می‌کند و بیشتر بر روی ساختار فیزیکی فایل تأکید دارد. در صورت نیاز به اطلاعات دقیق‌تر راجع به ساختار فایل‌های اجرایی در ویندوز می‌توانید به فصل ۸ مراجعه کنید.

شناسایی نوع کامپایلر

شناسایی نوع کامپایلر می‌تواند استراتژی ما را در مراحل بعد کاملاً تحت تأثیر قرار دهد. زیرا معمولاً هر کامپایلر در ویندوز ساختار خاصی را برای مدیریت و سازماندهی کدها ، داده‌ها و منابع در فایل اجرایی خاص خود در نظر می‌گیرد که با کامپایلرهای دیگر کاملاً متفاوت است. در نتیجه ، شناسایی نوع کامپایلر اولین قدم در جمع‌آوری اطلاعات محسوب می‌شود.

ابتدا بهتر است نگاهی کوتاه به مراحل کامپایل شدن یک برنامه بیاندازیم.



شکل (۲-۱)

احتمالاً با نمودارشکل (۲-۱) آشنایی دارید. در اینجا قصد نداریم به‌صورت جزیی مراحل کامپایل شدن را بررسی کنیم. ولی به برخی از نکاتی که برای ادامه کار مفید هستند اشاره می‌کنیم.

معمولاً کامپایلرها اسامی، نام‌ها و برچسب‌ها را حذف کرده و به جای آنها آدرس‌ها را در فایل اجرایی جایگزین می‌کنند.

معمولاً فایلهای اجرایی حاوی هیچگونه رشته‌های راهنما (Comment) نیستند زیرا در مرور اول تمام آنها حذف می‌شوند.

خصوصیات فایل‌های اجرایی در کامپایلرهای مختلف

اکثر کامپایلرهای سطح بالا ابتدا کدهای نوشته شده را به زبانی سطح پایین ترجمه کرده و سپس آن را کامپایل می‌کنند. این کار بیشتر به دلیل کاهش پیچیدگی‌های موجود در زبان‌های سطح بالا صورت می‌گیرد و در حقیقت بار کامپایل نهایی را کاهش می‌دهد. به عنوان مثال زبان Visual C++ متکی بر کامپایلر Macro Assembler و زبان C++ Builder متکی بر کامپایلرTurbo Assembler است.

هرچه در مراحل کامپایل جلوتر می‌رویم، خوانایی برنامه نوشته شده برای انسان کمتر شده و برنامه به سمت کد ماشین پیش می‌رود.

حال که با مراحل کامپایل یک برنامه آشنایی نسبی پیدا کردید ، برخی از خصوصیات مهم فایلهای اجرایی را در زبان‌های برنامه‌نویسی معروف مورد بررسی قرار می‌دهیم.

Visual C++

فایل‌های اجرایی تولید شده توسط Visual C++ بطور کامل به زبان ماشین ترجمه می‌شوند و اسامی و برچسب‌ها و سایر راهنماها در فایل اجرایی حذف شده و قابل دسترسی نیستند.

معمولاً منابع (Resource) فایلهای اجرایی تولید شده ، منابع استاندارد از قبیل Dialog ، Bitmap و ... هستند.

فایل‌های اجرایی تولید شده توسط Visual C++ از سرعت بالایی برخوردار هستند. به همین دلیل برای نوشتن درایورهای سخت‌افزاری و برنامه‌هایی که نیاز به سرعت بالا دارند از آن استفاده می‌شود.

C++Builder و Delphi

فایل‌های اجرایی تولید شده توسط این زبان‌ها ، تقریباً خصوصیات مشترکی دارند. از جمله اینکه اسامی کلاس‌ها و اشیاء موجود در برنامه در فایل اجرایی نهایی نیز وجود دارند که این امر باعث خوانایی بیشتر فایل‌های اجرایی آنها شده و در نتیجه مراحل Decompile شدن آنها به سادگی صورت می‌گیرد.

فایل‌های اجرایی تولید شده نیاز به فایل‌های جانبی کمتری دارند زیرا اطلاعات و ریز برنامه‌های لازم معمولاً بصورت Static به فایل اجرایی پیوند زده می‌شود به همین دلیل فایل‌های اجرایی تولید شده توسط این زبان‌ها از حجم بالایی برخوردار هستند. با این وجود به علت اینکه کدها به زبان ماشین ترجمه می‌شوند، فایل‌های کامپایل شده توسط آنها از سرعت نسبتاً خوبی برخوردار هستند.

فایل‌های اجرایی تولید شده توسط این زبان‌ها غیر از منابع استاندارد ، از منابع خاص دیگری نیز برای ذخیره‌سازی رابط کار بر برنامه استفاده می‌کنند.

Visual Basic

فایل‌های اجرایی تولید شده توسط Visual Basic کاملاً به زبان ماشین ترجمه نمی‌شوند و به طور معمول از توابع API به‌طور مستقیم استفاده نمی‌کنند کلیه کارهای اصلی از قبیل مقایسه‌ها، انتصاب‌ها، عملیات منطقی، فراخوانی توابع و... در این فایل‌ها توسط توابع کتابخانه‌ای MSVBVM انجام می‌شوند.

در حقیقت فایل‌های اجرایی تولید شده حاوی حجم اندکی کد ماشین بعلاوه فراخوانی‌های متعددی از فایل MsvbvmXX.dll هستند. به همین دلیل حجم فایل‌های اجرایی تولید شده بسیار کوچک بوده و از سرعت پایینی برخوردار هستند. به منظور روشن‌تر شدن مطلب نگاهی به شکل(۲-۲) بیاندازید:

VB Code		*Compiled VB Code*

Sub Main
Dim a As String
a = Clipboard.GetText
If a = "xman" Then MsgBox a
End Sub
	➔ ➔ ➔	.
		push jmp_MSVBVM60.DLL!__vbaExceptHandler
		mov eax,fs:[00000000h]
		.
		jnz L004016DC
		push L004022CC
		push L00401350
		call [MSVBVM60.DLL!__vbaNew2]
		mov esi,[L004022CC]
		.
		push esi
		push eax
		call [MSVBVM60.DLL!__vbaHresultCheckObj]
		lea ebx,[ebp-18h]
		.
		push esi
		push eax
		call [MSVBVM60.DLL!__vbaHresultCheckObj]
		mov edx,[ebp-18h]
		lea ecx,[ebp-14h]
		mov dword ptr [ebp-18h],00000000h
		call [MSVBVM60.DLL!__vbaStrMove]
		lea ecx,[ebp-1Ch]
		call [MSVBVM60.DLL!__vbaFreeObj]
		mov edx,[ebp-14h]
		push edx
		push L00401374
		call [MSVBVM60.DLL!__vbaStrCmp]
		test eax,eax
		jnz L004017D2
		.
		push 00000000h
		push ecx
		mov dword ptr [ebp-5Ch],00004008h
		call [MSVBVM60.DLL!MSVBVM60.595]
		lea edx,[ebp-4Ch]
		.
		push ecx
		push 00000003h
		call [MSVBVM60.DLL!__vbaFreeVarList]

شکل (۲-۲)

فایل‌های اجرایی تولید شده به طور معمول از هیچ یک از منابع استاندارد ویندوز مانند Dialog ها، Bitmapها و .... استفاده نمی‌کنند و در حقیقت فرمت ذخیره‌سازی و محل منابع با تمام فایل‌های اجرایی تولید شده توسط سایر کامپایلرها متفاوت بوده و فایل msvbvmXX.dll مسئولیت مدیریت آنها را در زمان اجرا بر عهده می‌گیرد.

.NET (..., VB .Net, C#)

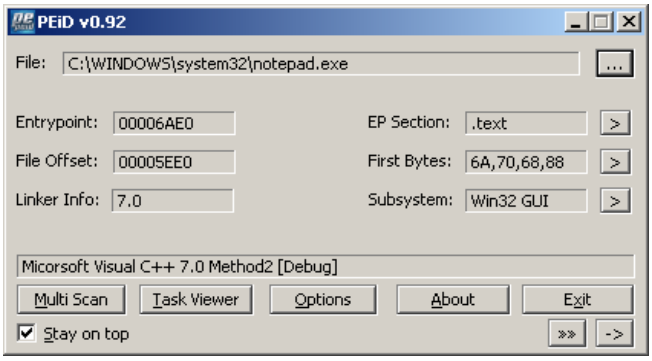
ساختار اجرایی فایل‌های تولید شده توسط کامپایلرهای .NET با ساختار فایل‌های تولید شده توسط سایر کامپایلرها کاملاً متفاوت است . کدها به زبان ماشین ترجمه نمی‌شوند بلکه به یک زبان واسط به نام IL (Intermediate Language) ترجمه شده و توابع موجود در کتابخانه‌های .NET Framework طی مراحلی آنها را به دستورات قابل اجرا توسط ماشین تبدیل می‌کنند. همین امر باعث شده است که فایل‌های اجرایی تولید شده توسط این زبان‌ها بسیار کند بوده و منابع سیستم را با سرعت بالایی مصرف کنند.

همانند Visual Basic این زبان‌ها نیز به طور معمول از منابع استاندارد ویندوز برای ایجاد رابط کاربر و ... استفاده نمی‌کنند و مدیریت جداگانه‌ای برای این موارد دارند.

حال که با خصوصیات فایل‌های اجرایی تولید شده توسط زبان‌های برنامه‌سازی معروف آشنایی نسبی پیدا کردید بهتر است به سراغ ابزارهای تشخیص نوع زبان و کامپایلر برویم.

نرم‌افزار PEiD

یکی از قوی‌ترین ابزارها برای تشخیص نوع کامپایلر و زبان مورد استفاده برای ایجاد فایل‌های اجرایی است که علاوه بر زبان، نوع کامپایل شدن برخی از فایل‌ها را نیز مشخص می‌کند. یکی دیگر از قابلیت‌های مفید این برنامه، سازگاری آن با کامپایلرهای .NET است. در شکل زیر، صفحه اصلی این برنامه را مشاهده می‌کنید.



شکل(۲-۳)

روش کار با این برنامه بسیار ساده بوده و نیازی به توضیح ندارد.

نسخه 0.92 از این نرم‌افزار در CD ضمیمه موجود می‌باشد

Tools\PEiD

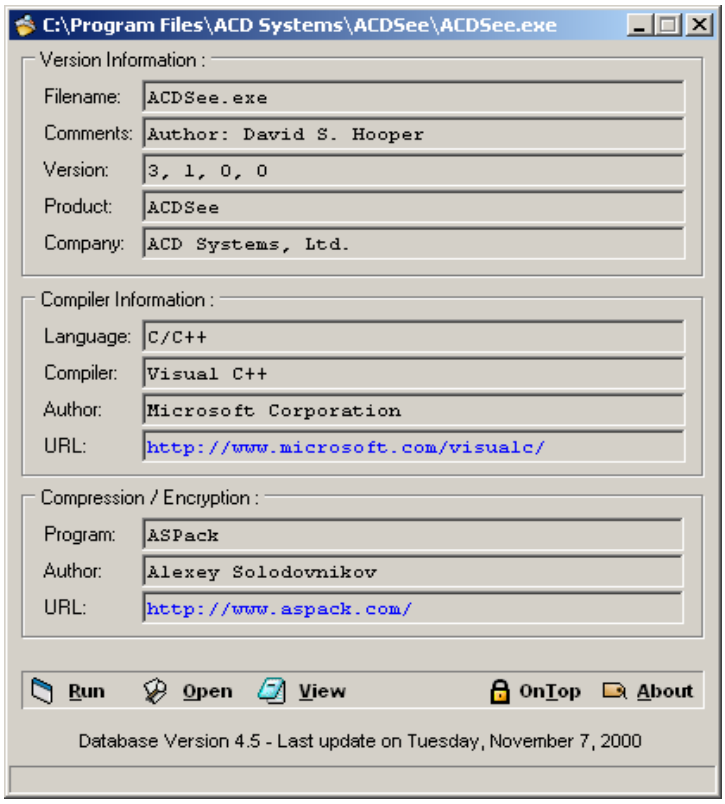
نرم‌افزار Language 2000

یکی دیگر از ابزارهای تشخیص نوع زبان و کامپایلر است که توسط آقای بابک فرخی طراحی شده است و طیف وسیعی از زبان‌های برنامه‌سازی را تحت پوشش قرار می‌دهد.

نسخه 4.5 این نرم‌افزار در CD ضمیمه موجود می‌باشد

Tools\Language2000

در شکل زیر، صفحه اصلی این نرم افزار را مشاهده می‌کنید.



شکل (۲-٤)

بررسی توابع ورودی

همان‌طور که می‌دانید، سیستم عامل ویندوز از کتابخانه‌هایی با لینک پویا (dll) برای به اشتراک گذاشتن کدها و منابع بین برنامه‌های مختلف استفاده می‌کند. یک نمونه از این موارد، کتابخانه‌های استاندارد ویندوز و توابع API هستند که فایل‌های اجرایی در این سیستم عامل از آنها به‌طور مشترک استفاده می‌کنند. فایل‌های اجرایی در ویندوز نام کتابخانه‌ها و توابع مورد نیاز را در خود ذخیره دارند. در نتیجه سیستم عامل در هنگام بارگذاری آنها، توابع و کتابخانه‌های مورد نیاز را به حافظه بارگذاری کرده و یک سری عملیات تصحیح آدرس و... را انجام می‌دهد. برای بدست آوردن اطلاعات کامل‌تر راجع به توابع ورودی و APIها می‌توانید به فصل ۸ مراجعه کنید.

با توجه به این نکته که برنامه‌ها در سیستم عامل ویندوز از توابع API برای انجام کارهای خود استفاده می‌کنند، درمی‌یابیم که داشتن اطلاعات کافی در مورد توابع مورد استفاده در یک فایل اجرایی و کاربرد هر یک از آنها می‌تواند در مورد شناسایی اعمال مورد نیاز برنامه‌ها، ما را یاری دهد. در نتیجه می‌توانید طرحی کامل‌تر و با جزییات بیشتر در مورد برنامه موردنظر در ذهن خود مجسم کنید.

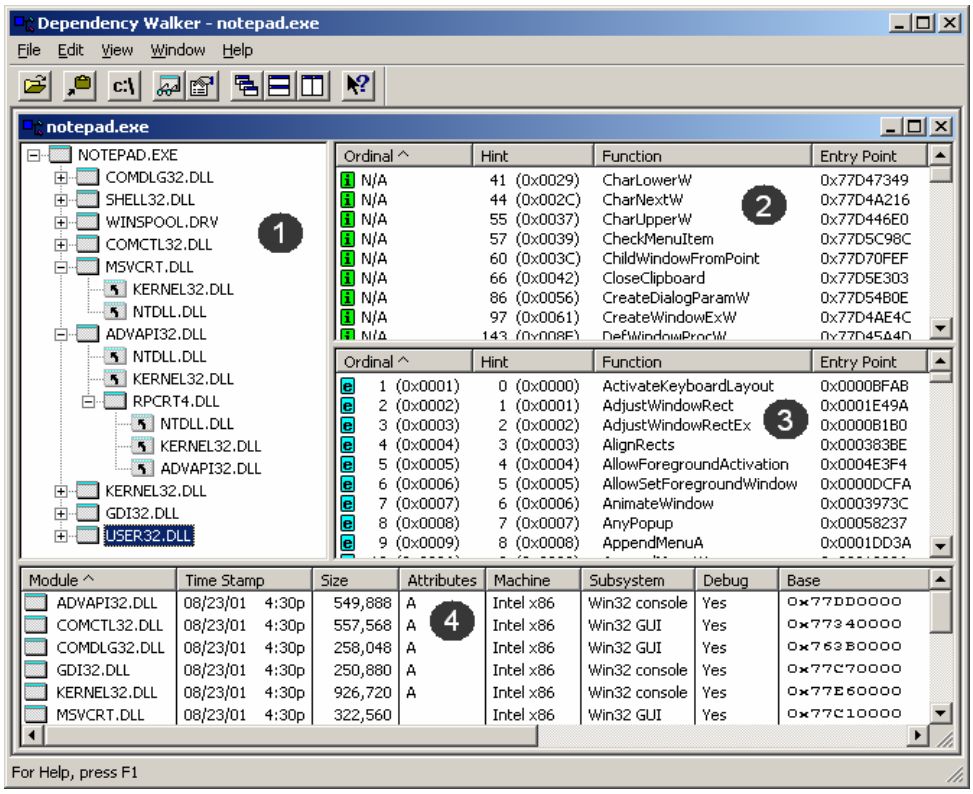
نرم‌افزار Dependency Walker

نرم‌افزارهای زیادی برای بررسی فایل‌های dll و توابع مورد نیاز یک فایل اجرایی وجود دارند که یکی از معروف‌ترین آنها نرم‌افزار Dependency Walker است که به همراه بسته نرم‌افزاری Visual Studio 6 نیز ارائه شده.

نسخه 1 این نرم‌افزار در CD ضمیمه موجود می‌باشد

Tools\DependencyWalker

در شکل (۲-۵) صفحه اصلی این نرم‌افزار را مشاهده می‌کنید که نتایج بررسی برروی فایل Notepad.exe را نشان می‌دهد.



شکل (۲-۵)

ناحیه شماره 1 نمودار درختی وابستگی‌ها را نمایش می‌دهد. به عنوان مثال همانگونه که مشاهده می‌کنید فایل اجرایی Notepad.exe از توابع کتابخانه Msvcrt.dll استفاده کرده است که خود به کتابخانه‌های ntdll.dll و Kernel32.dll وابسته است.

ناحیهٔ شمارهٔ 2 لیست توابع ورودی از یک کتابخانهٔ خاص را نمایش می‌دهد.

ناحیهٔ شمارهٔ 3 لیست کلیهٔ توابع موجود در کتابخانهٔ موردنظر را نشان می‌دهد.

ناحیهٔ شمارهٔ 4 جزییات بیشتری را در مورد فایل‌های کتابخانه‌های مورد استفاده در برنامه مشخص می‌کند. به عنوان مثال قسمت Time Stamp تاریخ کامپایل شدن کتابخانهٔ موردنظر را مشخص کرده و قسمت Base، آدرس مجازی کتابخانهٔ موردنظر را درهنگام نگاشت به حافظهٔ برنامه نشان می‌دهد.

بررسی منابع

منابع (Resource) یکی از مهمترین اجزاء فایل‌های اجرایی محسوب می‌شوند که درحقیقت داده‌های از پیش تعریف شده برنامه را نگهداری می کنند. این داده‌ها می‌توانند حاوی تصاویر، رشته‌های متنی، صداها، پنجره‌های از پیش تعریف شده و بسیاری از انواع دیگر باشند. سازندهٔ فایل اجرایی می‌تواند به دلخواه داده‌های متنوعی را با فرمت‌های موردنظر به منابع فایل اجرایی خود اضافه کند.

باتوجه به اینکه معمولاً رابط کاربر برنامه که توسط اشکال گرافیکی، پنجره‌ها و... ایجاد می‌شود، در قسمت منابع فایل اجرایی قرار دارد، بررسی منابع موجود در فایل اجرایی موردنظر می‌تواند دید بسیار خوبی نسبت به اجزاء و عملکرد آنها ایجاد کند.

در زیر لیستی از منابع استاندارد را به همراه توضیحات کوتاهی در مورد هر یک مشاهده می‌کنید:

Accelerator : کلیدهای میانبر منوهای برنامه را ذخیره می‌کنند.

Animated Cursor : کرسرهای متحرک هستند که می‌توانند به عنوان اشاره‌گر ماوس به کار گرفته شوند.

Animated Icon: آیکون‌های متحرک هستند که می‌توانند تصاویر متحرک کوچکی را ذخیره کنند.

Bitmap : تصاویر نقشه بیتی استاندارد ویندوز هستند که در طراحی رابط کاربر و ... از آنها استفاده می‌شود. در صورت نیاز به اطلاعات دقیق‌تر راجع به بیت‌مپ‌ها و نحوهٔ استفاده از آنها می‌توانید به فصل ۸ مراجعه کنید.

Cursor : کرسرهای گرافیکی مورد استفاده در برنامه که هر کدام دارای مشخصات خاصی از قبیل تعداد رنگ‌ها و ابعاد هستند. باتوجه به اینکه سخت‌افزارهای مختلف ممکن است قابلیت نمایش رنگ‌های موجود در یک کرسر را نداشته باشند ، این کرسرها به سخت‌افزار وابسته‌اند.

Dialog : دیالوگ باکس‌ها در حقیقت پنجره‌های از پیش تعریف شده‌ای هستند که در برنامه از آنها استفاده می‌شود و ویندوز سیستم مدیریتی پیش فرضی برای آنها درنظر گرفته است. در صورت نیاز به اطلاعات دقیق‌تر راجع به دیالوگ باکس‌ها و نحوهٔ استفاده از آنها می‌توانید به فصل ۸ مراجعه کنید.

Font : فونت‌های مورد نیاز برنامه هستند که می‌توانند در ویندوز ثبت شده و به کار گرفته شوند.

Group Cursor : گروهی از کرسرهای موجود در قسمت Cursors را به همراه مشخصات هر یک مشخص می‌کنند که سیستم عامل می‌تواند باتوجه به محیط سخت‌افزاری موجود مناسب‌ترین آنها را

برای نمایش انتخاب کند. در نتیجه می‌توان گفت که این گروه‌ها کرسرهایی مستقل از سخت‌افزار ایجاد می کنند.

Group Icon : گروهی از آیکون‌های موجود در قسمت Icons را مشخص می‌کنند که خصوصیاتی مشابه گروه‌های کرسر دارند.

Icon : آیکون‌های مورد نیاز برنامه هستند که معمولاً درطراحی رابط کاربر از آنها استفاده می‌شود.

Menu : منوهای مورد نیاز برنامه هستند که معمولاً در طراحی پنجره‌ها و رابط کاربر از آنها استفاده می‌شود. در صورت نیاز به اطلاعات دقیق‌تر راجع به منوها در ویندوز و نحوهٔ استفاده و مدیریت آنها می‌توانید به فصل ۸ مراجعه کنید.

String : رشته‌های متنی مورد استفادهٔ برنامه هستند که معمولاً برای ایجاد پیغام‌های مورد نیاز از آنها استفاده می‌شود.

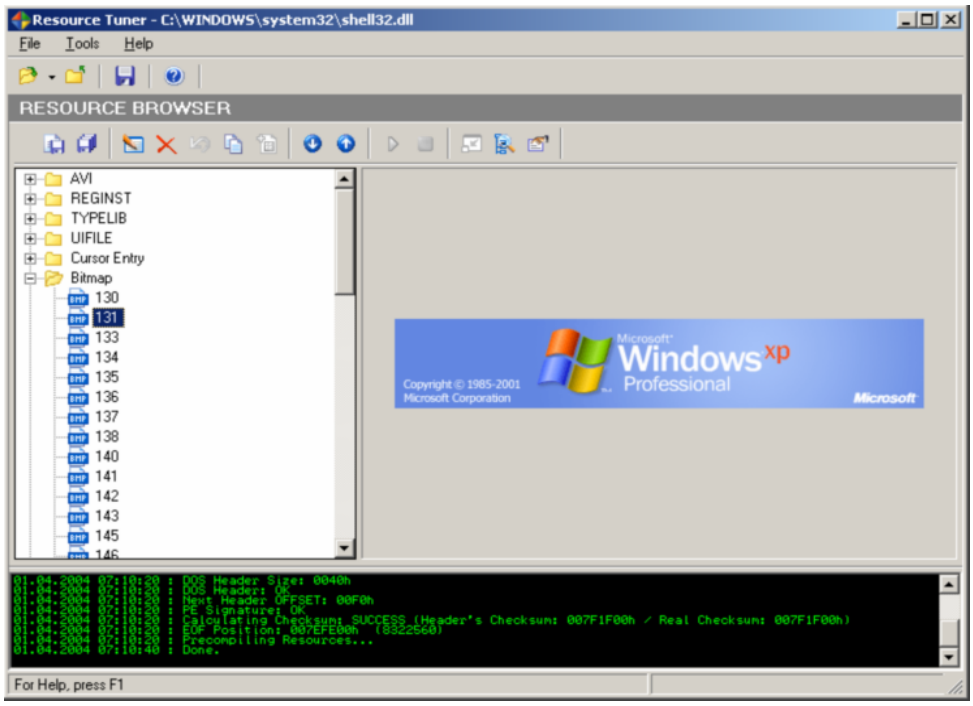
Version : اطلاعاتی راجع به نام اصلی محصول، شرکت سازنده، تاریخ ساخت و سایر موارد را مشخص می‌کند که در حقیقت شناسنامهٔ فایل اجرایی است.

حال که با انواع منابع آشنایی پیدا کردید، بهتر است به سراغ ابزارهای بررسی منابع فایل‌های اجرایی برویم:

نرم‌افزار Resource Tuner

یکی ازقوی‌ترین ابزارها برای بررسی ودسترسی به‌منابع فایل‌های اجرایی برنامهٔ Resource Tuner ساخت شرکت Heaventools است که از رابط کاربر بسیار خوبی نیز برخوردار می‌باشد. یکی از امکانات بسیار قوی در این نرم‌افزار، توانایی آن در بررسی منابع خاص کامپایلرهای ساخت شرکت Borland از قبیل Delphi و C++ Builder است.

در شکل (۲-۶) صفحهٔ اصلی این نرم‌افزار را در حال بررسی منابع فایل Shell32.dll مشاهده می‌کنید.



شکل (۲-۶)

بررسی منابع در این نرم‌افزار بسیار ساده بوده و نیاز به توضیح خاصی ندارد.

در صورت نیاز کاربر می‌تواند با استفاده از گزینه‌های Save در این نرم‌افزار، منابع موردنظر خود را به‌صورت فایل‌های جداگانه‌ای ذخیره کرده و از آنها استفاده کند.

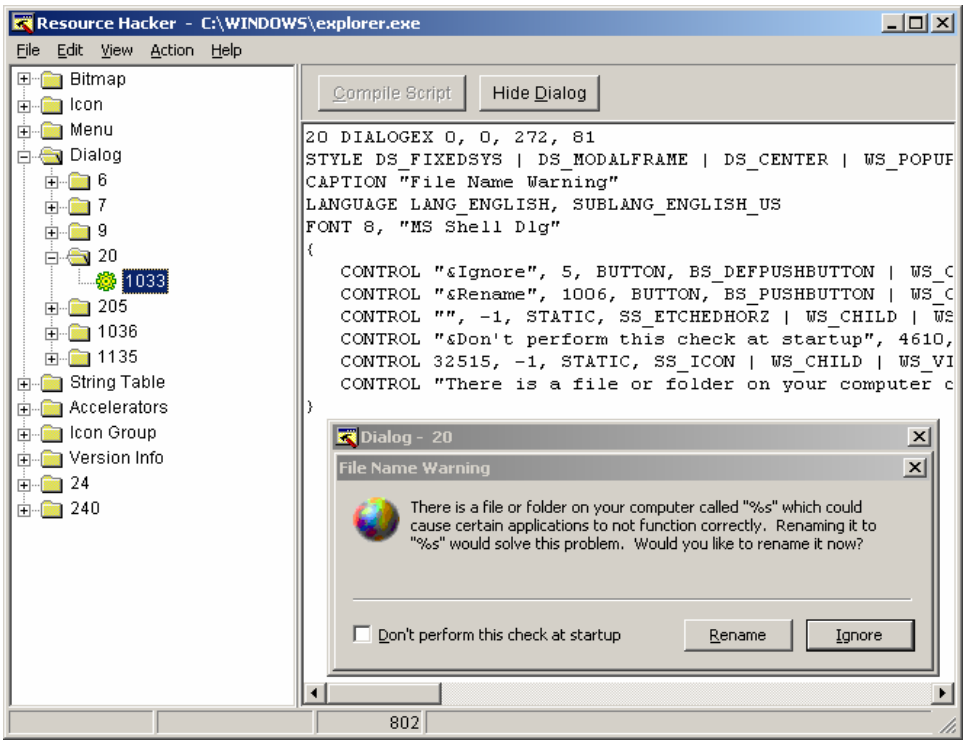
نسخه 1.97 این نرم‌افزار در CD ضمیمه موجود می‌باشد

Tools\ResourceTuner

نرم‌افزار Resource Hacker

یکی دیگر از نرم‌افزارهای مفید در زمینهٔ دسترسی به منابع فایل‌های اجرایی نرم‌افزار Resource hacker است که با وجود رابط کاربر ضعیف، از قابلیت‌های خوبی برخوردار می‌باشد.

در شکل (۲-۷) صفحهٔ اصلی این نرم‌افزار را در حال بررسی منابع فایل اجرایی Explorer.exe مشاهده می‌کنید.



شکل (۲-۷)

بررسی منابع در این نرم‌افزار بسیار ساده بوده و نیاز به توضیح خاصی ندارد. این نرم‌افزار نیز از قابلیت ذخیرهٔ منابع در فایل‌های جداگانه برخوردار است.

نسخه 3.4 این نرم‌افزار در CD ضمیمه موجود می‌باشد

Tools\ResourceHacker

در این قسمت بیشتر در مورد بررسی منابع مطالبی بیان شد. در فصل ایجاد تغییرات در مورد تکنیک‌ها و روش‌های ایجاد تغییرات در منابع بحث‌های بیشتری مطرح می شود و ابزارهای متنوعی نیز مورد بررسی قرار می‌گیرند.

بررسی ساختار فایل‌های اجرایی

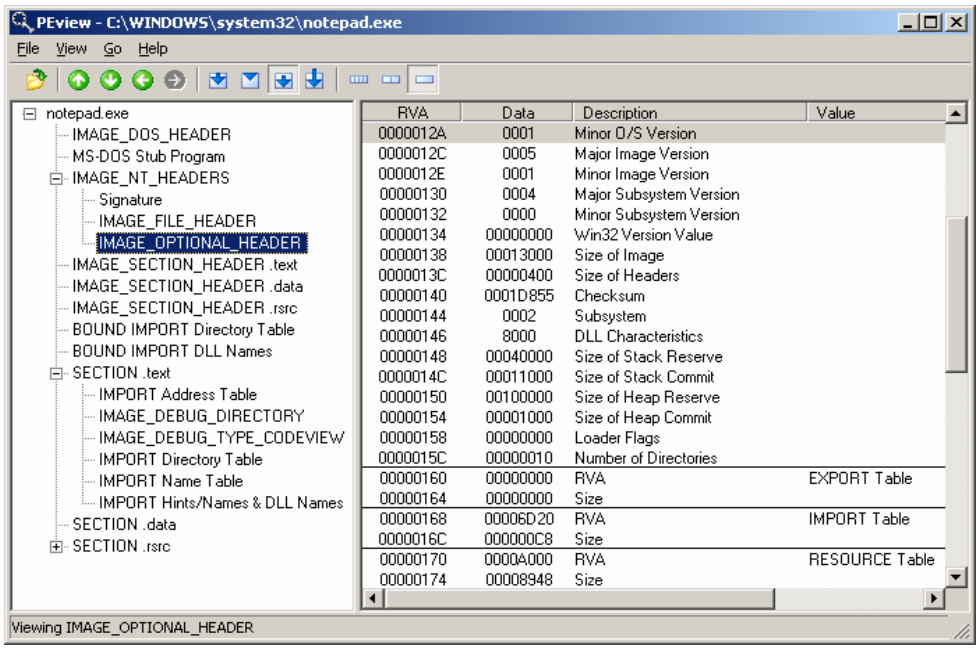
داشتن اطلاعات کافی در مورد داده‌های موجود در رکوردهای کنترلی و Header های یک فایل اجرایی می‌تواند نکات زیادی را در مورد نحوهٔ سازماندهی اطلاعات در فایل مورد نظر مشخص کند. برای بررسی این اطلاعات، داشتن آگاهی در مورد ساختار فایل‌های اجرایی ضروری است. به منظور آشنایی دقیق تر با ساختار فایل های اجرایی، ۷ بخش از فصل ۸ به طور کامل به این مطالب اختصاص یافته‌اند که در صورت نیاز می‌توانید به آنها مراجعه کنید.

حال به معرفی یکی از ابزارهای مفید در زمینهٔ بررسی ساختار فایل‌های اجرایی می‌پردازیم:

نرم‌افزار PEView

این نرم‌افزار اطلاعات موجود در فایل اجرایی را از ابتدا تا انتها بررسی و دسته‌بندی کرده و داده‌های موجود در رکوردهای کنترلی و Header ها را به همراه توضیحات کوتاهی در مورد هر یک نمایش می‌دهد. با نگاهی دقیق به اطلاعات نمایش داده شده در این نرم‌افزار می‌توانید طرح کلی از نحوهٔ سازماندهی اطلاعات در یک فایل اجرایی را مجسم کنید.

شکل (۲-۸)، صفحهٔ اصلی این نرم‌افزار را در حال نمایش داده‌های رکورد IMAGE_OPTIONAL_HEADER از فایل اجرایی Notepad.exe نشان می‌دهد.



شکل (۲-۸)

بررسی اطلاعات توسط این نرم‌افزار بسیار ساده بوده و نیاز به توضیح خاصی ندارد.

نسخه 0.8 این نرم‌افزار در CD ضمیمه موجود می‌باشد

Tools\PEView

بررسی فعالیت‌های فایل‌های اجرایی در مرحلهٔ اجرا

در قسمت قبل در مورد روش‌ها و تکنیک‌های بررسی فایل‌های اجرایی بصورت ایستا مطالبی را بیان کردیم. ولی معمولاً اینگونه بررسی‌ها نمی‌توانند اطلاعات دقیق و کاملی را در مورد نحوهٔ عملکرد برنامهٔ مورد نظر ایجاد کنند. در حقیقت بررسی‌های ایستا مقدمه‌ای برای انجام بررسی‌های زمان اجرا خواهند بود. حال ببینیم که بررسی‌های زمان اجرا چه هستند و چگونه عمل می‌کنند.

با اجرا شدن یک برنامه، سیستم عامل در صورت امکان، منابع اولیهٔ لازم برای شروع آن از قبیل فضای مجازی حافظه، فایل‌های اولیه مورد نیاز و ... را مهیا کرده و در حقیقت زمینه‌های لازم برای اجرای آن را ایجاد می‌کند. توجه داشته باشید که این آماده‌سازی‌ها، مستقیماً از طرف سیستم عامل

و اکثراً باتوجه به اطلاعات موجود در رکوردهای کنترلی فایل اجرایی انجام می‌شوند و در این مرحله هنوز هیچ دستوری از برنامهٔ مورد نظر اجرا نشده است. این مرحله، مرحلهٔ بارگذاری نام دارد. قسمتی ازاطلاعات مربوط به‌عملیات بارگذاری برای تمام فایل‌های اجرایی مشترک بوده و قسمت‌های دیگر توسط بررسی‌های ایستا نیز قابل دستیابی هستند.

با اجرا شدن دستورات ، برنامه بسته به شرایط، نیازهای دیگری نیز پیدا کرده که باید از طرف سیستم عامل به آنها پاسخ داده شود. یک نمونه از این موارد نیاز برنامه به کتابخانه‌هایی است که در مرحلهٔ قبل به حافظه بارگذاری نشده‌اند. موارد دیگری از قبیل نیاز به منابع بیشتر نیز وجود دارند که در زمان اجرا از طرف برنامه به سیستم عامل گزارش شده و در صورت امکان به آنها پاسخ مثبت داده می‌شود.

داشتن اطلاعات دقیق در مورد فعالیت‌های زمان اجرا می‌تواند در شناخت نحوهٔ عملکرد برنامه و روش‌های به کار گرفته شده توسط آن، بسیار مفید بوده و تصویر ذهنی واضح‌تری را برای مراحل بعد ایجاد کند.

بررسی جزئیات Process

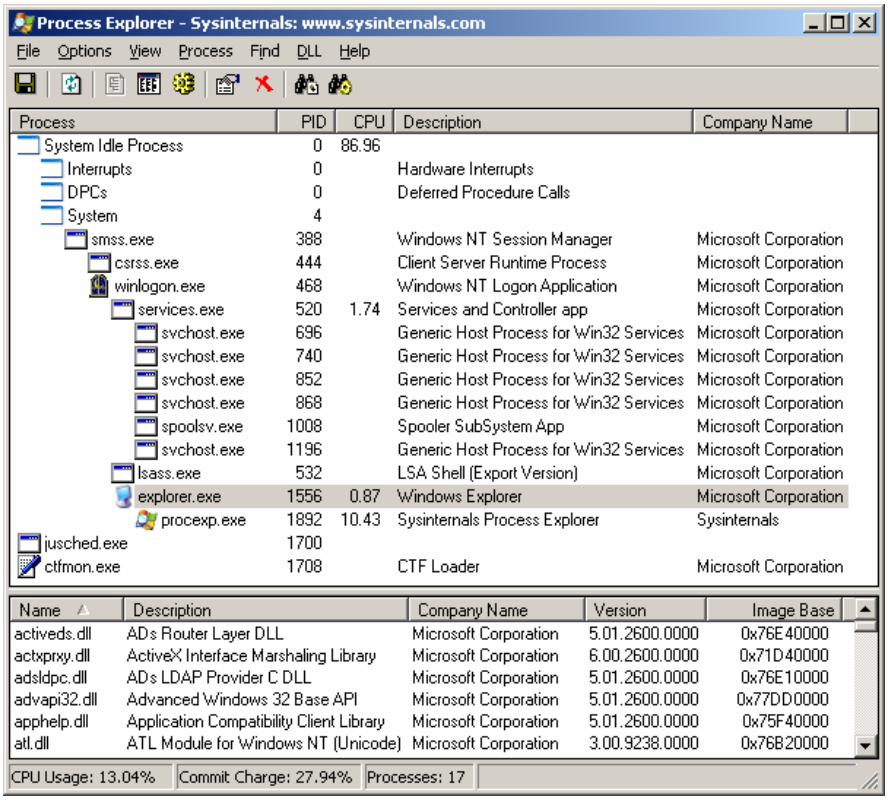
هر برنامهٔ در حال اجرا با تمام زیر روال‌ها، Thread ها و... در ویندوز به عنوان یک Process درنظر گرفته می‌شود که دارای فضای حافظهٔ اختصاصی، شماره‌های اختصاصی دسترسی به منابع و بسیاری از موارد اختصاصی دیگر است که در این قسمت به بررسی آنها نخواهیم پرداخت. در صورت نیاز به اطلاعات دقیق‌تر و کامل‌تر راجع به Process ها و نحوهٔ عملکرد و مدیریت آنها می‌توانید به فصل ۸ مراجعه کنید.

نرم‌افزارهای زیادی برای بررسی Process های موجود در سیستم و ایجاد تغییرات مورد نیاز در خصوصیات آنها وجود دارند . در این قسمت ابتدا دو نمونه از آنها را معرفی کرده و سپس برخی از اعمال کلیدی برروی Process ها را توسط آنها توضیح خواهیم داد. توجه داشته باشید که Processها در ویندوزهای خانواده NT دارای خصوصیات متفاوتی نسبت به ویندوزهای 9X هستند. یکی از اینگونه موارد خصوصیات امنیتی Process ها در ویندوزهای خانوادهٔ NT می‌باشد که در ویندوزهای 9X وجود ندارد.

نرم‌افزار Process Explorer

یکی از قوی‌ترین نرم‌افزارها برای کار بر روی Process ها، نرم‌افزار Process Explorer است که امکانات منحصر به فردی را در اختیار کاربر قرار داده و جزئیات بسیار دقیقی در مورد Processهای موجود در سیستم ارائه می‌کند. در ادامه درمورد نحوهٔ استفاده از برخی از قابلیت‌های کلیدی این نرم‌افزار مطالب بیشتری را بیان خواهیم کرد.

شکل (۲-۹) صفحهٔ اصلی این نرم‌افزار را در حال کار برروی سیستم عامل Win XP نشان می‌دهد.



شکل (۲-۹)

همانطور که مشاهده می‌کنید این نرم‌افزار لیستی از کلیه Process های فعلی را به‌صورت سلسله مراتبی نمایش می‌دهد.

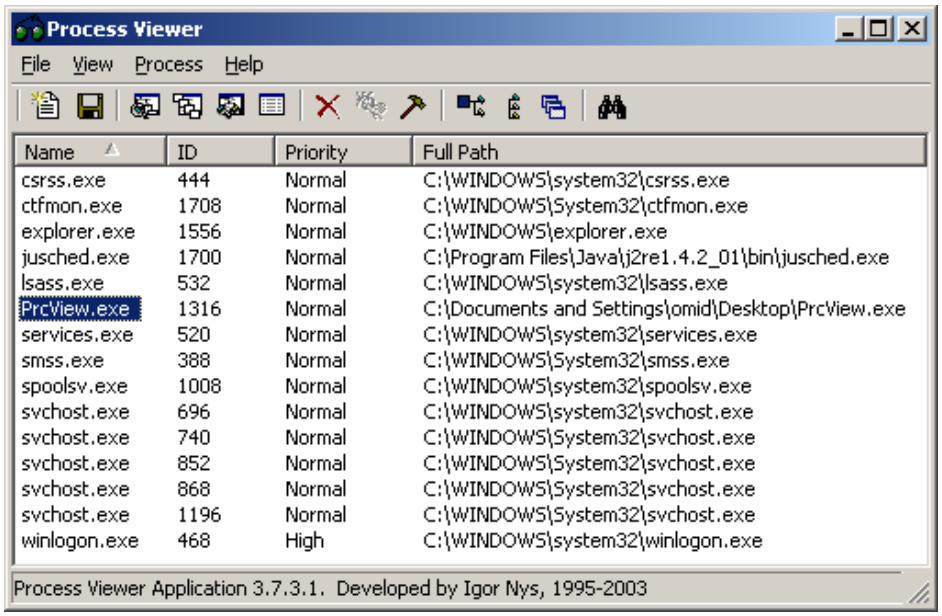
نسخه 8.2 این نرم‌افزار در CD ضمیمه موجود می‌باشد

Tools\ProcessExplorer

نرم‌افزار Process Viewer

Process Viewer یکی دیگر از نرم‌افزارهایی است که بیشتر برای بررسی خصوصیات Process ها از آن استفاده می‌شود و توانایی‌های اندکی برای ایجاد تغییرات دارد. در ادامه از برخی از قابلیت‌های این نرم‌افزار استفاده خواهیم کرد.

در شکل (۲-۱۰) صفحهٔ اصلی این نرم‌افزار را در حال کار برروی Win XP مشاهده می‌کنید.



شکل (۲-۱۰)

نسخه 3.7 این نرم‌افزار در CD ضمیمه موجود می‌باشد

Tools\ProcessViewer

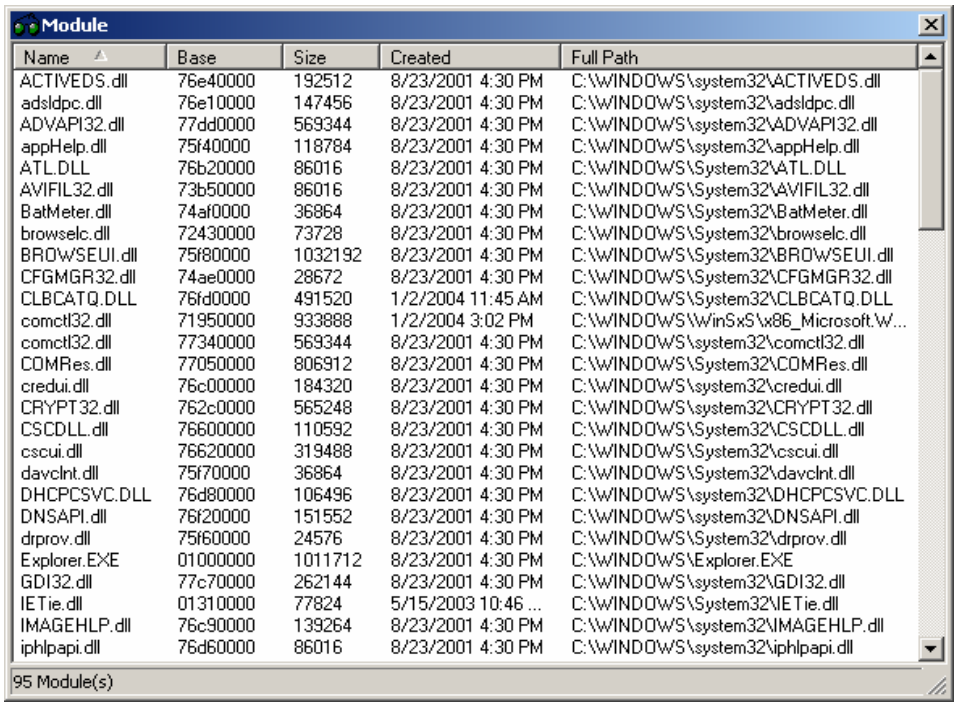
حال که با Process ها و نرم‌افزارهای مربوطه آشنایی نسبی پیدا کردید، به برخی از مراحل اصلی بررسی آنها اشاره می‌کنیم:

بررسی dll ها

همان‌طور که گفته شد برنامه‌ها در ویندوز از dllها به منظور توزیع و به اشتراک‌گذاری کدها و داده‌ها استفاده می‌کنند. برخی از این dllها که جزء توابع ورودی فایل اجرایی هستند، توسط سیستم عامل در هنگام بارگذاری فایل اجرایی به حافظه بارگذاری می‌شوند. ولی برنامهٔ مورد نظر در هنگام اجرا نیز می‌تواند dllهای دیگری را بسته به شرایط به حافظه بارگذاری کرده و یا از آن خارج کند. در برخی از نرم‌افزارها از لغت module به جای فایل‌های dll یا exe استفاده شده است که در این مبحث از نظر مفهومی تقریباً معادل هستند.

باتوجه به اینکه dll ها در هنگام بارگذاری به حافظهٔ نگاشت می‌شوند ، داشتن اطلاعات دقیق راجع به dll هایی که در یک وضعیت توسط برنامه از آنها استفاده می‌شود و همچنین آدرس شروع آنها در حافظهٔ مجازی برنامه می‌تواند در شناخت اجزاء و روابط بین آنها بسیار مفید باشد. همچنین در مراحل Debug داشتن این اطلاعات به عنوان اولین قدم محسوب می‌شود. در صورت نیاز به اطلاعات دقیق‌تر و کامل‌تر راجع به نحوهٔ بارگذاری dll ها و فایل‌های اجرایی توسط سیستم عامل می‌توانید به فصل ۸ مراجعه کنید.

به عنوان مثال در این قسمت از نرم‌افزار Process Viewer به منظور مشاهدهٔ اطلاعات مربوط به dllهای بارگذاری شده توسط explorer.exe, Process استفاده می‌کنیم. با کلیک راست برروی explorer.exe در لیست Process ها در صفحهٔ اصلی و انتخاب گزینهٔ Modules، پنجرهٔ Modules باز شده و لیستی از dllهای فعلی مورد استفاده توسط فایل اجرایی explorer.exe را به همراه جزییاتی در مورد آنها به نمایش می‌گذارد که در شکل (۲-۱۱) مشاهده می‌کنید.



شکل (۲-۱۱)

همان‌طور که مشاهده می‌کنید برای هر dll، اطلاعاتی نظیر آدرس آن در حافظهٔ مجازی برنامه، اندازه، تاریخ ایجاد و مسیر کامل ارائه شده است.

بررسی Threadها (صف های دستورات)

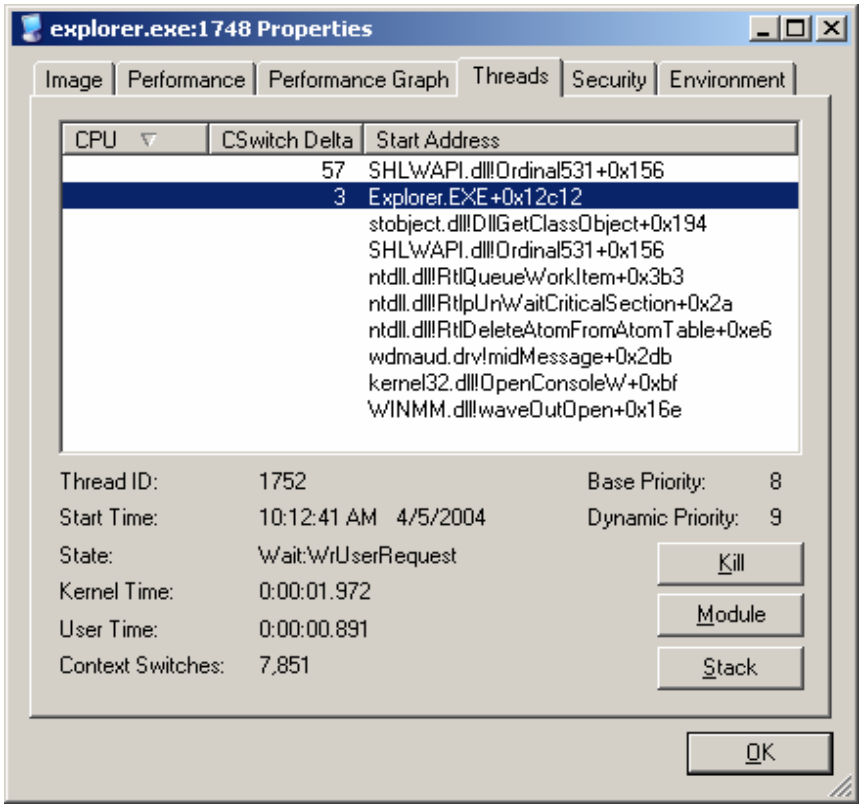
همان‌طور که می‌دانید در سیستم عامل‌های جدید به منظور پیاده‌سازی امکان اجرای چندین برنامه در یک لحظه و یا اجرای بخش‌های مختلف یک برنامه به‌صورت موازی و همزمان، از مفاهیمی مانند Process و Thread استفاده می‌شود.

یک برنامه می‌تواند به چند Thread تقسیم شود که به‌طور همزمان اجرا شده و هر کدام وظیفهٔ خاصی را برعهده دارند. به عنوان مثال در برخی از برنامه‌های کاربردی یک Thread مسئول ایجاد و نگهداری رابط کاربر بوده و Threadهای دیگر مسئولیت ارتباط با کاربر و پردازش داده‌ها را برعهده دارند. توجه داشته باشید که Threadهای مختلفی نیز وجود دارند که توسط فایل‌های dll و یا ActiveXهای مورد استفادهٔ برنامه ایجاد شده و مدیریت می‌شوند.

در صورت نیاز به اطلاعات دقیق‌تر راجع به Threadها و تکنیک‌های Multithreading در برنامه‌ها و نیز متدهای ارتباطی بین Threadها می‌توانید به فصل ۸ مراجعه کنید.

در این قسمت از نرم‌افزار Process Explorer به منظور مشاهدهٔ Threadهای موجود در explorer.exe و بررسی خصوصیات آنها استفاده می‌کنیم.

با کلیک راست برروی Process موردنظر در پنجرهٔ اصلی نرم‌افزار و انتخاب گزینهٔ Properties، صفحهٔ مربوط به خصوصیات process نمایش داده می‌شود. در بخش Threads از این صفحه، لیست کلیهٔ Threadهای در حال کار برای process موردنظر به همراه اطلاعات دقیقی راجع به خصوصیات و وضعیت فعلی آنها نمایش داده می‌شود که در شکل (۲-۱۲) آن را مشاهده می‌کنید.



شکل (۲-۱۲)

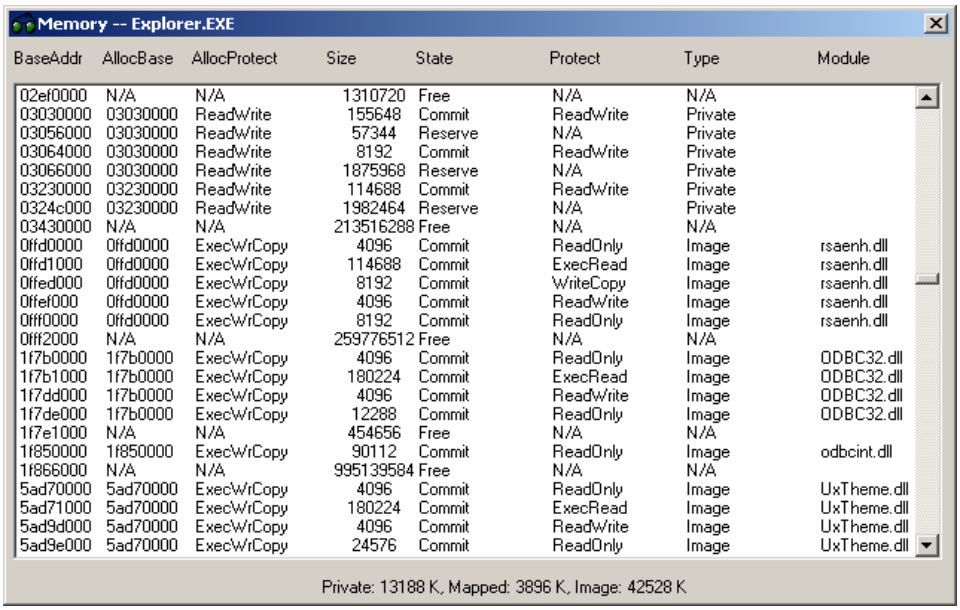
می‌بینید که لیست کلیهٔ Thread ها به همراه module سازندهٔ هر یک آورده شده و وضعیت فعلی هر یک از آنها نیز در قسمت State ذکر شده است. با انتخاب Thread موردنظر و کلیک برروی دکمهٔ Kill می‌توانید به روند اجرایی آن خاتمه دهید.

بررسی فضای حافظهٔ برنامه

همان‌طور که می‌دانید هر Process در ویندوز دارای ٤ گیگابایت فضای مجازی حافظه مخصوص به خود است. Dllهایی که توسط برنامه بارگذاری می‌شوند، تماماً به این حافظه نگاشت شده و در حقیقت جزیی از برنامه محسوب می‌شوند.

با توجه به فایل‌های نگاشت شده و سایر اطلاعات موجود، این فضا به تکه‌های گوناگونی با مشخصات متفاوت تقسیم می‌گردد. داشتن دانش کافی در مورد اطلاعات موجود در این فضا و نحوهٔ سازماندهی آنها می‌تواند در مراحل بعدی بخصوص مرحلهٔ Debug بسیار مفید باشد.

در این قسمت با استفاده از نرم‌افزار Process Viewer فضای حافظهٔ explorer.exe را مورد بررسی قرار می‌دهیم. با کلیک راست بر روی explorer.exe در صفحهٔ اصلی این نرم‌افزار و انتخاب گزینهٔ memory، پنجرهٔ memory باز شده و لیستی از نواحی موجود در فضای حافظهٔ explorer.exe را به همراه جزئیات کاملی در مورد هر یک به نمایش می‌گذارد. در شکل (۲-۱۳) این پنجره را به همراه جزییات مربوط به فضای حافظهٔ explorer.exe مشاهده می‌کنید.



شکل (۲-۱۳)

همان‌طور که مشاهده می‌کنید قسمت‌هایی از فضای حافظه به فایل‌های dll مورد استفادهٔ برنامه اختصاص یافته و قسمت‌هایی نیز برای ذخیره‌سازی داده‌های برنامهٔ اصلی رزرو شده‌اند. این داده‌ها

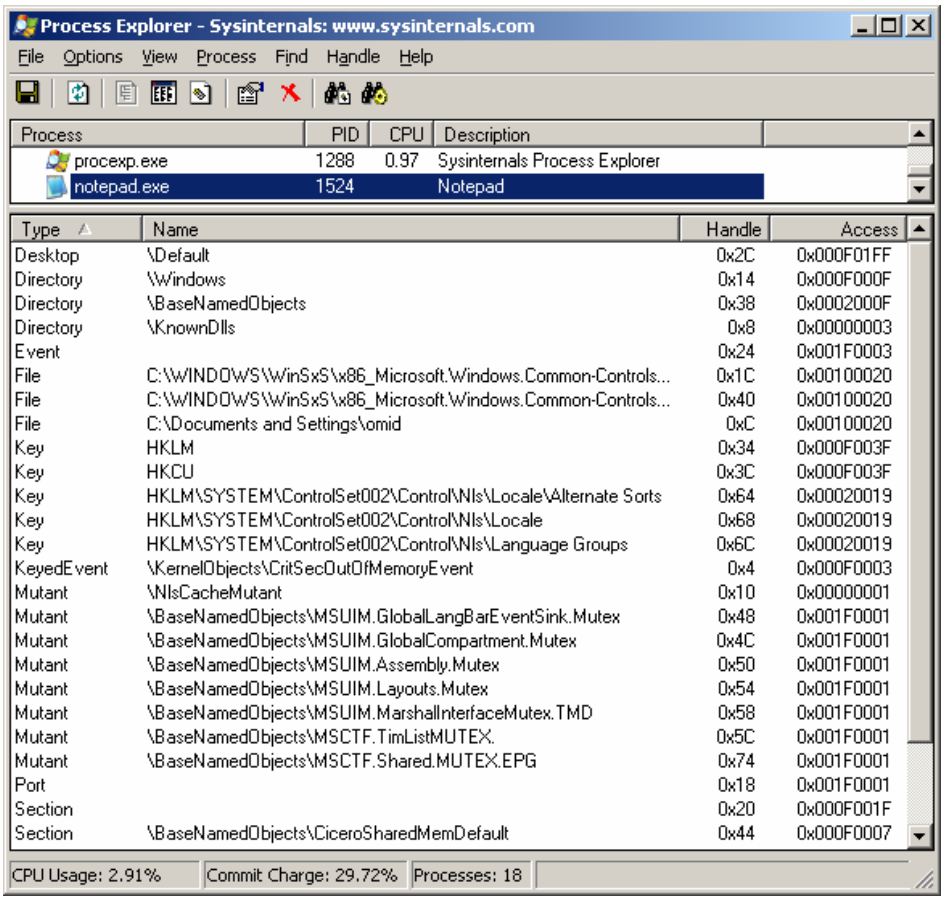
می‌توانند متغیرهای محلی، کد و یا سایر اطلاعات مورد استفادهٔ برنامه باشند. به هر ناحیه از این فضا یک Segment گفته می‌شود که دارای خصوصیات خاص خود است.

بررسی شماره‌های دسترسی

هر Process در ویندوز شماره‌های دسترسی مخصوص به خود را دارد. این شماره‌های دسترسی می‌توانند مربوط به فایل‌ها، Threadها، پورت‌ها، رویدادها و یا سایر موارد باشند. بدیهی است که بررسی جزئیات شماره‌های دسترسی مورد استفادهٔ یک برنامه می‌تواند نکات بسیار جالبی را در مورد نحوهٔ عملکرد و جزئیات پیاده‌سازی آن مشخص کند.

در این قسمت با استفاده از نرم‌افزار Process Explorer شماره‌های دسترسی مورد استفاده توسط برنامهٔ notepad.exe را مورد بررسی قرار می‌دهیم.

با انتخاب notepad.exe در صفحهٔ اصلی و فشردن کلیدهای Ctrl+H ناحیهٔ پایین صفحه لیستی از کلیهٔ شماره‌های دسترسی مورد استفاده توسط این برنامه را به همراه خصوصیات آنها نمایش می‌دهد که در شکل (۲-١٤) آنها را مشاهده می‌کنید.



شکل (۲-۱٤)

همان‌طور که مشاهده می‌کنید اطلاعات مربوط به شماره‌های دسترسی شامل نوع، نام، شماره و نوع دسترسی هستند. در صورت نیاز به اطلاعات بیشتر، می‌توانید برروی شمارهٔ دسترسی موردنظر کلیک راست کرده و گزینهٔ Properties را انتخاب کنید.

بررسی فعالیت‌ها در زمینهٔ فایل

همان‌طور که می‌دانید برنامه‌ها معمولاً از فایل‌ها برای ذخیره‌سازی و نگهداری اطلاعات مورد نیاز خود استفاده کرده و در صورت نیاز با خواندن اطلاعات ذخیره شده، داده‌های مورد نیاز خود را بدست می‌آورند. در سیستم عامل ویندوز توابع کار با فایل وظیفهٔ مدیریت اجزاء دیگری را نیز برعهده دارند که از آن جمله می‌توان به pipeها، پورت‌ها و یا درایورهای سخت‌افزاری اشاره کرد. بدیهی است که زیر نظر داشتن فعالیت‌های یک برنامه در زمینهٔ فایل می‌تواند نکات کلیدی و جالبی را در مورد نحوهٔ عملکرد، و منابع مورد استفادهٔ آن مشخص کند که در مراحل بعدی می‌تواند بسیار حائز اهمیت باشد. در صورت نیاز به اطلاعات دقیق‌تر راجع به نحوهٔ استفاده از فایل‌ها و مدیریت آنها در ویندوز می‌توانید به فصل ۸ مراجعه کنید.

فعالیت‌ها در زمینهٔ فایل معمولاً به چند دسته تقسیم می‌شوند که عبارتند از: باز کردن و یا بستن فایل، خواندن از فایل و نوشتن در فایل که هر کدام بسته به شرایط می‌توانند عملکردهای گوناگونی داشته باشند.

نرم‌افزارهای زیادی وجود دارند که در آنها بررسی فعالیت‌ها در زمینهٔ فایل نقش اساسی دارد که از آن جمله می‌توان به نرم‌افزارهای ضدویروس اشاره کرد. این نرم‌افزارها قبل از انجام هرگونه عملیات برروی فایل، ابتدا آن را مورد بررسی قرار داده و در صورت مشاهدهٔ موارد مشکوک از دسترسی به فایل مربوطه جلوگیری به عمل می‌آورند.

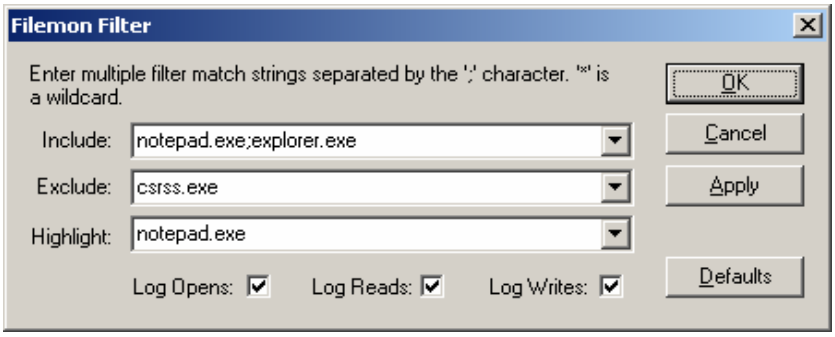
انجام چنین بررسی‌هایی که به Monitoring معروف هستند با استفاده از روش‌های گوناگونی قابل پیاده‌سازی است که از جمله آنها می‌توان به درایورهای مجازی اشاره کرد که در ادامه، مورد بحث قرار می‌گیرند.

نرم‌افزار File Monitor

یکی از قوی‌ترین ابزارها برای بررسی فعالیت‌ها در زمینهٔ فایل، نرم‌افزار File Monitor است که از قابلیت‌های منحصر به فردی در این زمینه برخوردار می‌باشد. این نرم‌افزار کلیهٔ فعالیت‌های یک Process و dllهای زیرمجموعهٔ آنرا به همراه اطلاعات دقیقی در مورد هر فعالیت به نمایش می‌گذارد.

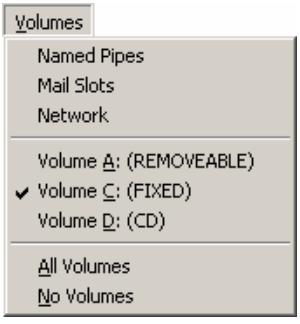
به‌طور معمول با اولین اجرای این نرم‌افزار هیچ فیلتری برای Processهای مورد بررسی وجود ندارد. با تنظیم این فیلترها می‌توانید از نمایش فعالیت‌های Processهای دلخواه جلوگیری کرده و یا Processهای دیگری را به لیست مورد بررسی اضافه کنید. برای تعیین فیلترها می‌توانید برروی

دکمهٔ آن در نوار ابزار کلیک کرده و یا کلیدهای Ctrl+L را فشار دهید. همان‌طور که در شکل (۲-۱۵) مشاهده می‌کنید با انجام این عمل پنجرهٔ Filter نمایش داده شده و به شما امکان تعیین فیلترهای مورد نظر را می‌دهد.



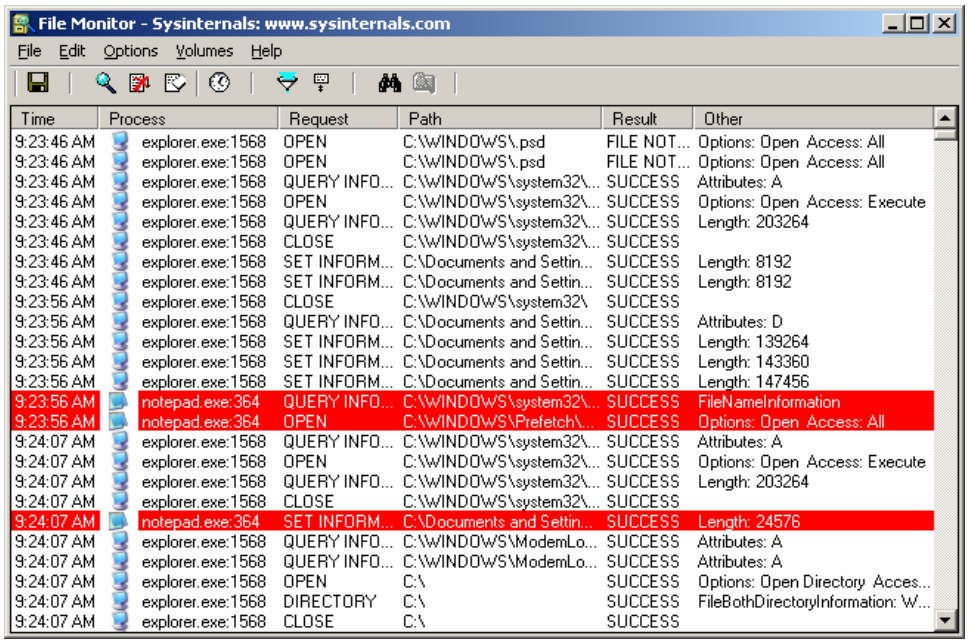
شکل (۲-۱۵)

به‌منظور تطابق هرچه بیشتر اطلاعات نمایش داده شده با موارد موردنظر، این نرم‌افزار امکان محدود کردن بررسی‌ها را برروی درایوهای خاصی از کامپیوتر شما دارد. به عنوان مثال می‌توانید فقط فعالیت‌هایی را مورد بررسی قرار دهید که برروی درایو CD شما اتفاق می‌افتد. با استفاده از منوی Volumes از صفحهٔ اصلی می‌توانید بررسی‌ها را به درایوهای خاص و یا انواع مختلفی از فایل محدود کنید.



شکل(۲-۱۶)

پس از تنظیم فیلترها و درایوها و سایر موارد با کلیک برروی دکمهٔ Capture از نوار ابزار، نرم‌افزار شروع به نمایش فعالیت‌ها می‌کند به عنوان مثال در شکل (۲-۱۷) این نرم‌افزار با استفاده از فیلترهای مشخص شده مشغول بررسی Process های explorer.exe و notepad.exe است.



شکل (۲-۱۷)

همان‌طور که مشاهده می‌کنید برای هر فعالیت، اطلاعات دقیقی از قبیل زمان، نام Process، نوع، مسیرفایل، نتیجهٔ فعالیت و برخی اطلاعات مفید دیگر مشخص شده است. در صورت نیاز در زمان بررسی‌ها نیز می‌توانید Process خاصی را از لیست حذف کرده و یا به آن اضافه کنید. اطلاعات جمع‌آوری شده می‌توانند ذخیره شده و در مواقع مورد نیاز دوباره مورد بررسی قرار گیرند.

نسخه 6.07این نرم‌افزار در CD ضمیمه موجود می‌باشد

Tools\FileMonitor

بررسی فعالیت‌ها در Registry

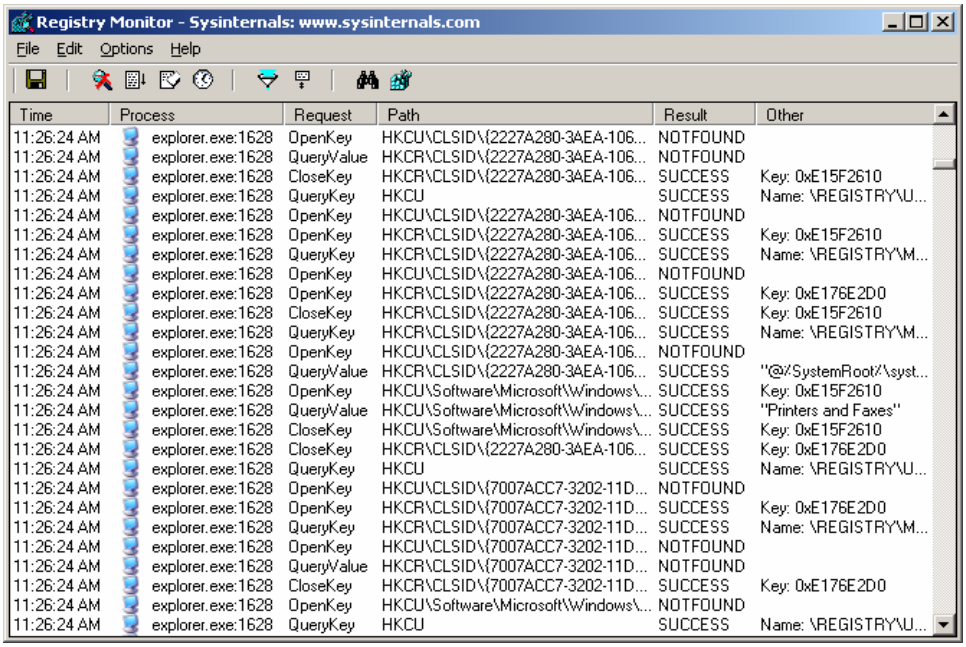
همان‌طور که می‌دانید Registry در ویندوز محلی برای ذخیره‌سازی تنظیمات مربوط به ویندوز و نرم‌افزارهای دیگر است که سیستم عامل وظیفهٔ مدیریت اطلاعات آن را برعهده دارد. نرم‌افزارها در صورت نیاز می‌توانند داده‌های مورد نیاز خود را در این محل ذخیره کرده و در مواقع لزوم آنها را خوانده و مورد استفاده قرار دهند. توابع API متعددی برای مدیریت داده‌ها در Registry وجود دارند که در حقیقت روش استانداردی را برای مدیریت داده‌ها بوجود می‌آورند. این توابع می‌توانند توسط نرم‌افزارهای مختلف و نیز خود سیستم عامل مورد استفاده قرار گیرند.

بررسی فعالیت‌ها در Registry می‌تواند نکات جالب و مفیدی را در مورد داده‌های کنترلی مورد استفاده برنامه مشخص کند. با بررسی و یا تغییر آنها و مشاهدهٔ واکنش‌های برنامه می‌توان به اطلاعات دقیقی راجع به نحوهٔ عملکرد آن دست یافت.

نرم‌افزار Registry Monitor

یکی از قوی‌ترین نرم‌افزارها در زمینهٔ بررسی فعالیت‌ها در Registry نرم‌افزار Registry Monitor است که اطلاعات دقیقی را در مورد جزئیات فعالیت‌هایی که توسط Processها انجام می‌گیرد به نمایش می‌گذارد.

در شکل (۲-۱۸) صفحهٔ اصلی این نرم‌افزار در حال نمایش فعالیت‌های Process ، explorer.exe نشان می‌دهد.



شکل (۲-۱۸)

همان‌طور که می‌بینید اطلاعات نمایش داده شده شامل زمان، نام Process، نوع فعالیت، مسیر، نتیجهٔ فعالیت و برخی از اطلاعات مفید دیگر است. با Double Click بر روی یک سطر، نرم‌افزار Registry Editor ویندوز باز شده و مسیر موردنظر را نمایش داده و به کاربر امکان ایجاد تغییرات را می‌دهد.

این نرم‌افزار در CD ضمیمه موجود می‌باشد

Tools\ Registry Monitor

بررسی فعالیت‌ها در زمینهٔ ارتباطات شبکه

درحال حاضر نرم‌افزارها به‌طور گسترده از شبکه‌های کامپیوتری و اینترنت برای تبادل داده‌ها استفاده می‌کنند. یک نمونه از این موارد نرم‌افزارهایی هستند که از اینترنت برای به روز رسانی امکانات و یا داده‌های خود استفاده می‌کنند. در موارد دیگر، از این اطلاعات به منظور بررسی کدهای ورود، سریال‌ها و اجازه‌های دسترسی نیز استفاده می‌شود.

امروزه ویروس‌های کامپیوتری نیز خود را با امکانات جدید تطابق داده و از پیشرفت‌های حاصل از فن‌آوری شبکه و اینترنت به عنوان ابزاری برای گسترش خود استفاده می‌کنند. احتمالاً به یاد دارید که در زمان سیستم عامل‌هایی مانند Dos انتقال ویروس‌ها معمولاً توسط ابزارهای ذخیره‌سازی اطلاعات مانند دیسک و CD صورت می‌گرفت. این امر باعث می‌شد که مقابله و جلوگیری از گسترش آنها نسبت به امروز از پیچیدگی‌های کمتری برخوردار باشد. در حال حاضر ویروس‌ها از روش‌های متعددی برای انتشار هر چه بیشتر خود استفاده می‌کنند. یکی از قوی‌ترین و خطرناک‌ترین آنها شبکه‌های کامپیوتری هستند. شبکه‌ها این امکان را به ویروس‌ها می‌دهند. که در کسری از ثانیه هزاران کامپیوتر را آلوده کرده و از آنها نیز عنوان سکوهایی برای انتشار هرچه بیشتر خود استفاده کنند.

باتوجه به مطالب ذکر شده، لزوم بررسی فعالیت‌های نرم‌افزارها در زمینهٔ شبکه‌های کامپیوتری به‌منظور کشف و زیر نظر داشتن متدهای ارتباطی آنها کاملاً مشهود است.

در این بخش مراحل بررسی فعالیت‌ها به دو قسمت تقسیم شده‌اند که عبارتند از:

بررسی Connection های مورد استفاده

به‌طور معمول نرم‌افزارهایی که قصد اتصال به شبکه و تبادل اطلاعات را داشته باشند، ابتدا با استفاده از توابع استاندارد موجود در سیستم عامل و تعیین آدرس و مشخصات مقصد موردنظر Connection هایی را ایجاد کرده و سپس به تبادل اطلاعات می‌پردازند.

بررسی Connection های مورد استفادهٔ یک نرم‌افزار و مقاصد هر یک می‌تواند نکات کلیدی را در مورد متدهای ارتباطی آن مشخص کند. با استفاده از بررسی‌های دقیق‌تر می‌توان به علت هر یک از این ارتباطات نیز پی‌برد.

نرم‌افزار Netstat

نرم‌افزاری معروف برای تهیهٔ لیستی از Connectionهای فعلی است که به همراه سیستم عامل‌های ویندوز و لینوکس ارائه می‌شود. این نرم‌افزار لیستی از Connection ها را به همراه جزئیات کاملی در مورد هر یک به نمایش می‌گذارد. این نرم‌افزار از نوع Console بوده و پارامترهای خاص خود را دارد که با استفاده از سویچ ?/. می‌توانید لیستی از پارامترهای مورد استفاده را به همراه توضیحات کوتاهی در مورد هر یک مشاهده کنید.

به‌عنوان مثال درشکل (۲-۱۹) این نرم‌افزار را درحال نمایش لیست Connectionها مشاهده می‌کنید.

```
netstat -o

Active Connections

Proto  Local Address   Foreign Address               State         PID

TCP    omidpc:3214     cs57.msg.dcn.yahoo.com:5050   ESTABLISHED   2696
TCP    omidpc:3257     www.sony.com:http             ESTABLISHED   3448
TCP    omidpc:3258     www.sony.com:http             ESTABLISHED   3448
TCP    omidpc:3259     www.sony.com:http             ESTABLISHED   3448
TCP    omidpc:3260     www.sony.com:http             ESTABLISHED   3448
TCP    omidpc:3220     origin2.microsoft.com:http    ESTABLISHED   2788
TCP    omidpc:3274     nameservices.net:http         ESTABLISHED   3568
TCP    omidpc:3276     nameservices.net:http         ESTABLISHED   3568
TCP    omidpc:3277     nameservices.net:http         ESTABLISHED   3568
TCP    omidpc:3278     nameservices.net:http         ESTABLISHED   3568
TCP    omidpc:3239     kundenserver.de:http          LAST_ACK      2864
TCP    omidpc:3241     unknown.Level3.net:http       ESTABLISHED   3008
TCP    omidpc:3267     64.154.80.250:http            LISTEN        3448
TCP    omidpc:3272     212.143.22.80:http            LISTEN        3568
TCP    omidpc:3273     g.websponsors.com:http        ESTABLISHED   3568
```

شکل (۲-۱۹)

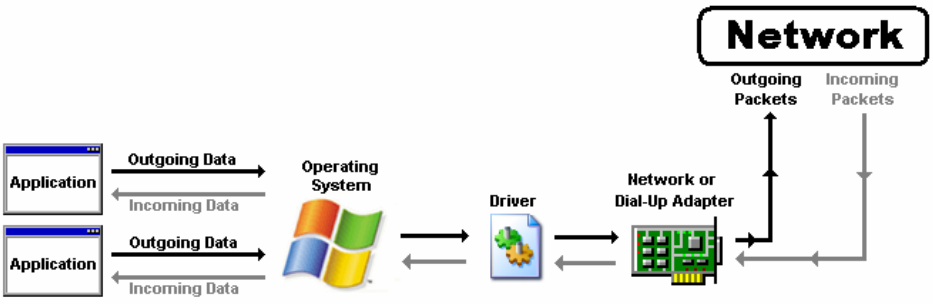
همان‌طور که مشاهده می‌کنید اطلاعات نمایش داده شده شامل نوع پرتکل، آدرس و شمارهٔ پورت کامپیوتر محلی، مقصد و وضعیت Connection است. در صورت نیاز می‌توانید با استفاده از سویچ o- به شمارهٔ Process ایجادکنندهٔ Connection نیز دسترسی داشته باشید.

بررسی تبادل داده‌ها در شبکه

در قسمت قبل در مورد نحوهٔ بررسی Connection ها نکاتی را آموختید. در این بخش قصد داریم داده‌های مبادله شده توسط نرم‌افزارها را مورد بررسی قرار دهیم.

کنترل داده‌های مبادله شده توسط نرم‌افزارها بخش اصلی بررسی فعالیت‌ها در زمینهٔ شبکه را تشکیل می‌دهد. این بررسی‌ها می‌تواند اطلاعات مفیدی را راجع به نحوهٔ ارتباط، پرتکل‌های مورد استفاده و داده‌های مبادله شده توسط آنها مشخص کند.

همان‌طور که می‌دانید معمولاً داده‌ها در شبکه‌های کامپیوتری ابتدا به بسته‌هایی (Packet) تبدیل شده و سپس مبادله می‌شوند. بدیهی است که هر بسته علاوه بر داده‌ها حاوی یک سری اطلاعات کنترلی از قبیل نوع پرتکل، آدرس مبدا، آدرس مقصد و کدهای کنترل خطا نیز می‌باشد. کنترل بسته‌های مبادله شده توسط نرم‌افزارها در یک کامپیوتر به‌طور معمول ممکن نیست زیرا سیستم عامل چنین اطلاعاتی را در اختیار نرم‌افزارها قرار نمی‌دهد. در شکل (۲-۲۰) روند کلی تبادل اطلاعات را در شبکه از دید یک کامپیوتر مشاهده می‌کنید.

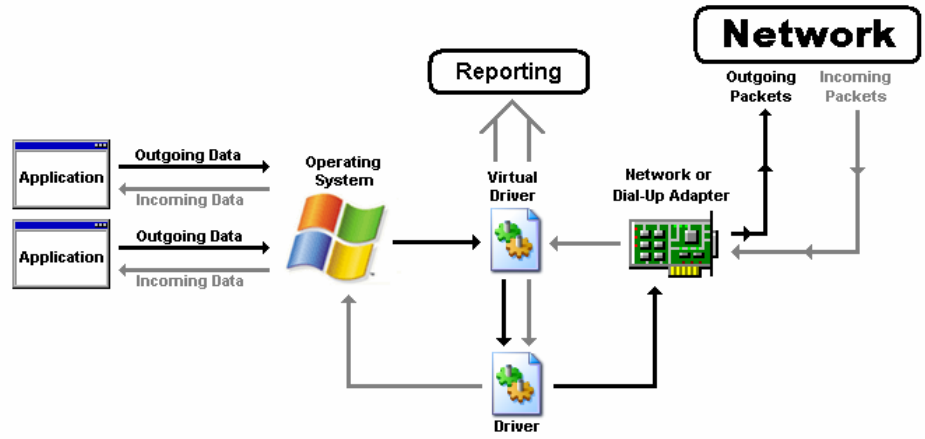


شکل (۲-۲۰)

همان‌طور که در شکل مشاهده می‌کنید، داده‌های فرستاده شده توسط برنامه ابتدا به سیستم عامل منتقل می‌شود. در مرحلهٔ بعد سیستم عامل عملیاتی را برروی آنها انجام داده و سپس آنها را به درایور سخت‌افزاری کارت شبکه یا مودم می‌فرستد. درایور با سخت‌افزار ارتباط برقرار کرده و بسته‌ها را بوسیلهٔ آن به شبکهٔ محلی و یا اینترنت می‌فرستد. مراحل دریافت اطلاعات از شبکه نیز مشابه مرحلهٔ قبل است با این تفاوت که روند دریافت از سخت‌افزار شبکه شروع شده و مراحل را بصورت معکوس تا رسیدن اطلاعات به نرم‌افزار موردنظر طی می‌کند.

درایورهای مجازی

به‌طور معمول برای گزارش‌گیری‌های دقیق و کامل از عملکرد نرم‌افزارها از درایورهای مجازی استفاده می‌شود به این صورت که درایور مجازی جایگزین درایور اصلی شده و بر تبادل داده‌ها نظارت کامل خواهد داشت. در حقیقت درایور مجازی به‌عنوان واسطی بین درایور حقیقی، سخت‌افزار و سیستم عامل عمل می‌کند. در شکل (۲-۲۱) نحوهٔ عملکرد درایورهای مجازی را مشاهده می‌کنید.



شکل (۲-۲۱)

همان‌طور که در شکل مشاهده می‌کنید، درخواست‌های سیستم عامل ابتدا به درایور مجازی فرستاده می‌شود. درایور مجازی اطلاعات دریافتی را مورد بررسی قرار داده و در صورت نیاز گزارشی از آنها تهیه می‌کند. در مرحلهٔ بعد درایور مجازی درخواست‌های رسیده را به درایور حقیقی منتقل کرده و روند ارسال اطلاعات دقیقاً مانند حالت عادی دنبال می‌شود.

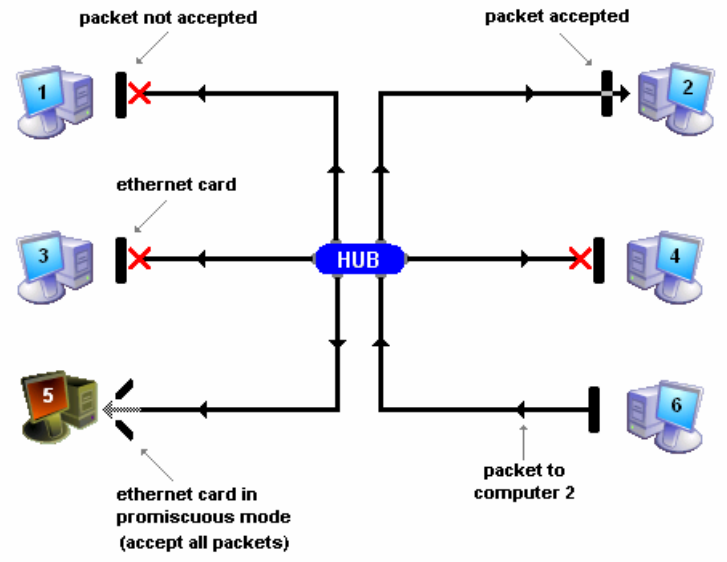
حالت بی‌قید (Promiscuous) در کارت‌های شبکه

همان‌طور که گفته شد، اطلاعات در شبکه‌های کامپیوتری معمولاً ابتدا به بسته‌هایی تبدیل شده و سپس فرستاده می‌شوند. بدیهی است که این بسته‌ها علاوه بر اطلاعات و داده‌های اصلی حاوی یک سری اطلاعات کنترلی از قبیل آدرس مبدأ و مقصد نیز می‌باشند.

در شبکه‌های محلی معمولی بسته‌های فرستاده شده توسط یک کامپیوتر علاوه بر کامپیوتر مقصد از طرف تمام کامپیوترهای دیگر موجود در شبکه نیز قابل رؤیت هستند. کارت‌های شبکه به‌طور معمول دارای بخشی به نام MAC (Media Access Control) هستند. وظیفه این بخش بررسی بسته‌های رسیده و قبول بسته‌هایی است که آدرس مقصدشان برابر آدرس داده شده به کارت شبکه باشد.

ولی اکثر کارت‌های شبکه حالتی به نام حالت بی‌قید دارند. در این حالت MAC کنترلی را برروی بسته‌های رسیده انجام نداده و تمام آنها را قبول می‌کند درنتیجه تمام بسته‌های مبادله شده توسط کامپیوترهای موجود در شبکه، توسط کارت شبکهٔ مذکور نیز پذیرفته شده و به درایور کارت شبکه و سیستم عامل فرستاده می‌شوند. با استفاده از درایورهای مجازی می‌توان بررسی‌های کاملی برروی این بسته‌ها انجام داده و در صورت نیاز آنها را ذخیره کرد. با استفاده از این روش می‌توانید کنترل کاملی برروی کلیه داده‌های مبادله شده در شبکه داشته باشید.

در شکل (۲-۲۲) روش تبادل داده‌ها و حالت بی‌قید در کارت‌های شبکه را مشاهده می‌کنید.



شکل (۲-۲۲)

همان‌طور که در شکل مشخص است، بستهٔ فرستاده شده علاوه بر مقصد که کامپیوتر شمارهٔ ۲ است، از طرف کارت شبکهٔ کامپیوتر شمار ۵ نیز مورد پذیرش قرار می‌گیرد. توجه داشته باشید که سایر کارت‌های موجود در شبکه از پذیرفتن بستهٔ موردنظر خودداری می‌کنند زیرا آدرس مقصد آن با آدرس اختصاص داده شده به آنها مطابقت ندارد.

Sniffer ها

Snifferها ابزارهایی برای جمع‌آوری و بررسی تبادل داده‌ها در شبکه هستند که معمولاً از حالت بی‌قید و درایورهای مجازی به منظور ایجاد کنترل کامل برروی کلیهٔ بسته‌های مبادله شده در شبکه استفاده می‌کنند. به‌طور معمول Snifferها دارای قسمتی به‌نام تحلیل‌گر پرتکل (Protocol Analyzer) هستند که وظیفهٔ شناسایی نوع پرتکل مورد استفاده در بسته و تفسیر لایه‌ها، اطلاعات کنترلی و داده‌های موجود در آن را برعهده دارد.

به دلیل حجم انبوه بسته‌های مبادله شده در شبکه‌ها، اکثر Sniffer ها مکانیزم‌های مختلفی را برای فیلتر کردن بسته‌های رسیده به کار می‌گیرند. با به‌کارگیری این Filter ها می‌توانید کنترل کاملی برروی بسته‌های دلخواه خود داشته و از بررسی سایر بسته‌های مبادله شده که مورد نظر شما نیستند جلوگیری کنید. روش استفاده از Filter ها در Sniffer های مختلف متفاوت است که در ادامه آنها را مورد بررسی قرار خواهیم داد.

ابزارهای Libpcap و Winpcap

ابزارهایی برای طراحی نرم‌افزارهای Sniffer هستند که شامل درایورهای مجازی و تعدادی توابع کتابخانه‌ای برای دریافت بسته‌های مبادله شده می‌باشند. این ابزار ها می‌توانند توسط نرم‌افزارهای مختلف مورد استفاده قرار بگیرند. در حقیقت این ابزارها بستر مناسبی را برای طراحی و ایجاد نرم‌افزارهای Sniffer مهیا کرده و روند ساخت چنین نرم‌افزارهایی را تسهیل می‌کند.

Winpcap و توابع کتابخانه‌ای آن برای کار در سیستم عامل ویندوز تهیه شده‌اند. این ابزار برروی کلیه ویندوزهای خانوادهٔ 9X و NT قابل نصب است.

نسخه 3.1 این نرم‌افزار در CD ضمیمه موجود می‌باشد

Tools\Winpcap

Libpcap قابلیت‌های مشابهی را برای لینوکس ارائه می‌دهد که می‌توانند در ساخت Sniffer ها در این سیستم عامل به کار گرفته شود.

نرم‌افزار Ethereal

یک Protocol Analyzer و Sniffer بسیار قوی و با امکانات منحصر به فرد است. این ابزار با استفاده از قابلیت‌های موجود می‌تواند کنترل کاملی را برروی بسته‌های مبادله شده در شبکه داشته باشد.

نسخه 0.10.4این نرم‌افزار در CD ضمیمه موجود می‌باشد

Tools\Ethreal

در ادامه مروری برروی برخی از مشخصات و قابلیت‌های این نرم‌افزار خواهیم داشت:

این نرم‌افزار Open Source بوده و امکان اضافه کردن قابلیت‌های جدید و یا تغییر قابلیت‌های موجود را به کاربر می‌دهد. همین امر سبب توسعه و بهبود روزافزون این نرم افزار گشته است.

نسخه‌های مختلفی از Ethereal برای سیستم عامل‌های مختلف وجود دارد.

کنترل بسته‌های مبادله شده می‌تواند از طریق سخت‌افزارهای مختلفی مانند کارت‌های شبکه، مودم‌ها و... صورت گیرد.

این نرم‌افزار توانایی تحلیل و بررسی بسته‌های ذخیره شده توسط طیف وسیعی از دیگر نرم‌افزارها را نیز دارا می‌باشد که از جمله می‌توان به MS Network Monitor , tcpdump و Sun Snoop اشاره کرد.

این نرم‌افزار می‌تواند اطلاعات و بسته‌های مبادله شده را با استانداردهای برخی از نرم‌افزارهای دیگر از جمله tcpdump و Sun snoop ذخیره کند. این اطلاعات ذخیره شده می‌توانند برای تحلیل و بررسی مجدد توسط این نرم‌افزارها به کار گرفته شوند.

یکی از قابلیت‌های مهم این نرم‌افزار استفاده از سیستم Filtering بسیار قوی و قابل انعطاف است. با استفاده از این امکان کاربر می‌تواند کنترل کاملی برروی بسته‌های موردنظر خود داشته باشد.

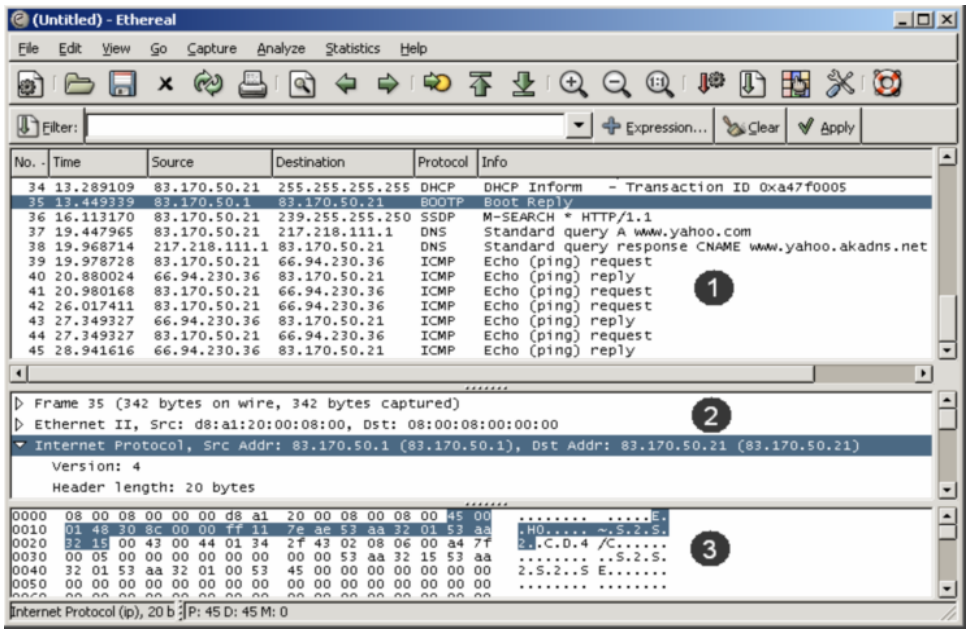
این نرم‌افزار توانایی شناسایی و تحلیل بیش از 18,000 پرتکل مختلف را دارد و با توجه به Open Source بودن، کاربر می‌تواند برحسب نیاز پرتکل‌های جدیدی را به آن اضافه کرده و در اختیار سایر کاربران قرار دهد.

باتوجه به موارد ذکر شده، اطلاعات دقیقی راجع به امکانات و مشخصات این نرم‌افزار بدست آوردید. حال برخی از کاربردهای این نرم‌افزار در قالب مثال‌هایی نشان داده خواهند شد. توجه

داشته باشید که تصاویر ارائه شده از نسخهٔ 0.10.4 این نرم‌افزار گرفته شده است. این نسخه احتمالاً با نسخه‌های دیگر متفاوت خواهد بود.

با توجه به استفادهٔ این نرم‌افزار از ابزارهای Winpcap (در ویندوز) و Libpcap (در Linux) ، قبل از نصب لازم است این ابزارها نیز تهیه و نصب شوند.

در شکل (۲-۲۳) صفحهٔ اصلی این نرم‌افزار را مشاهده می‌کنید:



شکل (۲-۲۳)

همان‌طور که مشاهده می‌کنید ، صفحه به سه قسمت اصلی تقسیم شده است:

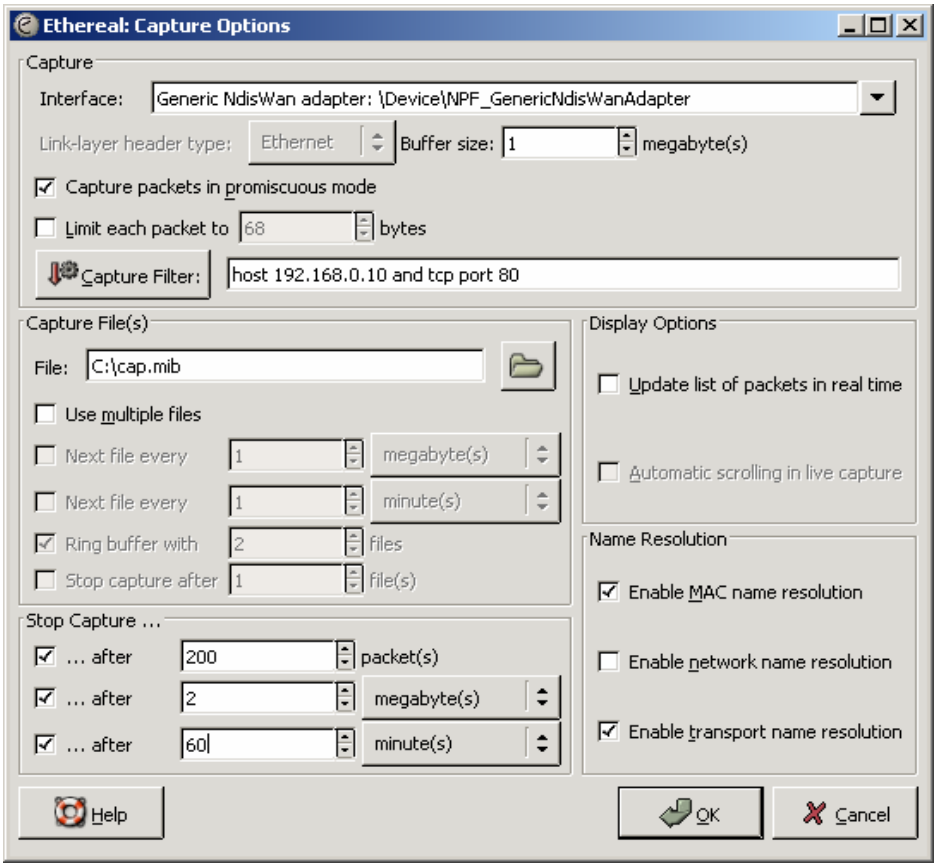
۱- این قسمت لیستی از بسته‌های مبادله شده را به همراه اطلاعات مختصر و مفیدی راجع به هر یک به نمایش می‌گذارد. با کلیک برروی هر بسته، بخش‌های ۲ و ۳ با اطلاعات مربوط به آن بسته به روز خواهند شد.

۲- این قسمت، اطلاعات دقیق و کاملی را راجع به رکوردهای کنترلی و داده‌های موجود در یک بسته نمایش می‌دهد. این اطلاعات می‌تواند به منظور بررسی‌های کامل‌تر و دقیق‌تر به کار گرفته شود. با کلیک بر روی هر قسمت از ساختار درختی نمایش داده شده برای هر بسته، اطلاعات خام آن قسمت در بخش سوم انتخاب شده و متمایز می‌شود.

۳- همان‌طور که ذکر شد در این قسمت اطلاعات خام مربوط به هر بسته نمایش داده می‌شود.

ضبط کردن و مدیریت بسته‌های مبادله شده

به منظور شروع ضبط تبادل بسته‌ها ، از منوی Capture گزینهٔ Start را انتخاب کنید با این کار صفحهٔ Capture Options همانند شکل (۲-۲٤) ظاهر شده و گزینه‌های موجود برای عملیات ضبط را نمایش می‌دهد.



شکل (۲-۲٤)

در زیر، برخی از گزینه‌های مهم این پنجره تشریح شده‌اند.

Capture / Interface

در این قسمت از پنجرهٔ Capture Options ، سخت‌افزار مورد استفاده برای ضبط انتخاب می‌شود که معمولاً یک کارت شبکه یا مودم است. توجه داشته باشید برای ضبط اطلاعات مبادله شده توسط مودم‌ها از گزینهٔ Generic Adapter استفاده می‌شود. این گزینه به صورت پیش فرض وجود دارد.

Capture / Capture Filter

در این قسمت فیلترهایی برای محدود کردن بسته‌های ضبط شده تعیین می‌شود. این فیلترها با استفاده از زبان خاصی تعریف شده و توسط Winpcap / Libpcap برروی بسته‌های مبادله شده اعمال می‌شوند.

به منظور ایجاد آشنایی نسبی با این فیلترها و قابلیت‌های آنها به مثال‌های ذکر شده توجه کنید:

Host 202.2.2.0

ضبط تمام بسته‌های ارسال شده و یا دریافت شده توسط آدرس IP ، 202.2.2.0.

Tcp port 80

ضبط بسته‌های مبادله شده توسط پورت 80 (http) از تمام کامپیوترهای موجود در شبکه.

Host omidpc and tcp port 80

ضبط بسته‌های ارسال شده و یا دریافت شده توسط پورت 80 از کامپیوتری با نام omidpc در شبکه.

IP proto UDP

ضبط بسته‌های UDP مبادله شده در شبکه.

Not host omidpc

ضبط تمام بسته‌های مبادله شده در شبکه به غیر از آنهایی که توسط کامپیوتر omidpc ارسال و یا دریافت می‌شوند.

توجه: *درصورت استفاده از نام کامپیوتر به‌جای IP درصورتی که کامپیوتر مذکور دارای چندین IP اختصاصی باشد، تمام آنها بررسی شده و درشرط اعمال می‌شوند.*

به‌منظور آشنایی دقیق‌تر و کامل‌تر با نحوهٔ تعریف و گرامر این فیلترها می‌توانید به مستندات این نرم‌افزار مراجعه کنید.

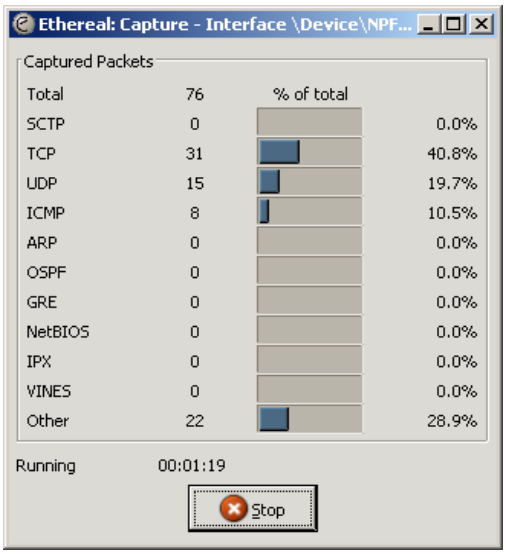
Capture File(s)

در این قسمت آدرس و نحوهٔ تشکیل فایل و یا فایل‌های مورد نیاز برای ذخیرهٔ بسته‌های مبادله شده مشخص می‌شود.

Stop Capture

در این قسمت محدودیت‌های خاصی از نظر زمانی، حجمی و یا تعداد بسته‌های دریافت شده برای عملیات ضبط تعیین می‌شود. به‌عنوان مثال می‌توانید تعیین کنید که عملیات ضبط پس از دریافت تعداد معینی از بسته‌ها به پایان برسد. توجه داشته باشید که در صورت انتخاب چندین محدودیت، هر کدام که زودتر اتفاق بیفتد باعث خاتمهٔ عملیات ضبط خواهد شد.

پس از تأیید گزینه‌های Capture ، نرم‌افزار شروع به ضبط بسته‌های مبادله شده در شبکه کرده و روند عملکرد خود را در صفحهٔ Capture همانند شکل (۲-۲۵) نشان می‌دهد.

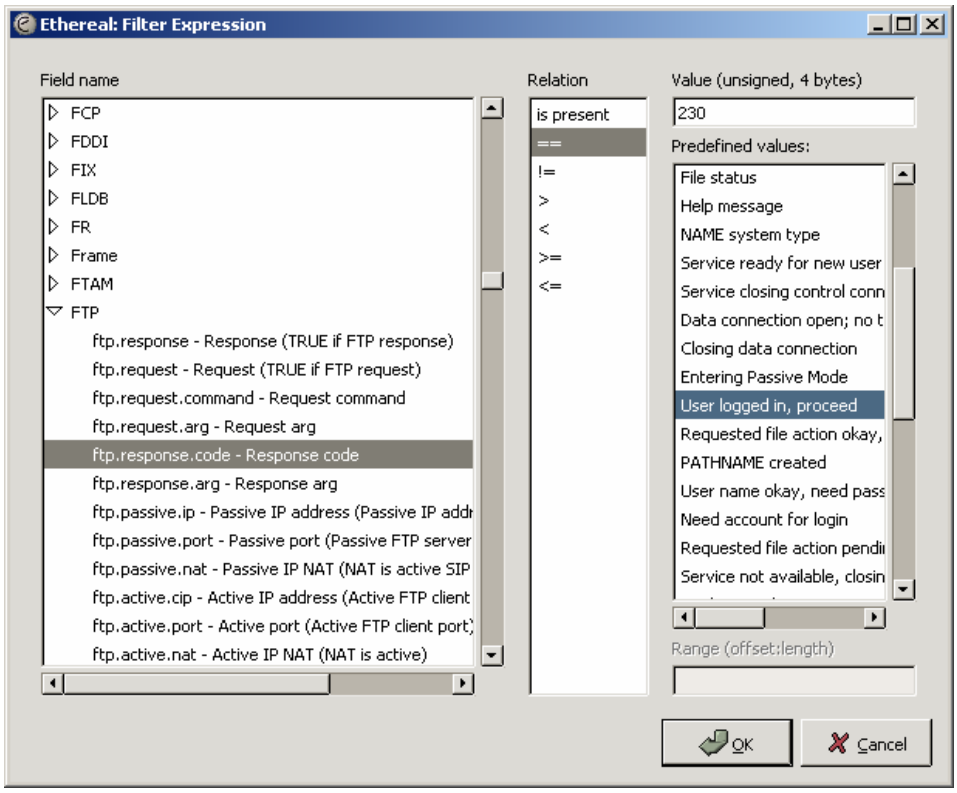


شکل (۲-۲۵)

همان‌طور که مشاهده می‌کنید این صفحه اطلاعات آماری راجع به تعداد و درصد بسته‌های رسیده با توجه به پرتکل آنها را ارائه می‌دهد. توجه داشته باشید که بسته‌های فیلتر شده توسط فیلترهای تعیین شده در مرحلهٔ قبل در این اطلاعات آماری وارد نشده و مورد محاسبه قرار نمی‌گیرند.

با اتمام عملیات ضبط به‌صورت دستی و یا اتفاق افتادن یکی از شروط تعیین شده در مرحلهٔ قبل، صفحهٔ اصلی ظاهر شده و بسته‌های ضبط شده را نمایش می‌دهد. در این مرحله نیز کاربر می‌تواند فیلترهایی را برروی بسته‌های نمایش داده شده اعمال نماید. توجه داشته باشید که گرامر و روش به‌کارگیری این فیلترها با فیلترهایی که در قسمت Capture Options ذکر شد، متفاوت است. خوشبختانه برای به‌کارگیری این فیلترها نیازی به آشنایی با گرامر خاصی نیست زیرا Ethereal سیستم ساده‌ای را برای ایجاد آنها درنظر گرفته است.

با کلیک برروی دکمهٔ Expression+ از نوار ابزار می‌توانید شرط و یا شروط خاصی را برای نمایش بسته‌ها تعیین کنید. با کلیک برروی این دکمه، صفحهٔ Filter Expression ظاهر می‌شود که در حقیقت محیطی برای ایجاد فیلترها است در شکل (۲-۲۶) این صفحه را در حال ایجاد فیلتری برای بسته‌های FTP مشاهده می‌کنید.



شکل (۲-۲۶)

با کلیک برروی دکمهٔ Ok، فیلتر مورد نظر به‌صورت یک رشتهٔ متنی ایجاد شده و در قسمت Filter قرار می‌گیرد. با استفاده از روش فوق شما می‌توانید فیلترهای متعددی ایجاد کرده و در صورت نیاز آنها را بوسیلهٔ عملگرهای and و or با یکدیگر ادغام کنید. در صورت نیاز برای تعیین تقدم عملگرها می‌توانید از پرانتزها نیز استفاده کنید.

برای آشنایی دقیق‌تر با نحوهٔ عملکرد این نرم‌افزار می‌توانید به راهنمای الکترونیکی آن که در CD ضمیمه موجود است، مراجعه کنید.

بررسی فعالیت‌ها در زمینهٔ پورت‌های سخت‌افزاری

همان‌طور که می‌دانید نرم‌افزارها و سخت‌افزارها به‌طور گسترده‌ای از پورت‌های سخت افزاری به‌منظور تبادل داده‌ها و اطلاعات خود استفاده می‌کنند. این داده‌ها نقش کلیدی در همگام‌سازی و نحوهٔ عملکرد سخت‌افزار و نرم‌افزار مربوطه ایفا می‌کنند. داشتن اطلاعات کامل و دقیق در مورد داده‌ها و اطلاعات مبادله شده از طریق این پورت‌ها و بررسی متدها و پرتکل‌های مورد استفاده می‌تواند نکات کلیدی و مفیدی را راجع به عملکرد سخت‌افزارها، و نرم‌افزارهای مرتبط با این سخت‌افزارها مشخص کند.

به‌عنوان مثال کشف پرتکل‌های ارتباطی بین دو سخت‌افزار و یا بین یک نرم‌افزار و یک سخت‌افزار از مواردی است که بسیار متداول بوده و در زمینهٔ مهندسی معکوس در رشتهٔ الکترونیک و سخت‌افزار نقش کلیدی را بازی می‌کند.

به‌طور معمول بررسی اطلاعات مبادله شده توسط پورت‌ها بوسیلهٔ ناظرهای سخت‌افزاری و نرم‌افزاری انجام می‌گیرد. این ناظرها در مسیر مبادلهٔ اطلات قرار گرفته و بر کلیهٔ داده‌های مبادله شده نظارت کامل دارند و در صورت نیاز می‌توانند گزارش کاملی از این فعالیت‌ها تهیه کرده و یا تغییر خاصی را بر روی داده‌های مبادله شده اعمال نمایند.

ناظرهای سخت‌افزاری

این ناظرها معمولاً در مسیر اتصال بین دو پورت قرار گرفته و یک کپی از داده‌ها و اطلاعات مبادله شده را در حافظهٔ خود ذخیره می‌کنند. این اطلاعات معمولاً توسط یک تحلیل‌گر نرم‌افزاری مورد بررسی قرار می‌گیرد. به علت عملکرد مستقل، این ناظرها از قدرت و قابلیت اطمینان بالایی برخوردار هستند.

در شکل (۲-۲۷) یک نمونه از این ناظرها و نرم‌افزار تحلیل‌گر آن را مشاهده می‌کنید.

شکل (۲-۲۷)

ناظرهای نرم‌افزاری

معمولاً اینگونه ناظرها با استفاده از درایورهای مجازی پیاده‌سازی شده و عملکردی مشابه Sinfferها دارند. این درایور مجازی بر اطلاعات مبادله شده نظارت کامل دارد و در صورت نیاز می‌تواند از داده‌های مبادله شده گزارش تهیه کند. در مرحلهٔ بعد این گزارش‌ها با فرمت‌های دلخواه ذخیره شده و یا نمایش داده می‌شوند.

در این قسمت دو نرم‌افزار برای نظارت بر پورت‌های سریال و USB معرفی می‌شوند که از کارآیی و قابلیت اطمینان بالایی برخوردار هستند.

پورت‌های سریال

نرم‌افزار Serial Monitor

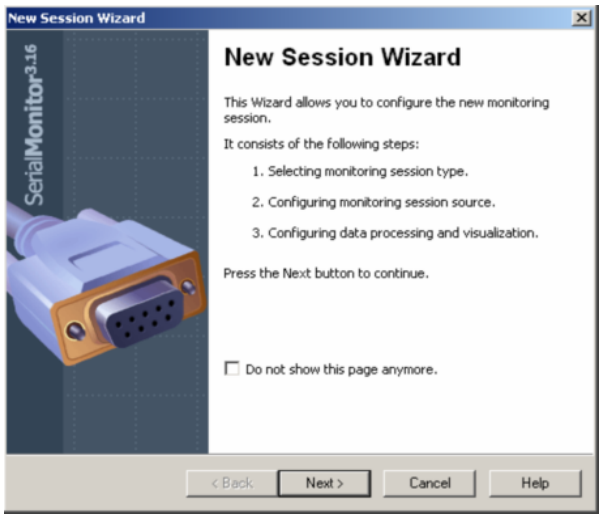
یکی از قوی‌ترین ناظرهای نرم‌افزاری برای نظارت بر پورت‌های سریال است که با امکانات وسیع خود می‌تواند بر روی نحوهٔ عملکرد و داده‌های مبادله شده توسط پورت‌های سریال نظارت دقیق و کاملی داشته باشد. یکی دیگر از قابلیت‌های بسیار مفید این نرم‌افزار توانایی آن در ضبط و ذخیره کردن کلیهٔ رویدادهای اتفاق افتاده برای یک پورت در یک محدودهٔ زمانی مشخص و انجام دوبارهٔ آنها در صورت نیاز است. با این عمل درایور مجازی کلیهٔ رویدادها را دوباره و با توجه به فواصل زمانی بین آنها به‌صورت مجازی ایجاد می‌کند. از نظر نرم‌افزارها و سخت‌افزارها این رویدادهای مجازی، هیچ تفاوتی با رویدادها و داده‌های اصلی ندارند. در نتیجه می‌توانند برای شبیه‌سازی عملکرد یک سخت‌افزار و یا یک نرم‌افزار به کار گرفته شوند.

نسخه 3.16 این نرم‌افزار در CD ضمیمه موجود می‌باشد

Tools\SerialMonitor

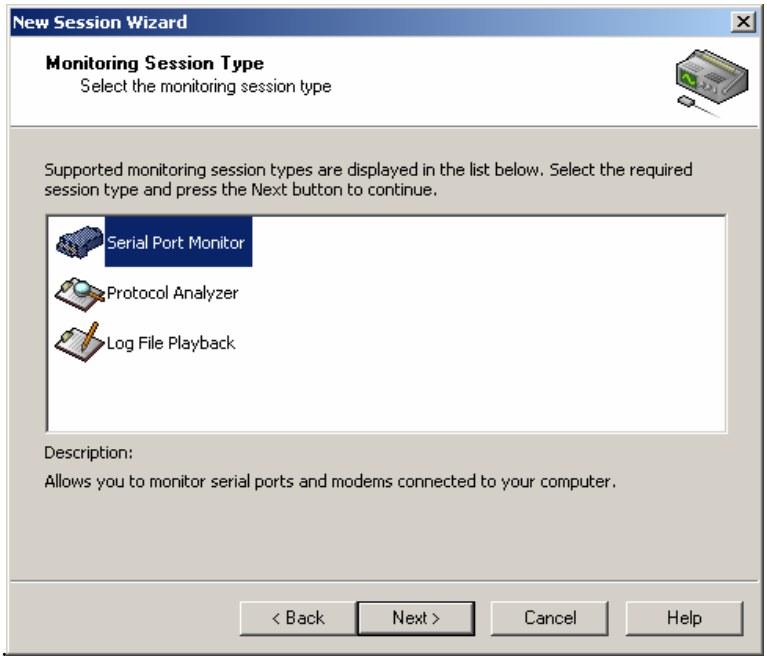
اکنون نحوهٔ عملکرد این نرم‌افزار را مورد بررسی قرار می‌دهیم.

برای شروع کار، بر روی دکمهٔ New Session کلیک کرده و یا کلید F2 را فشار دهید. با این کار صفحهٔ New Session Wizard مانند شکل (۲-۲۸) ظاهر شده و برای شروع کار شما را هدایت می‌کند.



شکل (۲-۲۸)

با کلیک برروی دکمهٔ Next صفحهٔ Monitor Session Type ظاهر می‌شود. در این صفحه نوع عملیات تعیین می‌شود.



شکل(۲-۲۹)

Serial Port Monitor

این مد عملکرد به منظور نظارت و ضبط رویدادها و داده‌های مبادله شده توسط یک پورت سریال به کار گرفته می‌شود. رویدادهای اتفاق افتاده می‌توانند در یک فایل ذخیره شده و در موقع نیاز توسط مد Log File Playback دوباره به‌صورت مجازی ایجاد شوند.

Protocol Analyzer

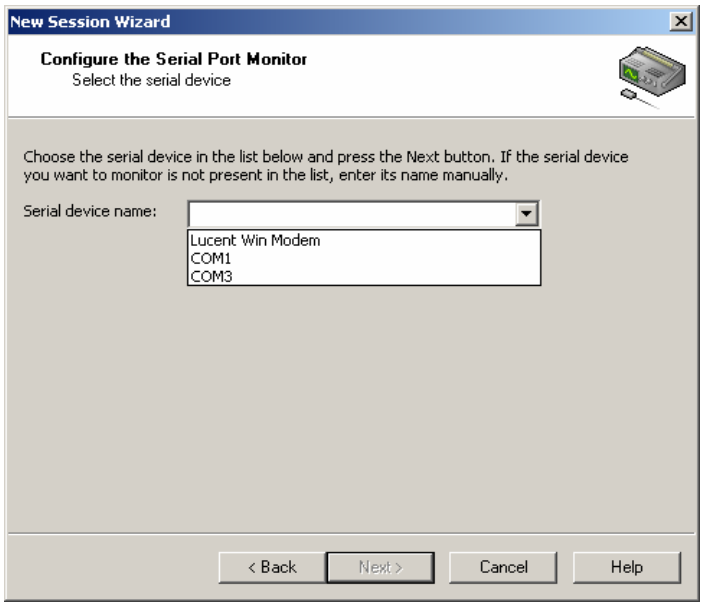
این مد عملکرد در حقیقت برروی تبادل داده‌ها بین دو پورت از کامپیوتر نظارت می‌کند. معمولاً از این مد برای تحلیل پرتکل‌های ارتباطی و نحوهٔ عملکرد آنها استفاده می‌شود.

Log File playback

از این مد به منظور ایجاد دوبارهٔ رویدادهای ضبط شده توسط مد Serial Port Monitor به‌صورت مجازی استفاده می‌شود. همان‌طور که گفته شد از نظر سخت‌افزارها و نرم‌افزارها این رویدادهای مجازی هیچ تفاوتی با رویدادها و داده‌های مبادله شدهٔ اصلی ندارند.

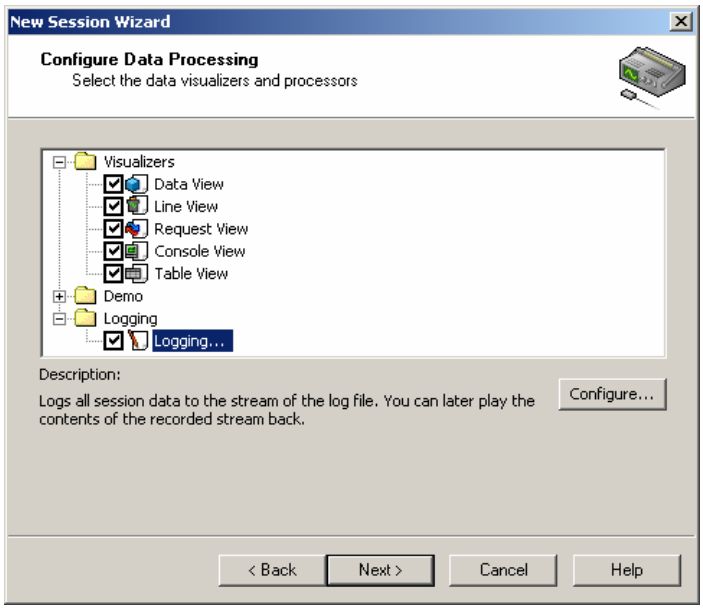
با انتخاب این گزینه و تعیین سرعت و بازهٔ زمانی، نرم‌افزار شروع به ایجاد دوبارهٔ رویدادها با ترتیب ضبط شده می‌کند. بازهٔ زمانی تعیین شده در حقیقت می‌تواند قطعه‌ای از اطلاعات و رویدادهای ضبط شده را به عنوان منبع برای نرم‌افزار معرفی کند. این بازهٔ زمانی بر حسب تاریخ و زمان ضبط اطلاعات تعیین می‌شود.

در این قسمت، مد Serial Port Monitor را انتخاب می‌کنیم. در صفحهٔ بعد پورت موردنظر همانند شکل (۲-۳۰) انتخاب می‌شود.



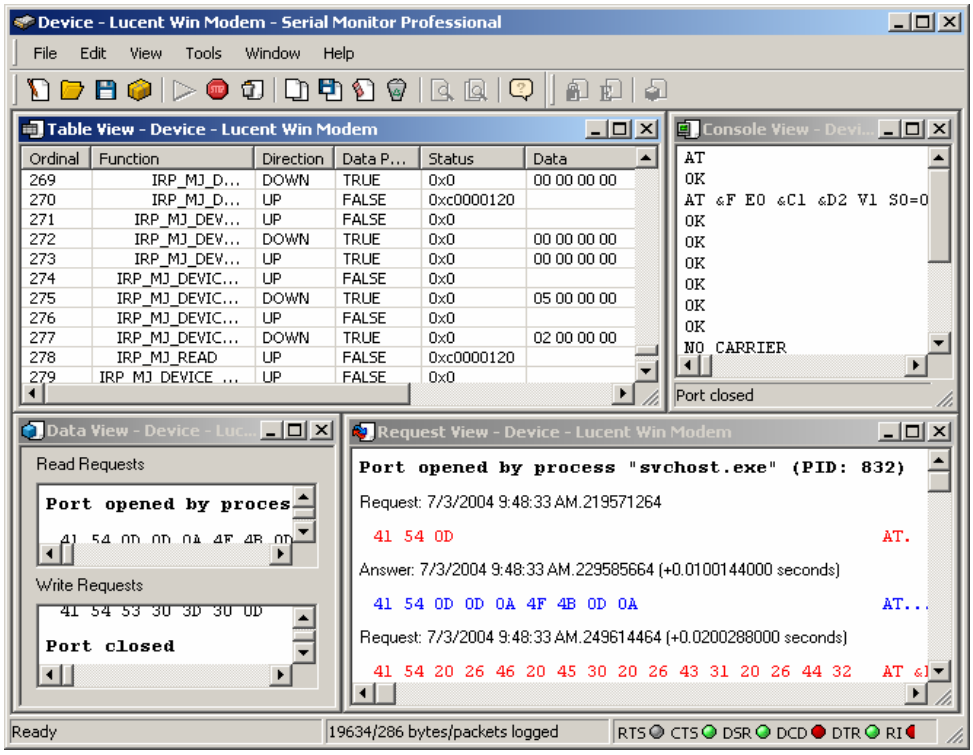
شکل (۲-۳۰)

در صفحهٔ بعد انواع گزارش های مورد نیاز انتخاب شده و در صورت لزوم فایلی برای ذخیره کردن رویدادها تعیین می‌شود.



شکل (۲-۳۱)

با خاتمهٔ مراحل Wizard، نرم‌افزار شروع به ضبط رویدادها، داده‌ها و حالت‌های پورت موردنظر کرده و این اطلاعات را پس از قالب‌بندی در پنجره‌های مربوطه به نمایش می‌گذارد.



شکل (۲-۳۲)

پورت‌های USB

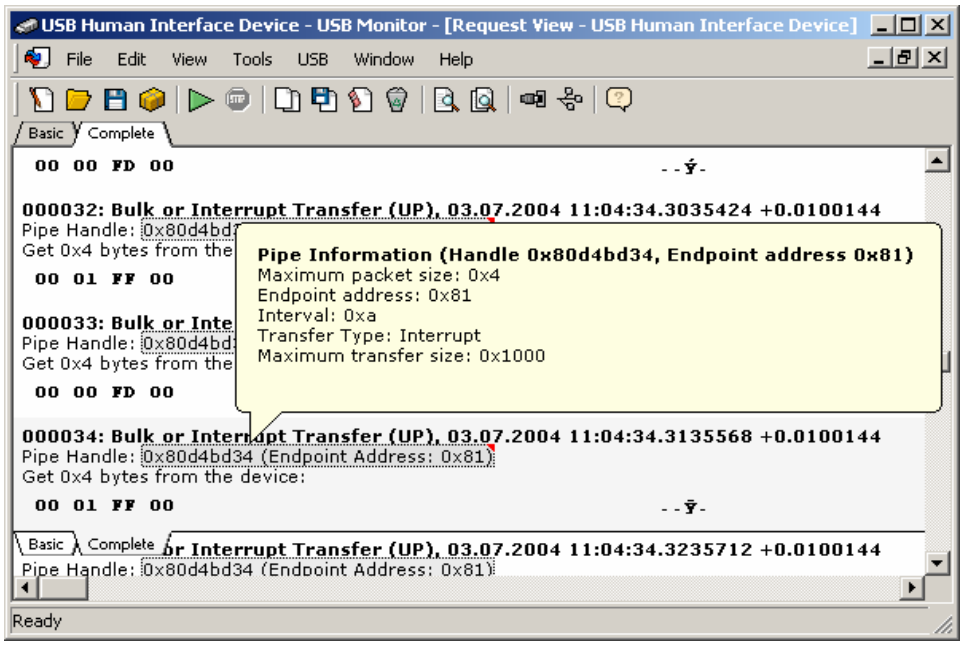
نرم‌افزار USB Monitor

در حال حاضر سخت‌افزارها و نرم‌افزارهای مختلف از پورت‌های USB استفاده‌های بسیار زیادی می‌کنند زیر USB از سرعت و قابلیت اطمینان بیشتری برخوردار بوده و از تکنولوژی Plug and Play به خوبی پشتیبانی می‌کند. نمونه‌هایی از این سخت‌افزارها عبارتند از: اسکنرها، دوربین‌های دیجیتالی، پرینترها و...

با توجه به استفاده‌های نرم‌افزاری و سخت‌افزاری وسیع از این‌گونه پورت‌ها، لزوم داشتن آگاهی کافی در مورد نحوهٔ عملکرد و داده‌های مبادله شده توسط آنها می‌تواند نقش اساسی در زمینهٔ شناسایی پرتکل‌های ارتباطی مورد استفادهٔ این سخت‌افزارها و نرم‌افزارهای جدید ایفا کند.

نرم‌افزار USB Monitor به عنوان ناظری برای پورت‌های USB به کار برده می‌شود که در حقیقت عملکردی مشابه نرم‌افزار Serial Monitor برای پورت‌های USB دارد.

در شکل (۲-۳۳) صفحهٔ اصلی این نرم‌افزار را در حال نمایش داده‌های مبادله شده توسط یک ماوس USB مشاهده می‌کنید.



شکل (۲-۳۳)

نسخه 2.26 این نرم‌افزار در CD ضمیمه موجود می‌باشد

Tools\USBMonitor

بررسی فراخوانی‌های توابع API

همان‌طور که می‌دانید نرم‌افزارهای طراحی شده برای سیستم عامل ویندوز جهت انجام اعمال مورد نیاز خود به شدت به توابع API موجود در این سیستم عامل وابسته هستند.

API در واقع مجموعه‌ای با بیش از ۶۰۰۰ تابع است که در فایل‌های کتابخانه‌ای ویندوز برحسب موضوع و نحوهٔ عملکرد دسته‌بندی شده‌اند. در زیر نام برخی از این فایل‌های کتابخانه‌ای را به همراه توضیحات کوتاهی در مورد هر یک مشاهده می‌کنید.

فایل‌های کتابخانه‌ای استاندارد ویندوز

User32.dll

این فایل کتابخانه‌ای حاوی توابعی برای ایجاد و مدیریت رابط کاربر است. توابع موجود در این کتابخانه کاربردهای فراوانی دارند. از آن جمله می‌توان به موارد زیر اشاره کرد:

- ایجاد پنجره‌ها و مدیریت آنها.
- ایجاد و مدیریت اشیاء گرافیکی مانند منوها ، لیست‌ها ، دکمه‌ها و...
- ایجاد و مدیریت دیالوگ‌ها.
- مدیریت ابزارهای ورودی کاربر از جمله ماوس و کی‌بورد.
- انتقال پیام‌ها بین پنجره‌های مختلف و مدیریت پروسیجرهای پنجره‌ها.
- ایجاد و مدیریت تایمرها در ویندوز.
- بارگذاری و مدیریت منابع در فایل‌های اجرایی از قبیل بیت‌مپ‌ها، کرسرها، آیکون‌ها، دیالوگ‌ها، منوها، رشته‌ها و...
- مدیریت کلیپ بورد.

Kernel32.dll

این کتابخانه و توابع درون آن در حقیقت وظیفهٔ مدیریت و کنترل اشیاء و منابع اصلی سیستم عامل از قبیل فایل‌ها، حافظه، Processها، Threadها را برعهده دارند. در زیر به برخی از وظایف کلیدی توابع موجود در این کتابخانه اشاره شده است.

- مدیریت و کنترل دایرکتوری‌ها، فایل‌ها و درایورهای سخت‌افزاری موجود در سیستم.
- ایجاد و مدیریت Process ها و Thread ها.
- همگام‌سازی و ارتباط بین Process ها و Thread ها.
- مدیریت و اختصاص منابع سیستم از قبیل حافظه و سخت‌افزارهای ورودی و خروجی.
- Debug کردن یک Process.
- مدیریت تاریخ و زمان سیستم.

GDI32.dll

توابع ارائه شده توسط این کتابخانه جهت مدیریت اشیاء گرافیکی به کار گرفته می‌شوند در زیر تعدادی از اعمالی که توسط این توابع قابل انجام است را مشاهده می‌کنید:

- اضافه کردن، مدیریت و چاپ فونت‌ها
- رسم اشکال گرافیکی مانند دایره، کمان، مستطیل، چند ضلعی و...
- چاپ تصاویر گرافیکی برروی صفحهٔ نمایش و یا دستگاه‌های خروجی مانند پرینترها، پلاترها و ...
- ایجاد مدیریت حافظه‌های گرافیکی (DC)
- بارگذاری و استفاده از فایل‌های گرافیکی استاندارد از قبیل Bitmap ، Metafile ، Cursor و Icon
- مدیریت و حمایت از ابزارهای گرافیکی مانند قلم‌ها و رنگ‌ها
- هماهنگ‌سازی ابزارها و حافظه‌های گرافیکی
- رسم رشته‌های متنی

Winmm.dll

توابع موجود در این فایل جهت استفاده از ابزارهای چند رسانه‌ای از قبیل صداها ، تصاویر متحرک و دسته‌های بازی به کار گرفته می‌شوند در زیر چند نمونه از عملیاتی که توسط این توابع قابل انجام است، ذکر شده است.

- مدیریت و استفاده از درایورهای چند رسانه‌ای مانند درایورهای صدا، CD ROMها و تصاویر متحرک
- ایجاد، ضبط و پخش اصوات
- مدیریت و استفاده از دسته‌های بازی (Joystick)
- پخش مدیریت تصاویر متحرک از قبیل Avi ها ، MPEG ها و ....

با توجه به مطالب ذکر شده می‌توانید تصور کنید که داشتن اطلاعات دقیق در مورد توابع API فراخوانی شده توسط یک برنامه، زمان هر فراخوانی، دستیابی به پارامترهای فرستاده شده به هر یک و مقدار برگشتی این توابع چقدر می‌تواند در دستیابی به طرحی جامع، کامل و بدون نقص از نحوهٔ عملکرد و اجزاء آن برنامه مفید واقع شود.

برای شروع کار بهتر است به نحوهٔ فراخوانی توابع API در ویندوزهای خانوادهٔ 9X و NT نگاهی بیاندازیم. توجه داشته باشید که مطالب ذکر شده طرح بسیار ساده‌ای از مراحل بارگذاری کتابخانه‌ها و فراخوانی توابع API بوده و بیشتر به‌منظور ایجاد تصور ذهنی بهتر شما ذکر شده است. درصورت نیاز به اطلاعات کامل‌تر و دقیق‌تر راجع به توابع API، نحوهٔ استفاده از آنها و مراحل بارگذاری فایل‌های dll در حافظهٔ برنامه‌ها می‌توانید به فصل ۸ مراجعه کنید.

عملیات بارگذاری و فراخوانی توابع API

همان‌طور که می‌دانید فایل‌های اجرایی در ویندوز در قسمتی به نام Import Table، نام توابع API و فایل‌های کتابخانه‌ای مورد نیاز خود را مشخص می‌کنند. درهنگام بارگذاری یک فایل اجرایی، سیستم عامل ویندوز فایل‌های کتابخانه‌ای مورد نیاز آنرا به حافظهٔ اختصاصی در نظر گرفته شده برای اجرای آن فایل نگاشت می‌کند. در مرحلهٔ بعد عملیات تصحیح آدرس‌های توابع ارائه شده توسط فایل‌های کتابخانه‌ای انجام می‌شود. این عمل سبب می‌شود که کدهای برنامه بتوانند با عملیات Jump و Call به‌صورت ساده‌ای با کدهای توابع API ارتباط برقرار کرده و آنها را مانند توابع داخلی خود فراخوانی کنند.

فایل‌های اجرایی در مرحلهٔ اجرا نیز می‌توانند فایل‌های کتابخانه‌ای مورد نظر خود را بارگذاری کرده و با استخراج آدرس توابع مورد از آنها استفاده کنند.

حال بهتر است به برخی از ابزارهای نظارت بر فراخوانی‌های API که در اصطلاح به آنها API Sniffer نیز گفته می‌شود، نگاهی بیاندازیم.

نظارت بر فراخوانی‌های توابع API

نرم‌افزار API Monitor

این نرم‌افزار یکی از معروف‌ترین ابزارهای API Sniffing به شمار می‌آید. این ابزار کاربرد بسیار ساده‌ای داشته و از امکانات و قابلیت‌های بسیار مفیدی برخوردار است و می‌تواند برای رسیدن به مقاصد گوناگون به کار گرفته شوند. یکی از قابلیت‌های مفید این نرم‌افزار توانایی آن در اعمال فیلترهایی براساس موضوع API و نام Process فراخوانی‌کنندهٔ آنها است. این فیلترها می‌توانند برروی حجم انبوه فراخوانی‌های API در کل سیستم اعمال شده و از نمایش فراخوانی‌هایی که موردنظر کاربر نیستند، جلوگیری کند.

نسخه 1.5 این نرم‌افزار در CD ضمیمه موجود می‌باشد

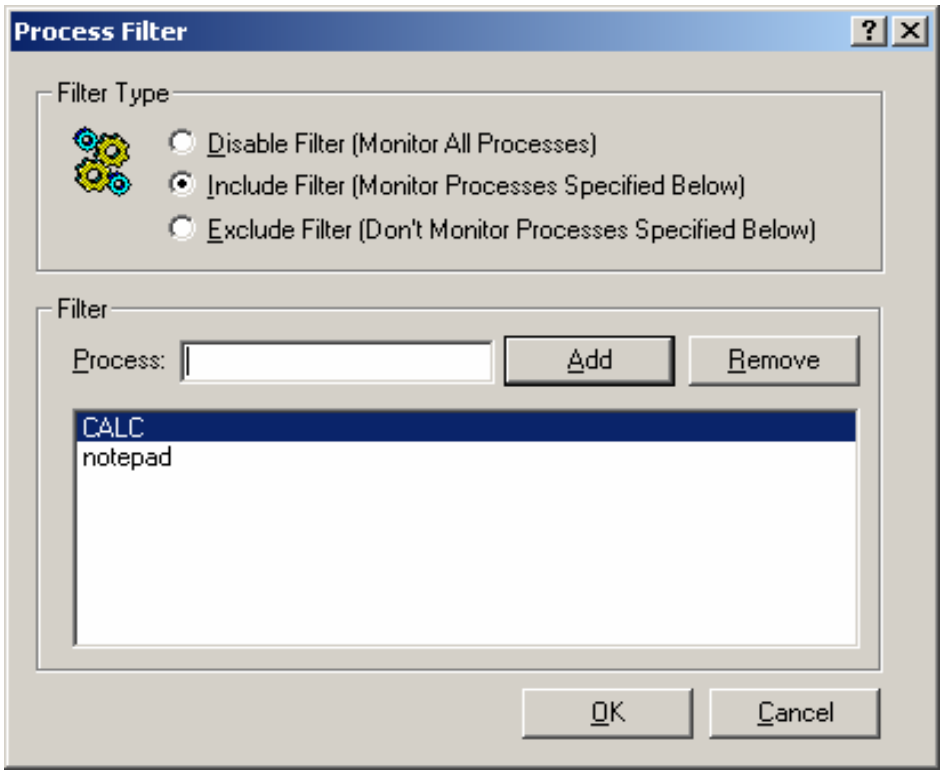
Tools\APIMonitor

حال نحوهٔ کار با این نرم‌افزار را مورد بررسی قرار می‌دهیم. معمولاً اولین قدم در استفاده از نرم‌افزارهای Sniffing، تعیین فیلتر است که نتایج حاصل از مراحل بعدی را به شدت تحت تأثیر قرار می‌دهد.

تعیین فیلتر برای Process ها

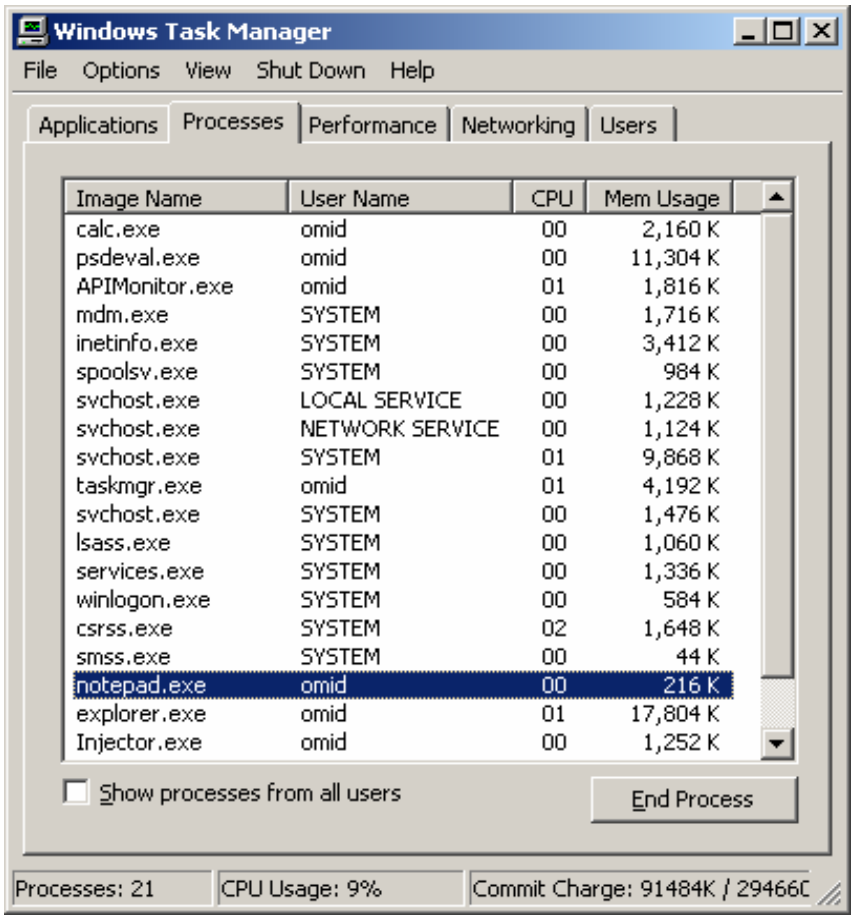
با توجه به اینکه فراخوانی‌های توابع API ممکن است در Processهای مختلف در حال اجرا در سیستم صورت بگیرد، می‌توان با تعیین فیلتری برای Processها، نمایش فراخوانی‌های انجام شده توسط آنها را محدود کرد.

با انتخاب گزینهٔ Process Filter از منوی Capture، صفحهٔ Process Filter مانند شکل (۲-۳۵) ظاهر شده و امکان تعیین فیلترهایی را برای Process های موجود در سیستم ایجاد می کند.



شکل (۲-۳۵)

همان‌طور که در شکل (۲-۳٦) می‌بینید، نام Processهای موجود در سیستم را می‌توانید در قسمت Processes از Task Manager ویندوز مشاهده کنید.



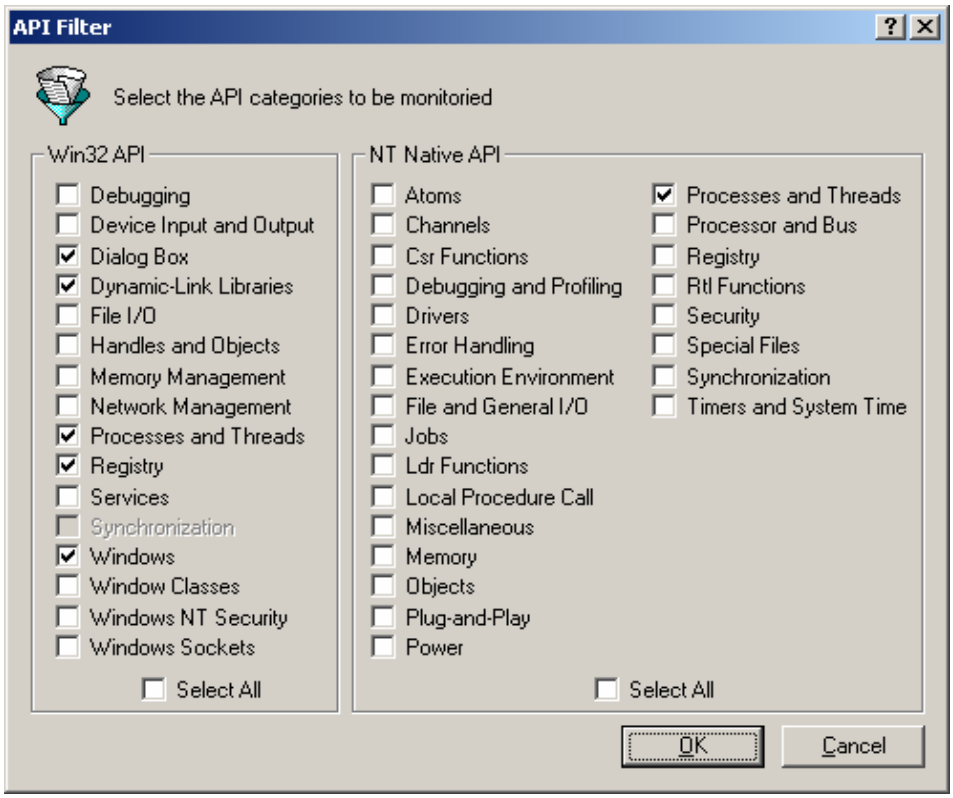
شکل (۲-۳۶)

تعیین فیلترهایی برای توابع API

همان‌طور که ذکر شد، توابع API براساس موضوع و نحوهٔ عملکرد خود به دسته‌هایی تقسیم می‌شوند. نرم‌افزار API Monitor این امکان را به شما می‌دهد که نمایش فراخوانی‌های API را به دسته‌های مشخصی از آنها محدود کنید و در حقیقت با این عمل موضوع مورد نظر خود را مشخص کنید.

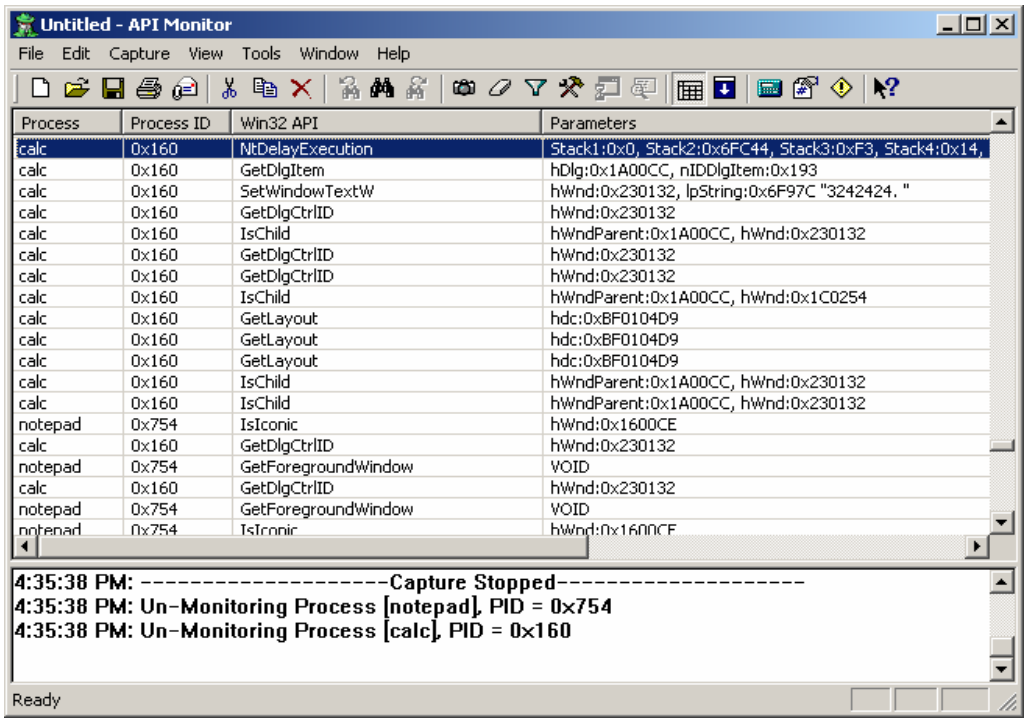
همان‌طور که در شکل (۲-۳۷) مشاهده می‌کنید با انتخاب گزینهٔ API Filter از منوی Capture، پنجرهٔ API Filter نمایش داده شده و به شما امکان انتخاب موضوعات موردنظر خود را می‌دهد.

بدیهی است که با تعیین این فیلترها، دیگر فراخوانی‌های API غیر دلخواه شما در پنجرهٔ خروجی نمایش داده نخواهد شد.



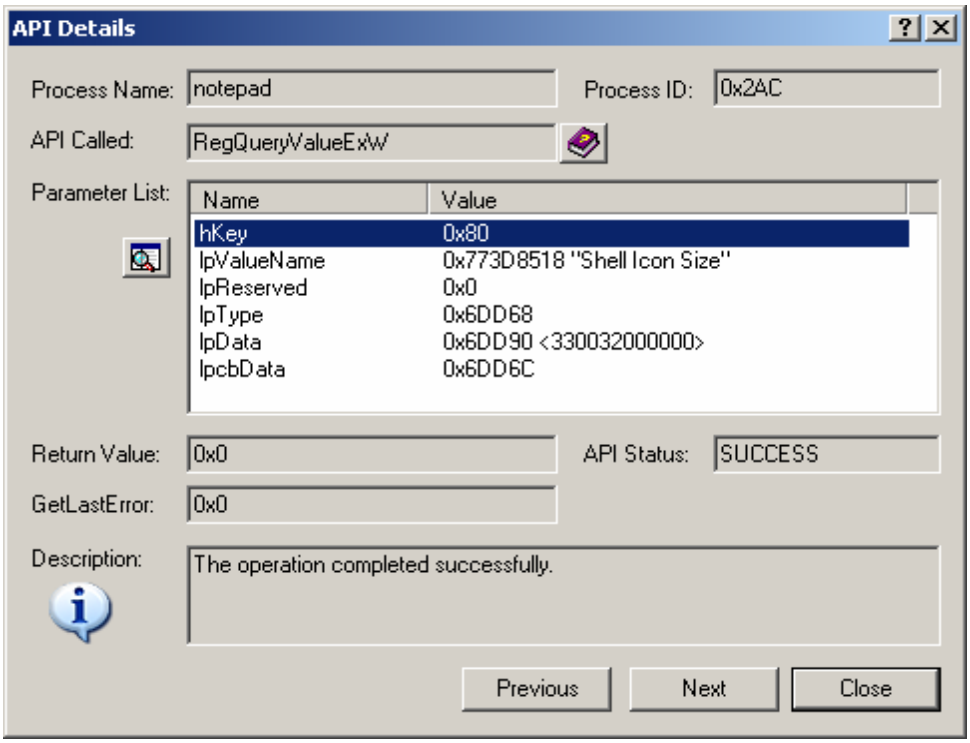
شکل (۲-۳۷)

برای شروع ضبط و نمایش فراخوانی‌های API، گزینهٔ Capture API Events از منوی Capture را انتخاب کنید. همان‌طور که در شکل (۲-۳۸) مشاهده می‌کنید، لیستی از فراخوانی‌های انجام شده توسط دو Process با نام‌های calc و notepad باتوجه به فیلترهای تعیین شده برای توابع API در صفحهٔ اصلی برنامه نمایش داده می‌شود.



شکل (۲-۳۸)

در صورت نیاز می‌توانید از امکانات جستجو به‌منظور پیدا کردن رشته‌های دلخواه در لیست استفاده کنید. به‌منظور کسب اطلاعات دقیق‌تر راجع به یک فراخوانی می‌توانید برروی آن Double Click کنید. با این کار پنجرهٔ API Details همانند شکل (۲-۳۹) باز شده و اطلاعات کاملی را راجع به پارامترهای ارسالی به تابع و نیز مقدار برگشتی آن نمایش می‌دهد.



شکل (۲-۳۸)

نرم‌افزار Smart Check

یکی از قوی‌ترین و معروف‌ترین ابزارهای موجود برای رفع مشکلات اجرایی نرم‌افزارها است. این ابزار می‌تواند به عنوان یک API Sniffer نیز به کار برده شود

این نرم‌افزار می‌تواند فراخوانی‌های API انجام شده توسط یک فایل اجرایی را ضبط کرده و آنها را به‌صورت سلسله مراتبی برحسب dllها و ActiveXهای مورد استفاده و یا خود فایل اجرایی نمایش دهد. این امر باعث خوانایی بیشتر و ایجاد قابلیت درک بهتر اطلاعات نمایش داده شده در مورد فراخوانی‌ها می‌گردد.

اطلاعات نمایش داده شده در مورد پارامترهای ارسال شده به توابع از جمله ثابت‌ها، رکوردها و ... نیز از خوانایی بسیار مناسبی برخوردار هستند. همچنین این نرم‌افزار اطلاعات مفیدی را راجع به شروع و خاتمهٔ Threadهای مختلف برنامه و نیز خطاهای اتفاق افتاده نمایش می‌دهد.

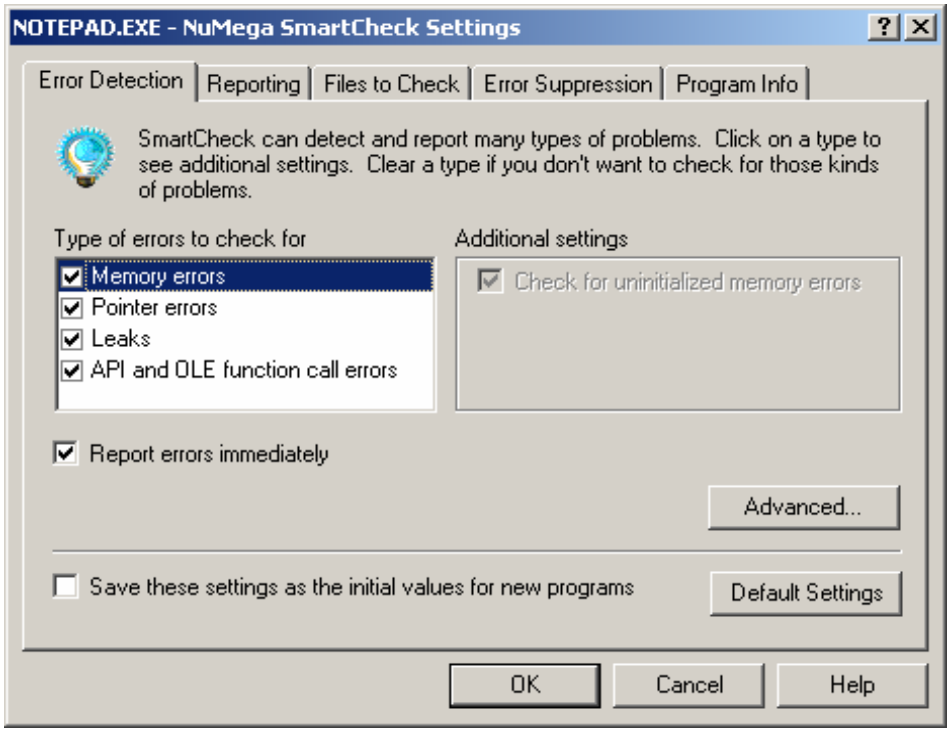
نسخه 6.3 این نرم‌افزار در CD ضمیمه موجود می‌باشد

Tools\SmartCheck

حال که با امکانات این نرم‌افزار آشنایی نسبی پیدا کردید، بهتر است نحوهٔ استفاده از آن را مورد بررسی قرار دهیم.

برای استفاده از این نرم‌افزار به عنوان یک API Sniffer، باید در قسمت تنظیمات آن تغییراتی را ایجاد کنید که در ادامه به آنها اشاره خواهد شد. توجه داشته باشید که تنظیمات ذخیره شده برای هر فایل اجرایی می‌تواند با فایل‌های دیگر متفاوت باشد به این معنی که این نرم‌افزار برای هر فایل اجرایی باز شده، و اجرا شده، تنظیمات جداگانه‌ای را ذخیره می‌کند.

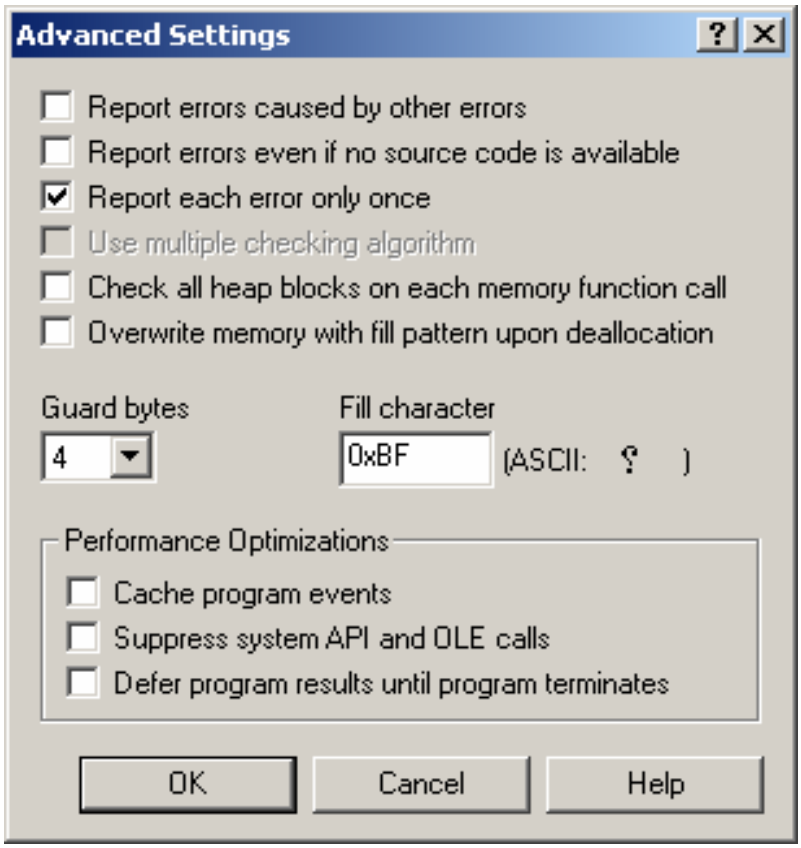
پس از باز کردن یک فایل اجرایی، با انتخاب گزینهٔ Settings از منوی Program، پنجرهٔ Settings همانند شکل (۲-۳۹) باز نشده و به شما امکان ایجاد تغییراتی را در تنظیمات موردنظر برای فایل اجرایی باز شده می‌دهد.



شکل (۲-۳۹)

توجه داشته باشید که درصورت نیاز به استفاده از این نرم‌افزار به‌عنوان یک API Sniffer می‌توانید از تنظیمات نشان داده شده در تصاویر استفاده کنید.

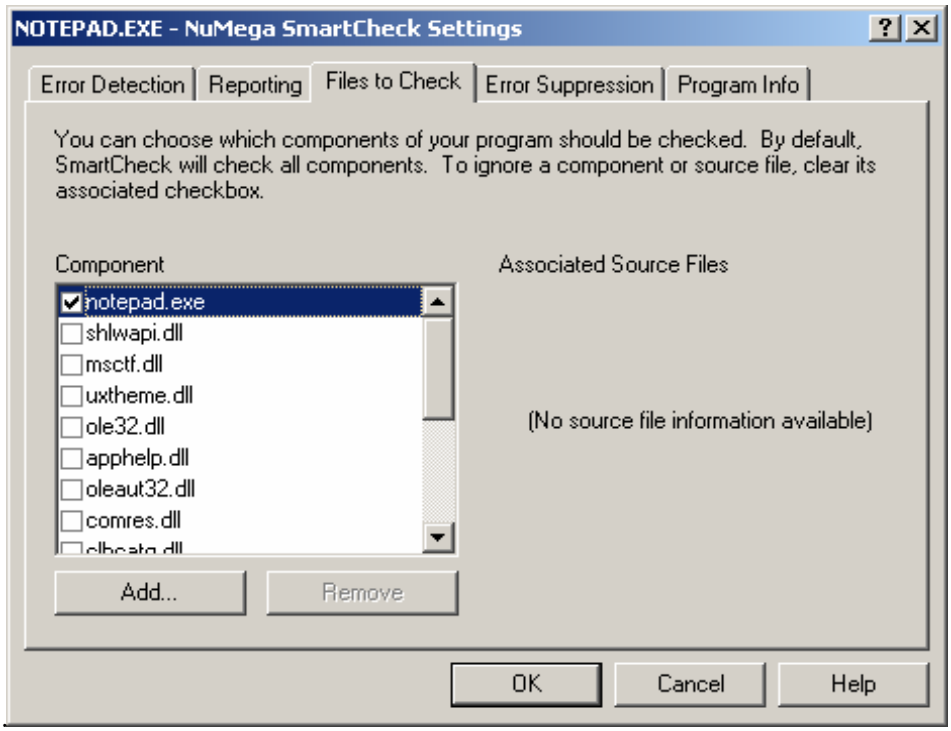
با کلیک برروی دکمه Advanced از قسمت Error Detection پنجره Advanced Settings مانند شکل (۲-۴۰) ظاهر شده و به شما امکان تغییر در تنظیمات پیشرفته را می‌دهد.



شکل (۲-۴۰)

خارج کردن گزینه Suppress System API and OLE Calls از حالت انتخاب، سبب می‌شود که این نرم افزار فراخوانی‌های انجام شده توسط فایل اجرای و یا سایر فایل‌های جانبی وابسته به آن از قبیل dllها، OLEها و یا ActiveXها را تشخیص داده و نمایش دهد.

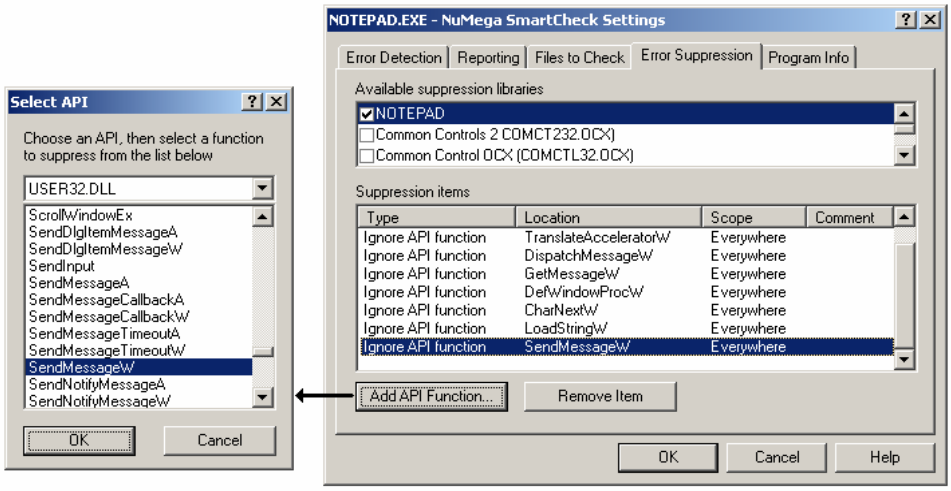
همان‌طور که در شکل (۲-۴۱) مشاهده می‌کنید، در قسمت Files to Check از پنجره Settings می‌توانید عملیات گزارش‌گیری از فراخوانی‌ها را به فایل‌های خاصی محدود کنید.



شکل (۲-٤۱)

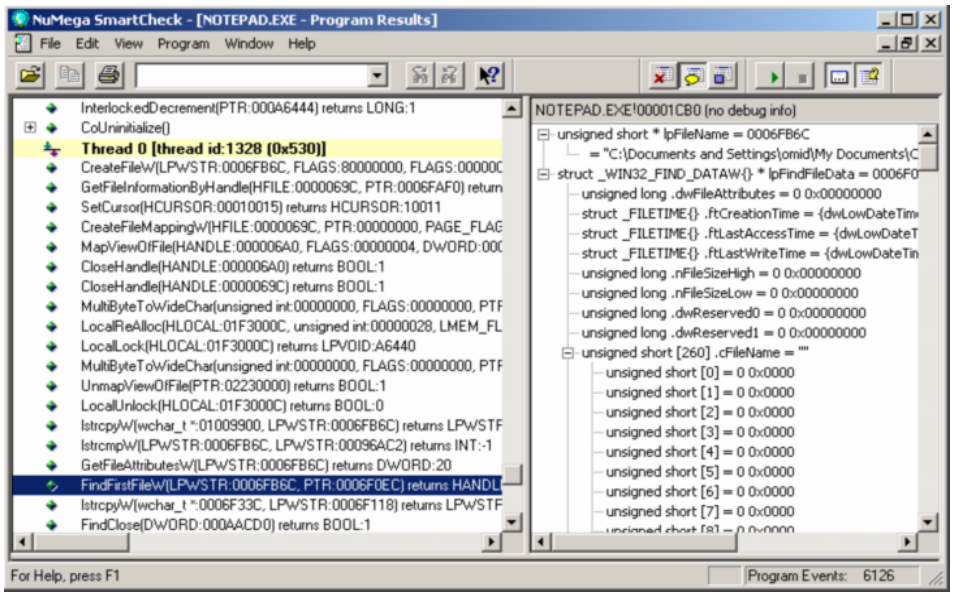
توجه داشته باشید که فایل‌های جانبی نمایش داده شده، توسط فایل اجرایی مورد استفاده قرار می‌گیرند و در نتیجه در صورت نیاز به دریافت اطلاعات در مورد فراوانی‌ها و نحوه عملکرد این فایل‌ها می‌توانید آنها را در لیست نمایش داده شده انتخاب نمایید. فایل‌های جانبی مورد استفاده فایل اجرایی با اجرا شدن توسط این نرم افزار تشخیص داده شده و در لیست مذکور قرار می‌گیرند.

در قسمت Error Suppression از پنجره Settings می‌توانید توابع خاصی را از لیست فراخوانی‌ها حذف کنید و در حقیقت با این عمل فیلتری را برای توابع API تعیین کنید. اعمال این فیلترها باعث خوانایی بیشتر و کاهش تعداد فراخوانی‌های نمایش داده شده می‌شود. به عنوان مثال در شکل (۲-٤۲) فیلترهای تعیین شده برای فایل اجرایی notepad.exe را مشاهده می‌کنید.



شکل (۲-۴۲)

به‌منظور اجرای فایل اجرایی و بررسی فراخوانی‌های انجام شده می‌توانید گزینه Start از منوی Program را انتخاب کرده و یا کلید F5 را فشاردهید. با انجام این عمل نرم‌افزار موردنظر اجرا شده و صفحه اصلی نرم‌افزار Smart Check فراخوانی‌های صورت گرفته را به همراه اطلاعات دقیق و کاملی در مورد نوع و مقادیر پارامترهای ارسال شده به هر یک نمایش می‌دهد. در شکل (۲-۴۳) صفحه اصلی این نرم‌افزار را درحال نمایش فراخوانی‌های ضبط شده از نرم‌افزار notepad مشاهده می‌کنید.



شکل (۲-۴۳)

در صورت نیاز برای شروع و یا توقف عملیات ضبط و نمایش فراخوانی‌ها در هنگام اجرای فایل اجرایی از گزینه Event Reporting که در منوی program قرار دارد استفاده کنید. توجه داشته باشید که برای مشاهده فراخوانی‌های انجام شده باید گزینه Show all events از منوی view در حالت فعال قرار داشته باشد.

نرم افزار SoftSnoop

این نرم‌افزار در حقیقت یک Debugger است که قابلیت‌هایی نیز در زمینه بررسی فراخوانی‌های توابع API دارد. یکی از امکانات بسیار مفید این نرم‌افزار که آن را از دیگر نرم‌افزارهای API Sniffing متمایز می‌سازد، توانایی آن در متوقف ساختن روند اجرایی فایل اجرایی در صورت فراخوانی توابع API موردنظر کاربر و اعمال تغییرات در پارامترهای ارسالی تابع و مقدار برگشتی آن می‌باشد. با این کار در روند اجرایی فایل اجرایی موردنظر تغییراتی اعمال می‌شود. این تغییرات می‌توانند به‌منظور بررسی نحوه عملکرد و واکنش‌های یک نرم‌افزار به کار گرفته شوند.

نسخه 1.3 این نرم‌افزار در CD ضمیمه موجود می‌باشد

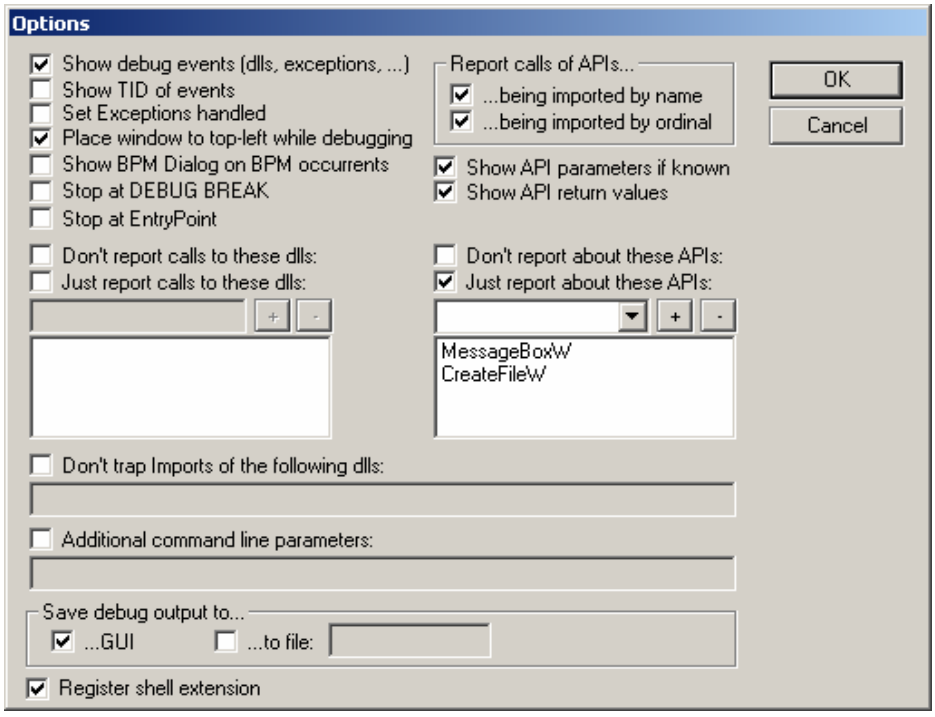
Tools\SoftSnoop

حال نحوه عملکرد این نرم‌افزار را در قالب مثال‌هایی مورد بررسی قرار می‌دهیم. در قسمت اول مراحل لازم برای نظارت بر فراخوانی‌های انجام گرفته، تشریح شده است و سپس نحوه ایجاد تغییرات در این فراخوانی‌ها مورد بررسی قرار می‌گیرد.

نظارت بر فراخوانی‌های API

همانند نرم‌افزارهای قبلی که آنها را مورد بررسی قرار دادیم، این نرم‌افزار نیز از روش‌های خاصی جهت فیلتر کردن توابع API نمایش داده شده و نیز فایل‌های کتابخانه‌ای استفاده می‌کند. این امر باعث متمرکز شدن هر چه بیشتر کاربر برروی فراخوانی‌های موردنظر خود می‌گردد.

قبل از باز کردن یک فایل اجرایی بهتر است به سراغ قسمت تنظیمات این نرم‌افزار رفته و آنها را به منظور نمایش فراخوانی‌ها و اطلاعات با فرمت و فیلترهای مورد نظر خود تغییر دهیم. برای این کار ابتدا گزینه Set Options را از منوی Options انتخاب کرده و یا کلید F1 را فشاردهید. همان‌طور که در شکل (۲-٤٤) مشاهده می‌کنید پنجره Options نمایش داده شده و به شما امکان ایجاد تغییرات در تنظیمات برنامه را قبل از شروع فایل اجرایی می‌دهد.

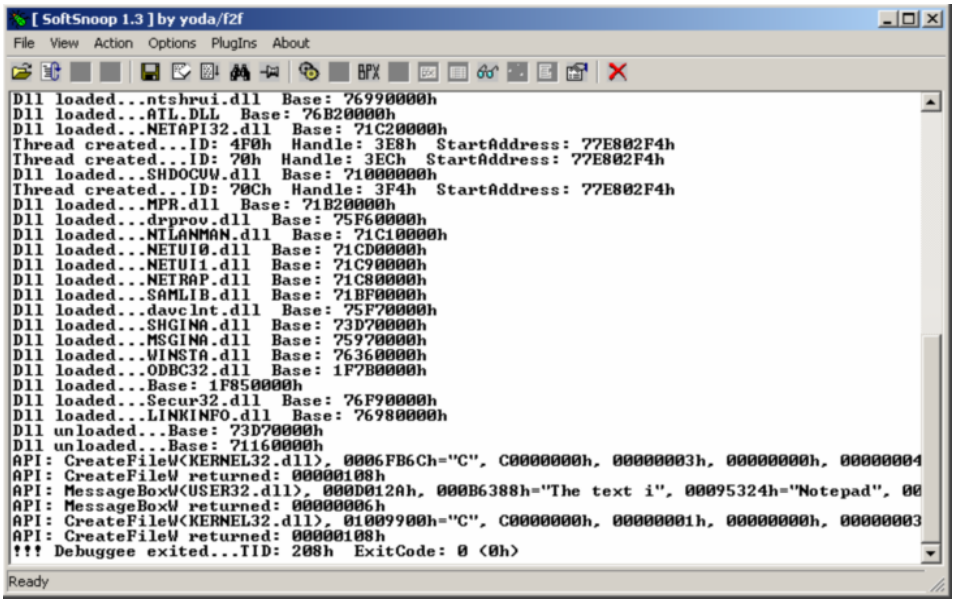


شکل (۲-٤٤)

همان‌طور که مشاهده می‌کنید، فیلترهای موردنظر را می‌توانید با توجه به نام توابع و یا فایل‌های کتابخانه‌ای تعیین کنید.

با استفاده از فیلتر تعیین شده در این مثال تنها فراخوانی‌های دو تابع API با نام‌های MessageBoxW و CreateFileW در صفحه نمایش داده خواهند شد. به‌منظور نمایش فراخوانی‌های انجام شده توصیه می‌شود که از تنظیمات مثال مذکور استفاده کنید.

پس از اعمال تغییرات مورد نیاز در قسمت Options می‌توانید فایل اجرایی مورد نظر را باز کرده و با اجرای آن، اطلاعات نمایش داده شده را مشاهده کنید. در شکل (۲-٤٥) اطلاعات نمایش داده شده پس از اجرای فایل notepad.exe را با توجه به تنظیمات مذکور مشاهده می‌کنید.

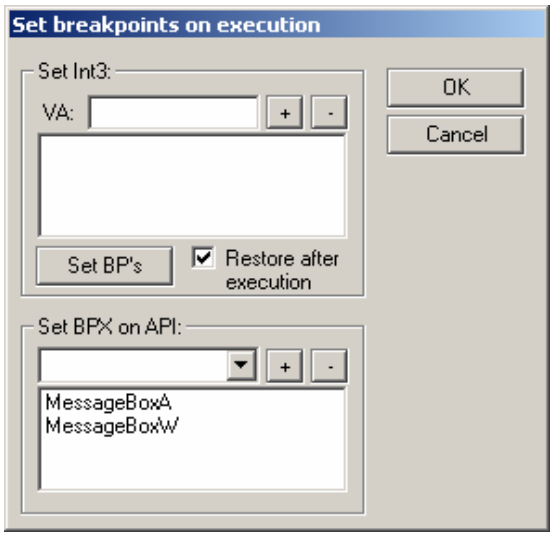


شکل (۲-٤٥)

ایجاد تغییرات در فراخوانی‌های API

همان‌طور که گفته شد، این نرم‌افزار توانایی اعمال تغییرات در پارامترهای ارسالی و مقدار برگشتی توابع API فراخوانی شده را دارد. از این‌گونه تغییرات به‌منظور بررسی نحوه عملکرد و واکنش‌های برنامه، استفاده‌های فراوانی می‌شود. به عنوان مثال برای بررسی واکنش‌های یک برنامه در برابر خطاهای احتمالی می‌توان تغییراتی را در مقدار برگشتی از توابع API موردنظر به وجود آورد.

به‌منظور ایجاد تغییرات در پارامترهای ارسال شده به توابع API ابتدا باید نقاط توقفی را برای توابع موردنظر خود ایجاد کنید. به این منظور گزینه Set BPX را از منوی Action را انتخاب کنید. پنجره Set Breakpoint همانند شکل (۲-٤٦) نمایش داده شده و به شما امکان ایجاد توقف در آدرس‌های مجازی خاص ویا توابع API مورد نظر را می‌دهد.

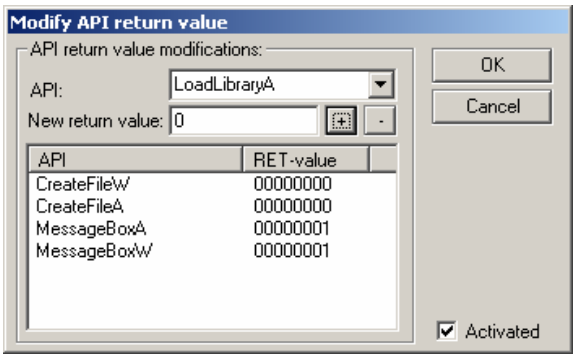


شکل (۲-٤٦)

همان‌طور که در شکل مشاهده می‌کنید، در این مثال نقاط توقفی برای دو تابع MessageBoxA و MessageBoxW در نظر گرفته شده است. در هنگام اجرای فایل اجرایی در صورت فراخوانی توابع تعیین شده روند اجرایی آن موقتاً متوقف شده و به شما امکان ایجاد تغییرات در پارامترهای ارسالی به توابع داده می‌شود.

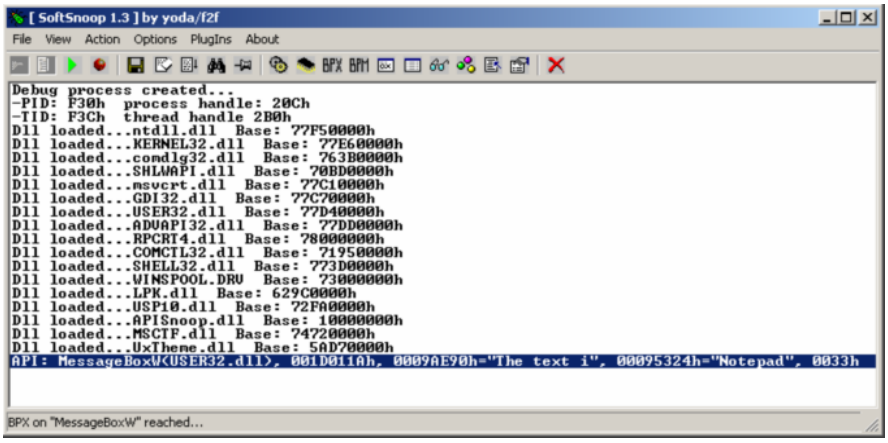
همان‌طور که قبلاً ذکر شد شما می‌توانید در مقادیر برگشتی ازتوابع API موردنظر خود نیز تغییراتی را اعمال کنید. به این منظور گزینه Modify API Return Value از منوی Action را انتخاب کنید.

همان‌طور که در شکل (۲-٤۷) مشاهده می‌کنید با این عمل پنجره Modify API Return Value نمایش داده شده و به شما امکان تعیین مقادیر برگشتی موردنظر خود را برای توابع دلخواه می‌دهد.



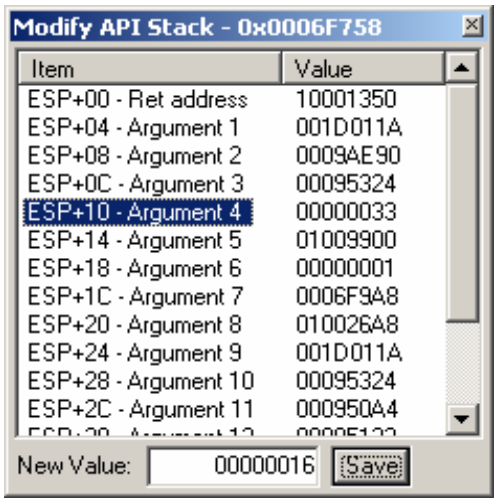
شکل (۲-۴۷)

به‌منظور روشن شدن مطالب ذکر شده، با توجه به نقاط توقف تعریف شده در مراحل قبل، فایل اجرایی notepad.exe را باز کرده و آن را اجرا می‌کنیم. به‌منظور آزمایش نقاط توقف، ابتدا متنی را درون notepad تایپ کرده و سپس سعی کنید از آن خارج شوید. به علت ذخیره نشدن تغییرات، این نرم‌افزار با استفاده از تابع MessageBoxW شما را مطلع می‌کند. ولی با توجه به تعریف نقطه توقف برای این تابع، روند اجرایی notepad پس از فراخوانی تابع MessageBoxW متوقف شده و صفه اصلی نرم افزار Softsnoop همانند شکل (۲-۴۸) ظاهر می‌شود.



شکل (۲-۴۸)

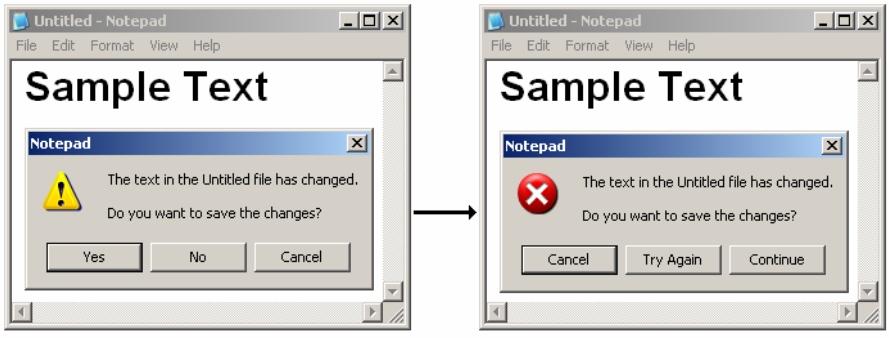
در این مرحله شما می‌توانید تغییرات دلخواه خود را بر روی پارامترهای ارسال شده به تابع MessageBoxW از طرف نرم‌افزار notepad اعمال نمایید. به این منظور می‌توانید گزینه Modify API Stack را از منوی Action انتخاب کرده و یا کلید F4 را فشار دهید. با این عمل پنجره Modify API Stack همانند شکل (۲-۴۹) ظاهر شده و به شما اجازه تغییر پارامترهای ارسالی به تابع MessageBoxW و سایر مقادیر درج شده در Stack را می‌دهد.



شکل (۲-۴۹)

همان‌طور که می‌دانید پارامترهای مورد نیاز توابع توسط Stack به آنها فرستاده می‌شوند. در نتیجه با تغییر مقادیر درج شده در Stack می‌توان در پارامترهای رسالی تغییرات دلخواه را ایجاد کرد.

در این مثال ما مقدار پارامتر چهارم ارسال شده به تابع MessageBoxW که تعیین کننده نوع و شکل پنجره پیغام و دکمه‌های موجود در آن است را تغییر می‌دهیم و به جای مقدار اولیه (۳۳) مقدار ۱۶ را جایگزین کرده و ذخیره می‌کنیم. حال برای ادامه روند اجرایی نرم افزار notepad، گزینه Resume All Threads را از منوی Action انتخاب می‌کنیم. به این ترتیب روند اجرایی این نرم افزار ادامه پیدا کرده و پیغام اخطار نمایش داده می‌شود. همان‌طور که در شکل (۲-۵۰) مشاهده می‌کنید، به دلیل تغییر پارامترهای ارسالی، شکل ظاهری پیغام نمایش داده شده با پیغام اصلی متفاوت است.



شکل (۲-۵۰)

فصل سوم

بررسی کد

فصل سوم

بررسی کد

در مراحل جمع‌آوری اطلاعات، شناسایی اجزاء و تحلیل نحوه عملکرد یک نرم‌افزار، بررسی کدها و دستورالعمل‌های آن نرم‌افزار می‌تواند یکی از مطمئن‌ترین، دقیق‌ترین و قوی‌ترین راه‌ها برای رسیدن به این مقصود باشد. این روش مطمئناً یکی از سخت‌ترین روش‌ها نیز محسوب می‌شود و نیازمند داشتن دانش کافی و صرف وقت زیاد است.

با توجه به این که در اکثر موارد در محصول نهایی ما به کدهای اصلی نرم‌افزار که معمولاً با استفاده از یک زبان سطح بالا مانند C / C++ نوشته شده‌اند دسترسی نداریم، مجبور هستیم فعالیت خود را برروی تحلیل کدهای کامپایل شده که به‌صورت دستورالعمل‌های ماشین مقصد هستند متمرکز کنیم. معمولاً در اولین قدم به وسیله نرم‌افزارهای Disassembler کدهای کامپایل شده به زبان اسمبلی ماشین مقصد که در حقیقت همان دستورالعمل‌های ماشین هستند ترجمه می‌شوند. در مراحل بعدی به‌منظور ایجاد خوانایی بیشتر کدهای اسمبلی تولید شده، تحلیل‌های متعددی به‌منظور شناسایی ساختارهای متداول در زبان‌های سطح بالا مانند حلقه‌ها، توابع، ارجاع‌ها و... برروی آن صورت گرفته و راهنماهای متعددی بهمنظور ایجاد خوانایی بهتر به آن اضافه می‌گردد.

در برخی از موارد در صورت مشخص بودن نوع زبان و کامپایلر استفاده شده برای ایجاد فایل اجرایی، نرم‌افزارهای Decompiler سعی می‌کنند با بررسی دستورالعمل‌های موجود در فایل اجرایی نهایی و تحلیل ساختارهای استاندارد مورد استفاده در آن کامپایلر و زبان، هر چه بیشتر کدهای تولید شده را به زبان موردنظر نزدیک کنند. در عمل کدهای نهایی تولید شده توسط Decompiler ها معمولاً به‌صورت شبه کدی از کدهای اصلی نرم‌افزار بوده و بیشتر به‌منظور شناسایی و تحلیل بهتر ساختارها و عملکرد نرم‌افزار به کار گرفته می‌شوند.

Disassembler ها

به‌عمل ترجمه یک فایل اجرایی و یا هرگونه کد کامپایل شده، به‌برنامه اسمبلی معادل، Disassemble کردن گفته می‌شود. با وجود مشکلات بسیار مطمئناً تجزیه و تحلیل برنامه اسمبلی تولید شده بسیار آسان‌تر از کدهای ماشین موجود در فایل اجرایی است. به‌منظور ایجاد تصویر ذهنی بهتر، در شکل (۳-۱) تکه‌ای از یک برنامه اسمبلی نوشته شده و کدهای ماشین معادل آن برای CPUهای X86 را در فایل اجرایی نهایی مشاهده می‌کنید.

Assembly Code	Machine Code (HEX)	Compiled Machine Code (Binary)
push 00000100h	68 00 01 00 00	01101000 00000000 00000001 00000000 00000000
push [ebp+08h]	FF 75 08	11111111 01110101 00001000
call [4020B4]	FF 15 B4 20 40 00	11111111 00010101 10110100 00100000 01000000 00000000
mov ebx,eax	8B D8	10001011 11011000
mov eax,[403106]	A1 06 31 40 00	10100001 00000110 00110001 01000000 00000000
mov ecx,0000000Ah	B9 0A 00 00 00	10111001 00001010 00000000 00000000 00000000
xor edx,edx	33 D2	00110011 11010010
div ecx	F7 F1	11110111 11110001
add edx,00000030h	83 C2 30	10000011 11000010 00110000
add eax,00000030h	83 C0 30	10000011 11000000 00110000
shl edx,08h	C1 E2 08	11000001 11100010 00001000
add eax,edx	03 C2	00000011 11000010
mov [40316D],ax	66 A3 6D 31 40 00	01100110 10100011 01101101 00110001 01000000 00000000
mov edx,[40310A]	8B 15 0A 31 40 00	10001011 00010101 00001010 00110001 01000000 00000000
mov eax,403159	B8 59 31 40 00	10111000 01011001 00110001 01000000 00000000
cmp edx,00000002h	83 FA 02	10000011 11111010 00000010

شکل (۳-۱)

همان طور که مشاهده می‌کنید، هر دستور در اسمبلی، مستقیماً به کد ماشین ترجمه می‌شود که در اصطلاح به آن Opcode گفته می‌شود. با توجه به مراحل بسیار ساده ای که برای کامپایل کردن یک برنامه اسمبلی مورد استفاده قرار می‌گیرد، مطمئناً عملیات لازم برای Disassemble کردن کدهای

ماشین تولید شده نیز از پیچیدگی خاصی برخوردار نیست. در حقیقت مهم‌ترین و مشکل‌ترین بخش کار تشخیص و تمایزدادن دستورالعمل‌ها از داده‌ها در فایل اجرایی است که نیاز به تفسیر و تحلیل زیادی دارد.

توجه داشته باشید که کدهای نهایی تولید شده توسط کامپایلر بستگی به ماشین اجراکننده آنها دارد. به عنوان مثال ماشین‌هایی مانند 80x86 , Z80 و یا حتی ماشین مجازی JAVA ساختارهای متفاوتی را برای کدگذاری دستورالعمل‌های خود استفاده کرده و هر کدام دستورالعمل‌های خاص خود را دارند.

به‌منظور خوانایی بیشتر کدهای اسمبلی تولید شده، اکثر Disassemblerها بررسی‌ها و تحلیل‌های متعددی را برروی آن انجام داده و راهنماهای خاصی را به کد اسمبلی تولید شده اضافه می‌کنند. به عنوان مثال این راهنماها می‌توانند شروع و پایان زیر برنامه‌ها و یا رشته‌های کاراکتری موجود در کد اسمبلی را مشخص کنند.

نکته دیگری که باید به آن توجه داشته باشید این است که در عمل برنامه اسمبلی تولید شده توسط Disassemblerها نمی‌تواند بدون تصحیح از طرف کاربر، به عنوان ورودی برای کامپایل مجدد به کار گرفته شود.

حال که با کاربردهای Disassembler ها آشنایی نسبی پیدا کردید بهتر است چند نمونه از این نرم‌افزارها را مشاهده کرده و قابلیت‌های آنها را مورد بررسی قرار دهیم.

نرم‌افزار W32Dasm

یکی از معروفترین و قدیمی‌ترین ابزارها برای Disassemble کردن فایل‌های اجرایی ویندوزهای ۱۶ بیتی و ۳۲ بیتی است. این نرم‌افزار قابلیت‌های استانداردی را برای بررسی و جستجوی کدهای اسمبلی تولید شده ارائه می‌کند. کدهای اسمبلی تولید شده به دلیل عدم تجزیه و تحلیل قوی از طرف این نرم‌افزار، از خوانایی بالایی برخوردار نبوده و راهنماهای زیادی نیز در آنها مشاهده نمی‌شود. تنها راهنماهای اضافه شده به کد اسمبلی، راهنماهایی برای فراخوانی‌های توابع API و نیز برخی ارجاع‌ها به توابع و منابع موجود در فایل اجرایی است.

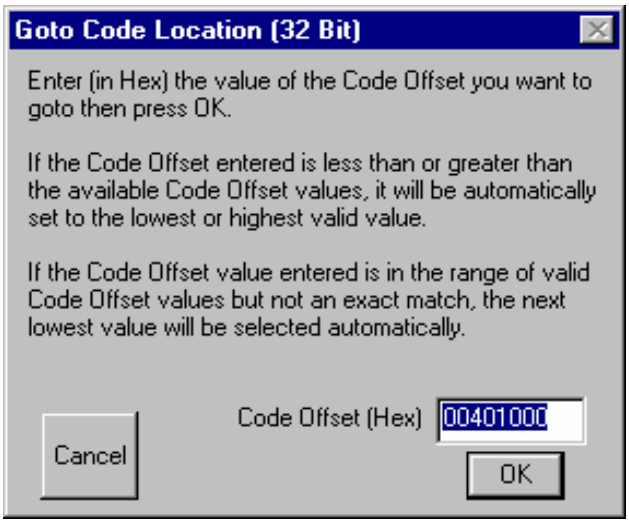
نسخه 8.7 این نرم‌افزار در CD ضمیمه موجود می‌باشد

Tools\W32Dasm

حال نحوه کار با این نرم‌افزار را در قالب مثال‌ها و نکاتی مورد بحث و بررسی قرار می‌دهیم.

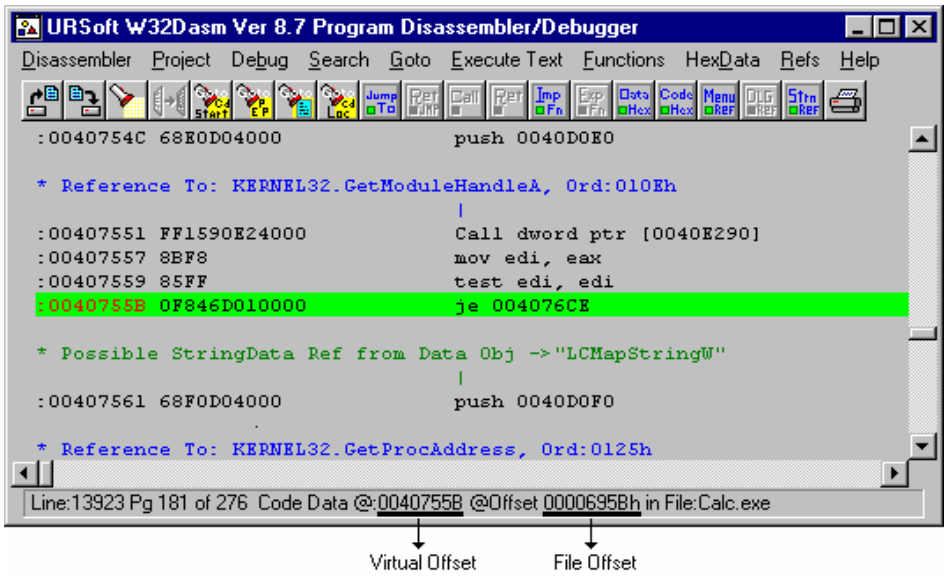
۱- با انتخاب گزینه Go to program entry point از منوی Go to و یا فشردن کلید F10 می‌توانید به مدخل فایل اجرایی منتقل شوید.

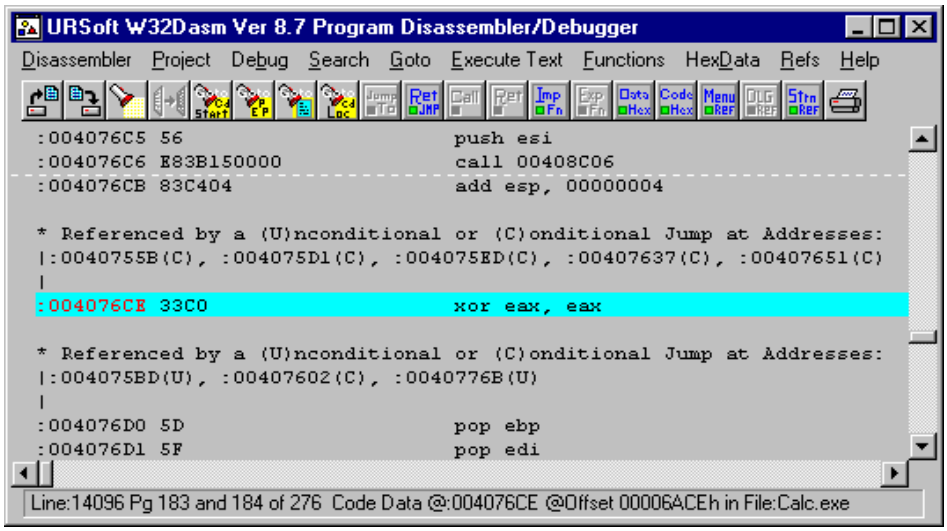
۲- با انتخاب گزینه Go to Code Location از منوی Go to و یا فشردن کلید F12، پنجره Go to code location همانند شکل (۳-۲) نمایش داده شده و امکان رفتن به آدرس دلخواه از فایل اجرایی را برای کاربر ایجاد می‌کند. توجه داشته باشید که آدرس موردنظر در این قسمت، آدرس مجازی در فایل اجرایی است. در صورت نیاز برای کسب اطلاعات بیشتر راجع به آدرس‌های مجازی در فایل‌های اجرایی می‌توانید به فصل ۸ مراجعه کنید.



شکل (۳-۲)

۳- برای دنبال کردن دستورات پرشی مانند JMP , JE ,... در حالتی که آدرس قطعی پرش مشخص است می‌توانید با Double Click کردن برروی دستور پرشی موردنظر آن را انتخاب کرده و با استفاده از کلید جهت راست و یا انتخاب گزینه Excute Jump از منوی Excute Text به آدرس مقصد آن دستور پرشی منتقل شوید. در شکل (۳-۳) این عمل برای یک دستور JE انجام شده است. که نتیجه آن را نیز مشاهده می‌کنید.



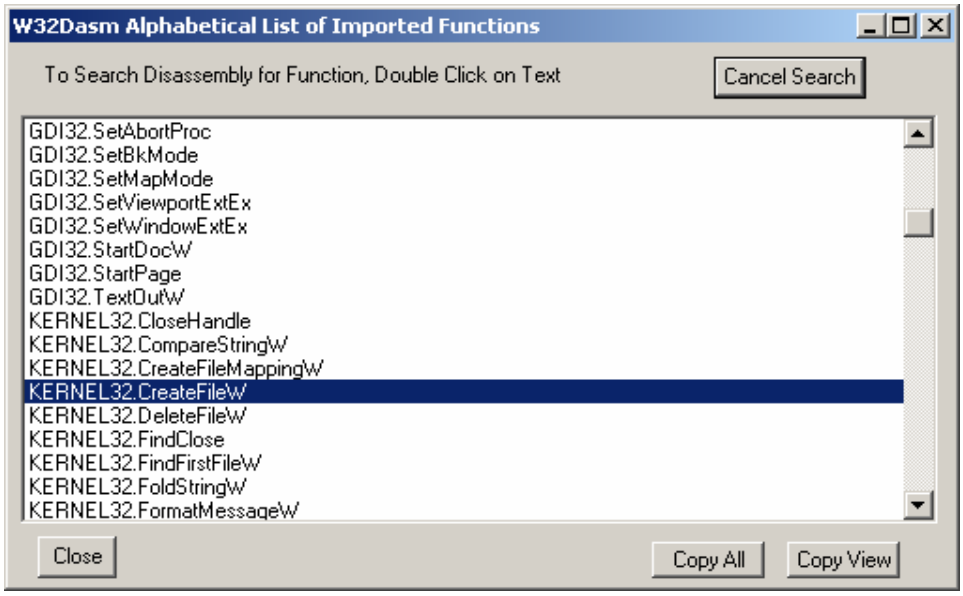


شکل (۳-۳)

۴- برای بازگشت از آخرین دستور پرشی دنبال شده می‌توانید از ترکیب کلیدهای Ctrl + Left استفاده کرده و یا گزینه Return from Last Jump را از منوی Execute text انتخاب کنید.

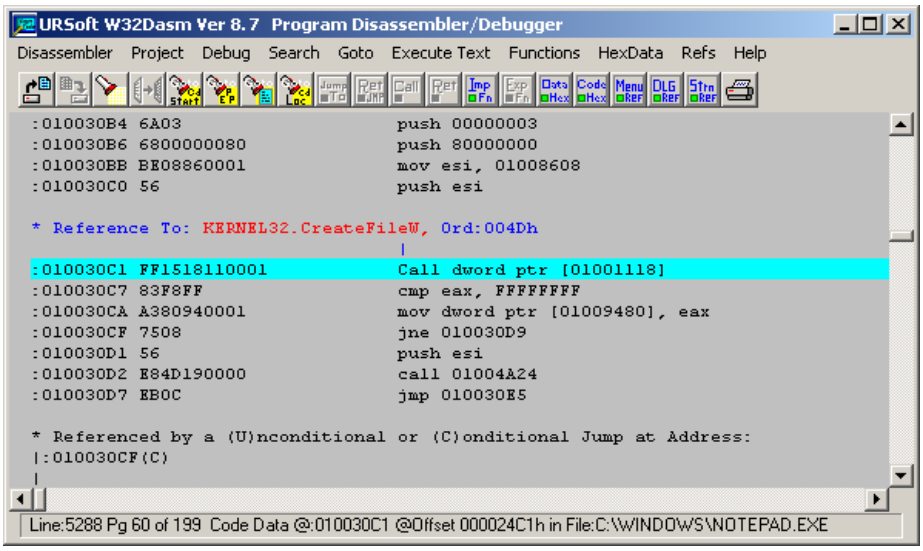
۵- به‌منظور دنبال کردن فراخوانی‌های توابع درحالتی که آدرس قطعی تابع موردنظر مشخص است، می‌توانید با Double Click کردن برروی دستور Call موردنظر، آن را انتخاب کرده و با استفاده از کلید جهت راست و یا انتخاب گزینه Execute Call از منوی Execute Text به آدرس تابع موردنظر منتقل شوید. برای بازگشت از آخرین فراخوانی دنبال شده می توانید از کلید جهت چپ استفاده کرده و یا گزینه Imports از منوی Functions را انتخاب کنید.

۶- به‌منظور بررسی ارجاع‌های صورت گرفته به توابع API ورودی فایل اجرایی، می‌توانید گزینه Imports از منوی Functions را انتخاب کنید. با این کار پنجره Imported functions همانند شکل (۳-٤) باز شده و لیستی ازتوابع API ورودی مورد استفاده فایل اجرایی را به همراه نام کتابخانه مربوطه به نمایش می‌گذارد.



شکل (۳-٤)

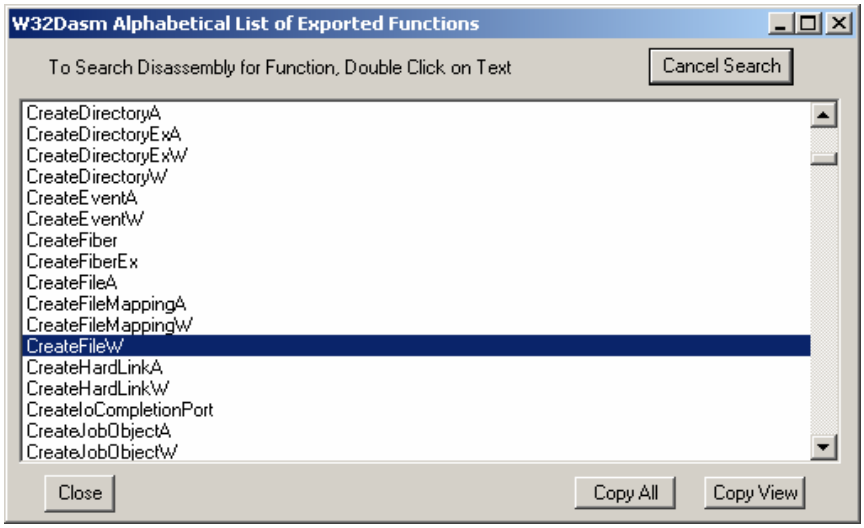
با Double Click کردن برروی نام تابع موردنظر، پنجره اصلی برنامه همانند شکل (۳-۵) به آدرس اولین ارجاع به آن تابع API منتقل خواهد شد. برای بررسی ارجاع‌های بعدی به تابع موردنظر می‌توانید عمل مذکور را دوباره تکرار کنید. با این کار به ترتیب کلیه ارجاع‌های صورت گرفته به یک تابع کنترل می‌شود.



شکل (۳-۵)

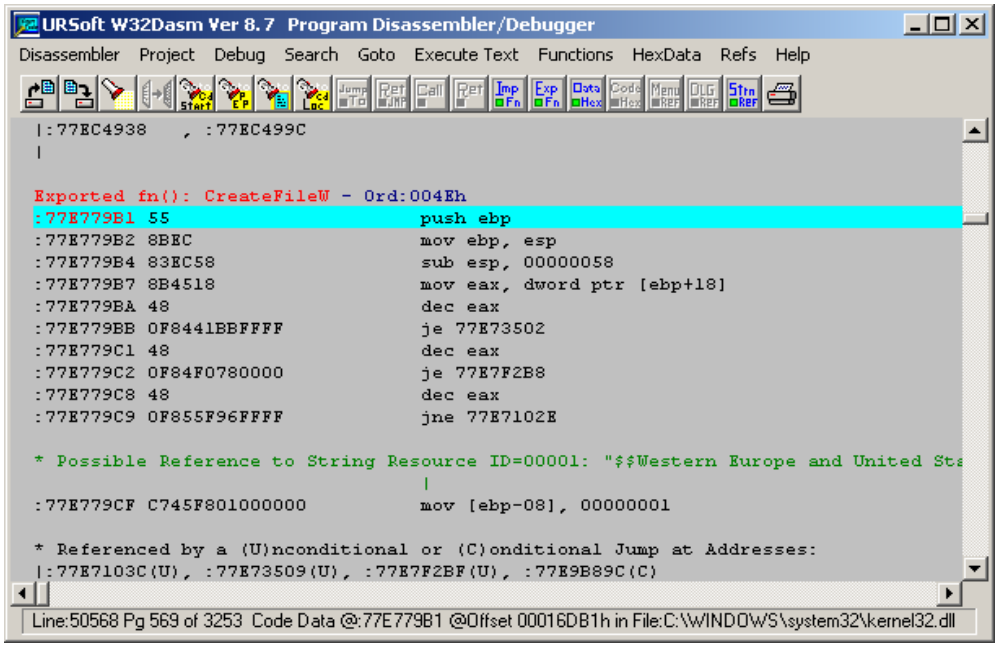
۷- به‌منظور بررسی توابع صادر شده از یک فایل اجرایی می‌توانید گزینه Exports از منوی Functions را انتخاب کنید. با این کار پنجره Exported functions باز شده و لیستی از کلیه توابع صادر شده از طرف فایل اجرایی را به نمایش می‌گذارد.

در شکل (۳-۶) لیست توابع صادر شده از کتابخانه Kernel32.dll را مشاهده می‌کنید.



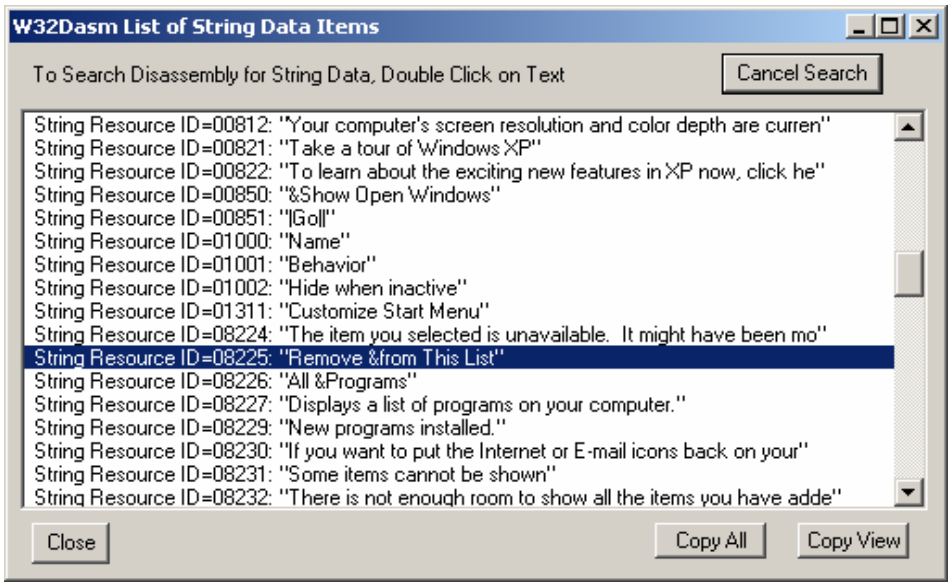
شکل (۳-۶)

۸- با Double Click کردن برروی توابع موجود در لیست ، آدرس شروع تابع موردنظر همانند شکل (۳-۷) در صفحه اصلی برنامه نمایش داده خواهد شد.



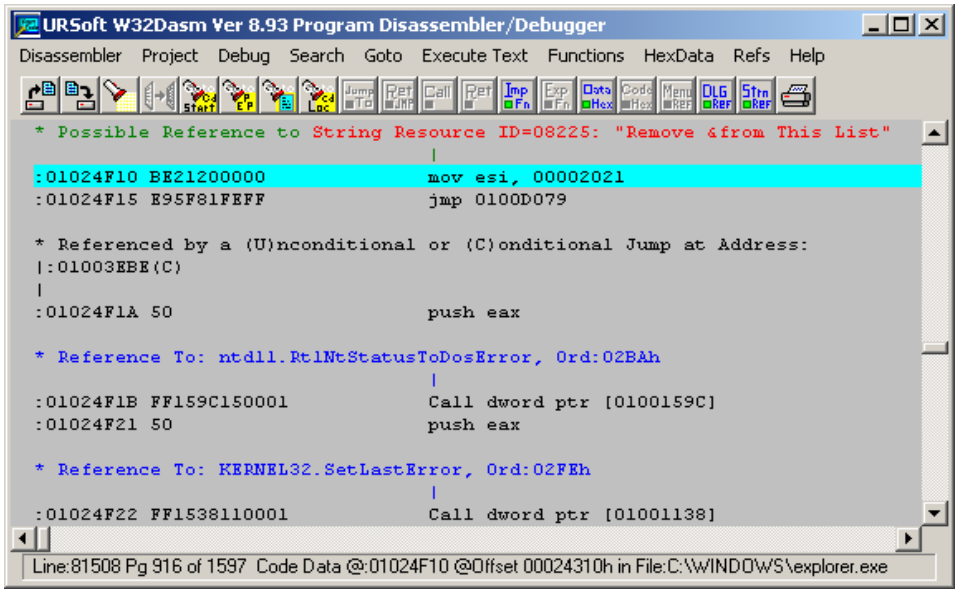
شکل (۳-۷)

۹- به‌منظور بررسی ارجاع‌ها به منابع موجود در فایل اجرایی از قبیل دیالوگ باکس ها، رشته های متنی و منوها می‌توانید از گزینه‌های موجود در منوی Refs استفاده کنید. با انتخاب هر گزینه ، لیستی از منابع مربوطه در فایل اجرایی به نمایش گذاشته می‌شود. به‌عنوان مثال در شکل (۳-۸) لیست نمایش داده شده برای منابع رشته‌ای موجود در فایل اجرایی explorer.exe را مشاهده می‌کنید.



شکل (۳-۸)

با Double Click کردن برروی منبع موردنظر، پنجره اصلی نرم افزارهمانند شکل (۳-۹) به آدرس اولین ارجاع به آن منبع منتقل خواهد شد. برای بررسی ارجاع‌های بعدی به منبع موردنظر می‌توانید عمل مذکور را دوباره تکرار کنید.



شکل (۳-۹)

نرم‌افزار PE Explorer

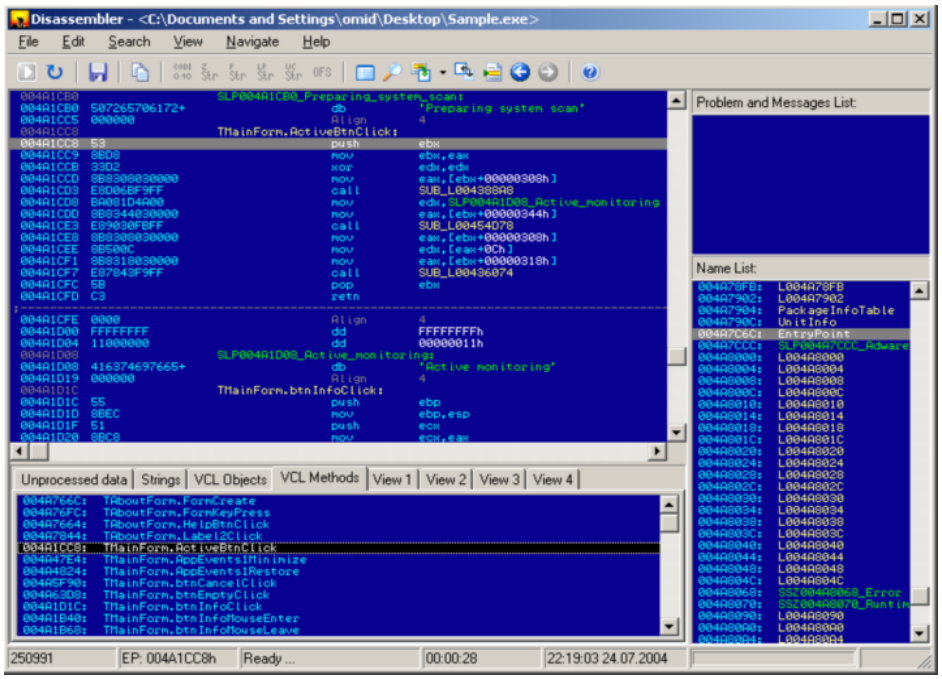
یکی از محبوب‌ترین نرم‌افزارها برای بررسی فایل‌های اجرایی در ویندوز است که از رابط کاربر بسیار خوبی نیز برخوردار است. از قابلیت‌های این نرم‌افزار در فصول قبل استفاده شده و در ادامه نیز از آنها استفاده خواهیم کرد. علاوه بر قابلیت‌های خوب این نرم‌افزار در زمینه بررسی ایستای فایل‌های اجرایی، بخش Disassembler آن نیز بسیار قوی بوده و از کاربری بسیار آسانی برخوردار است. کدهای اسمبلی تولید شده توسط این نرم‌افزار از خوانایی خوبی برخودار بوده و راهنماهای مفیدی را در بردارند و می‌توانند به‌منظور ایجاد قابلیت درک بهتر به کار گرفته شوند.

یکی دیگر از قابلیت‌های بسیار مفید این نرم‌افزار توانایی آن در تحلیل و بررسی ساختارها و اشیاء مورد استفاده در کامپایلرهای شرکت Borland از قبیل C++ Builder و Delphi است. در ادامه آنها را مورد بحث قرار خواهیم داد.

نسخه 1.97 این نرم‌افزار در CD ضمیمه موجود می‌باشد

Tools\PEExplorer

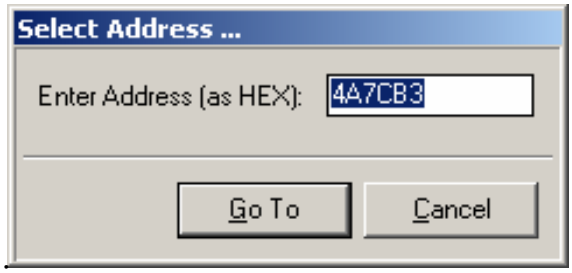
حال که با برخی از مشخصات و امکانات PE Explorer آشنایی نسبی پیدا کردید، نحوه کار با این نرم‌افزار را مورد بررسی قرار خواهیم داد. در شکل (۳-۱۰) نمای کلی از صفحه Disassembler این نرم‌افزار را مشاهده می‌کنید.



شکل (۳-۱۰)

جستجو در کد

برای انتقال صفحه اصلی Disassembler به آدرس مجازی خاصی از فایل اجرایی می‌توانید گزینه Select Address از منوی Navigate را انتخاب کرده و یا از کلیدهای Ctrl + G استفاده کنید. با این عمل پنجره Select Address مطابق شکل (۳-۱۱) نمایش داده شده و امکان درج آدرس موردنظر را به شما می‌دهد.

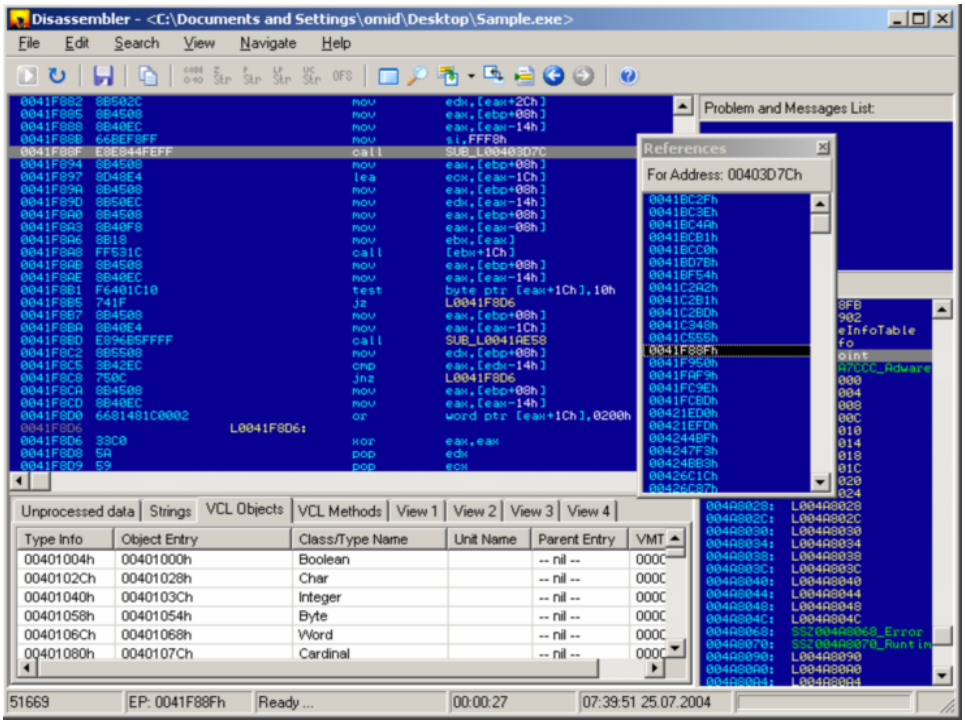


شکل (۳-۱۱)

به‌منظور دنبال کردن آدرس‌های استفاده شده در دستورات اسمبلی مانند JMP , Call و یا سایر دستورات می‌توانید با انتخاب سطر موردنظر و فشردن کلید Enter و یا Double Click کردن برروی آن، به آدرس موردنظر منتقل شوید. در این صورت برای بازگشت به محل اول می‌توانید از کلید Esc استفاده کنید.

بررسی ارجاع ها

به‌منظور بررسی ارجاع‌های مستقیم انجام شده به یک آدرس خاص، ابتدا آن آدرس را انتخاب کرده و سپس کلید R را فشار دهید. با این عمل پنجره References باز شده و لیستی از ارجاع‌های انجام شده به آدرس موردنظر را نمایش می‌دهد. در صورت نیاز برای انتقال صفحه اصلی به محل مراجعه‌ها می‌توانید برروی ارجاع موردنظر در لیست References ، Double Click کنید. در شکل (۳-۱۲) پنجره Disassembler را در حال نمایش ارجاع‌های انجام شده به آدرس مجازی 403D7C مشاهده می‌کنید.



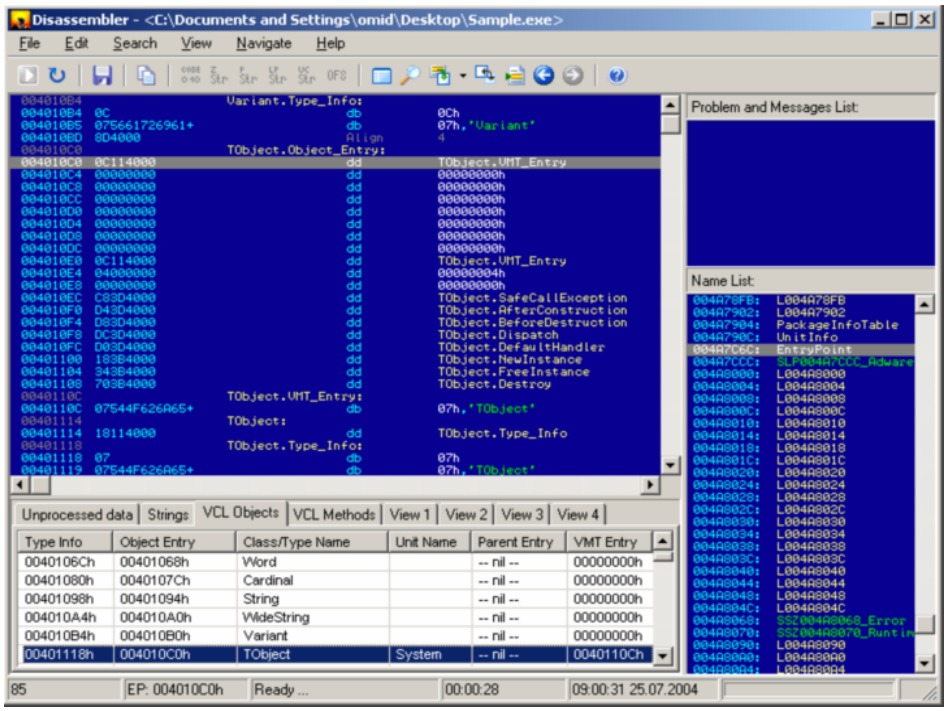
شکل (۳-۱۲)

بررسی اشیاء VCL

VCL (Virtual Component Library) در حقیقت کتابخانه‌ای از اشیاء استاندارد مورد استفاده در فایل‌های اجرایی تولید شده به وسیله کامپایلرهای C++ Builder و Delphi ساخت شرکت Borland است. با توجه به ساختار خاص فایل‌های اجرایی مذکور، کلیه انواع داده‌ها، متغیرها، کنترل‌ها بصری، خطاها و سایر اجزاء مورد نیاز، به‌صورت اشیاء متفاوتی در کتابخانه VCL تعریف شده‌اند. این کتابخانه در هنگام کامپایل شدن به‌صورت ایستا به فایل اجرایی لینک شده و در حقیقت درون آن قرار می‌گیرد.

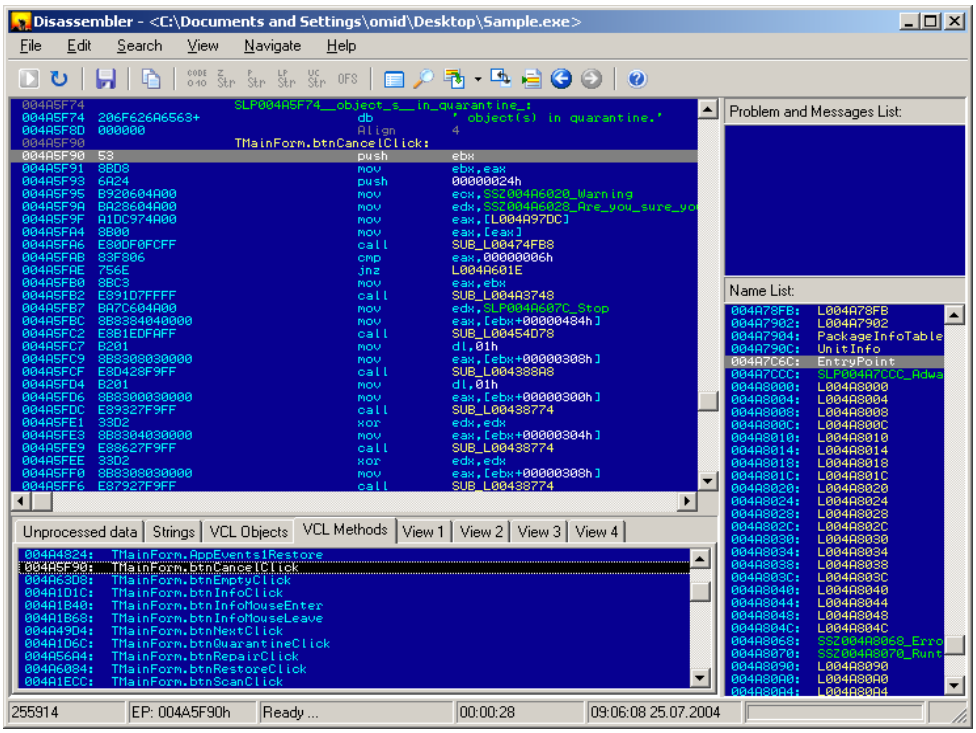
همچنین این کامپایلرها ساختار منظمی را برای درج متدهای پاسخ گویی به رویدادهای ایجاد شده (Event handlers) برای اشیاء VCL در فایل اجرایی درنظر گرفته‌اند که می‌تواند راهنمای خوبی برای دستیابی به درک صحیح‌تر از نحوه عملکرد فایل اجرایی موردنظر باشد.

همان‌طور که ذکر شد PE Explorer کتابخانه‌ها و متدهای پاسخ‌گویی به رویدادهای VCL را در فایل اجرایی شناسایی کرده و با توجه به ذکر نام هر یک، به کاربرامکان دنبال کردن تعاریف اشیاء و متدهای VCL موردنظر را می‌دهد. با بررسی ارجاع‌های انجام شده به اشیاء VCL در فایل اجرایی می‌توانید نکات بسیار مفیدی را راجع به نحوه عملکرد و روش‌های مورد استفاده در این نوع فایل‌های اجرایی بدست آورید. شکل (۳-۱۳) پنجره Disassembler را در حال نمایش تعریف شیء TObject در فایل اجرایی sample.exe نشان می‌دهد.



شکل (۳-۱۳)

شکل (۳-۱٤) پنجره Disassembler را درحال نمایش کدهای پاسخ‌گویی به رویداد Click برروی دکمه Cancel از فرمی با نام TMainForm در فایل اجرایی sample.exe نشان می‌دهد.



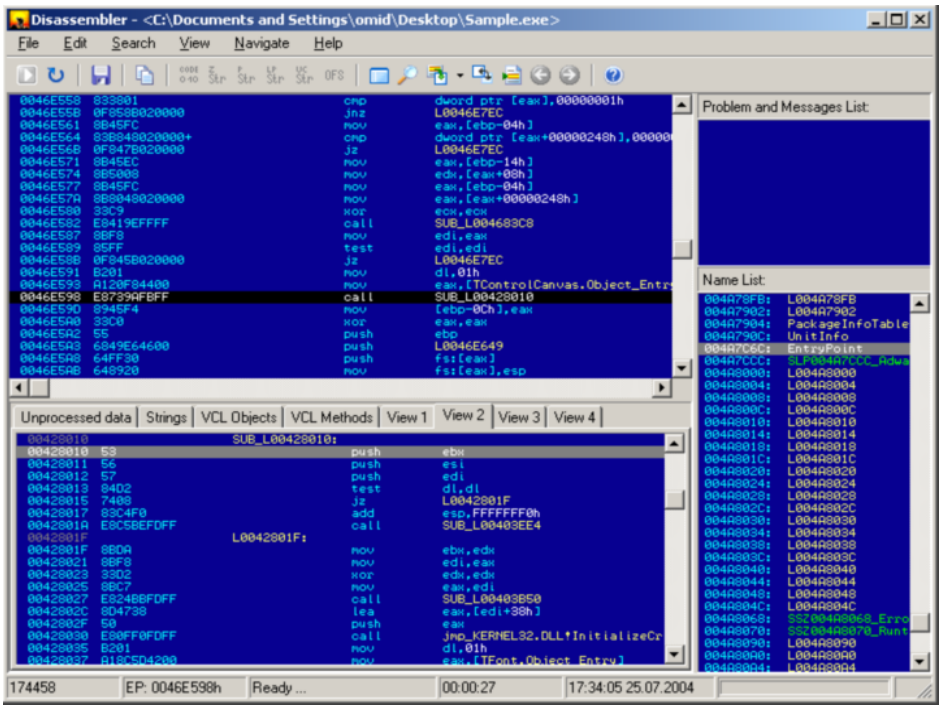
شکل (۳-۱۴)

استفاده از نماها

در هنگام جستجو در کدهای اسمبلی با توجه به حجم زیاد ارجاع‌ها و آدرس‌های مجازی استفاده شده در دستورات، استفاده از یک پنجره، برای نمایش آنها کاری بسیار دشوار است و سرعت بررسی کد را بسیار پایین می‌آورد.

این نرم‌افزار برای کمک به کاربر، از پنج نمای متفاوت استفاده می‌کند که یکی از آنها نمای اصلی و چهارتای دیگر به‌عنوان نماهای کمکی و با نام‌های View 1 ~ 4 در پایین صفحه قرار دارند. با استفاده از این نماها می‌توانید در یک زمان حداکثر برروی پنج محدوده متفاوت از کد اسمبلی تسلط داشته و در صورت نیاز از آنها استفاده کنید. به این منظور، دستور موردنظر را که در آن به آدرس مجازی خاصی ارجاع شده است انتخاب کرده و سپس برای نمایش آدرس ارجاع شده، در یکی از پنجره‌های view، از کلیدهای F6 تا F9 برای Viewهای ۱ تا ٤ استفاده کنید.

به‌عنوان مثال در شکل (۳-۱۵) با انتخاب دستور Call SUB_L00428010 و فشردن کلید F7، View2، آدرس مجازی 428010 از فایل اجرایی را در خود نمایش خواهد داد.



شکل (۳-۱۵)

در صورت نیاز برای جابجا کردن یک نمای کمکی با نمای اصلی می‌توانید از گزینه Swap Current View از منوی View استفاده کرده و یا کلید F5 را فشار دهید. در حقیقت با این عمل محدوده نمایش داده شده در نمای اصلی با محدوده نمایش داده شده در نمای کمکی تعویض می‌گردد.

نرم‌افزار Interactive Disassembler (IDA Pro)

قوی‌ترین و کامل‌ترین نرم‌افزار موجود برای Disassembler کردن و تحلیل فایل‌های اجرایی است که ابزارهای قدرتمند و منحصر به فردی را برای تحلیل و بررسی کدهای اسمبلی تولید شده در نظر گرفته است. با توجه به تشخیص اکثر ساختارها و توابع استاندارد مورد استفاده کامپایلرهای امروزی، و نیز تحلیل و بررسی بسیار دقیق کدهای اسمبلی، اطلاعات خروجی این نرم‌افزار از خوانایی بسیار خوبی برخوردار بوده و راهنماهای متعددی را نیز در خود دارند که می‌توانند به منظور ایجاد قابلیت درک بهتر از کدهای اسمبلی تولید شده به کار گرفته شوند.

نمودارها و گراف‌های تولید شده توسط این نرم افزار روند شناسایی اجزاء و روابط آنها را بسیار تسهیل کرده و باعث صرفه‌جویی در وقت و کاهش خطاهای احتمالی در مراحل جمع‌آوری اطلاعات و شناسایی اجزاء می‌گردد.

این نرم‌افزار توانایی Disassemble کردن و تحلیل فایل‌های اجرایی برای پردازنده‌های متداول امروزی را دارد که از آن جمله می‌توان به پردازنده‌های X86 ، Z80 و یا حتی پردازنده‌های مجازی JAVA و Net. اشاره کرد.

نسخه 4.6 این نرم‌افزار در CD ضمیمه موجود می‌باشد

Tools\IDAPro

توجه داشته باشید که نسخه آزمایشی این نرم‌افزار دارای محدودیت‌های زیر است:

۱- تنها پردازشگر قابل پشتیبانی، پردازشگرهای مدل 80x86 هستند.

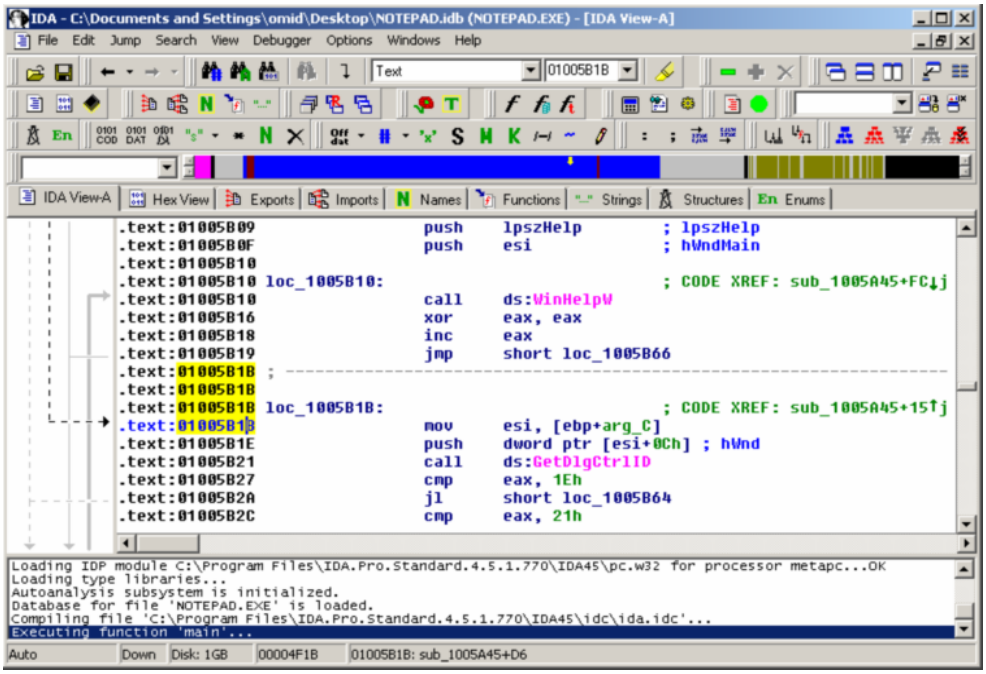
۲- تنها فرمت فایل اجرایی مورد قبول فرمت PE در Win 32 است که فرمت استاندارد فایل‌های اجرایی ویندوزهای 32 بیتی محسوب می‌گردد.

۳- تنها فایل‌های اجرایی Widows GUI که از رابط کاربر استاندارد و ویندوز استفاده می‌کنند قابل پشتیبانی هستند.

٤- توانایی ذخیره کردن پروژه در این نسخه وجود ندارد.

٥- برای استفاده از این نسخه محدودیت زمانی در نظر گرفته شده است.

حال که با برخی از خصوصیات و امکانات این نرم‌افزار آشنایی نسبی پیدا کردید، بهتر است به بررسی نحوه کار با آن بپردازیم. در شکل (۳-۱۶) ، صفحه اصلی این نرم‌افزار را در حال کار بر روی فایل اجرای notepad.exe مشاهده می‌کنید.



شکل (۳-۱۶)

مدیریت پروژه

با توجه به قابلیت‌های این نرم‌افزار در زمینه تصحیح کدهای Disassemble شده توسط کاربر و ایجاد نام‌ها و اختصاص آن به آدرس‌ها، داده‌ها و ...، سیستم ذخیره‌سازی و مدیریت پروژه کاملی برای فایل‌های Disassemble شده در نظر گرفته شده است.

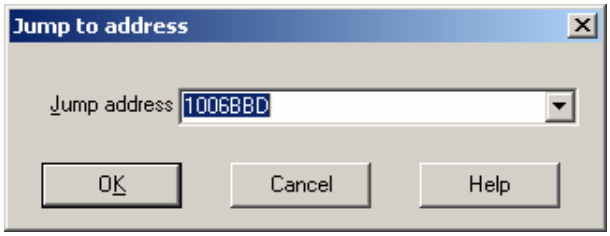
به‌طور معمول این نرم‌افزار از چندین فایل برای ذخیره پروژهٔ جاری استفاده می‌کند ولی در صورت نیاز توانایی ذخیره کامل پروژه در یک فایل را دارد. این فایل که با نام پایگاه داده پروژه و پسوند idb شناخته می‌شود می‌تواند وضعیت فعلی پروژه را در خود ذخیره کرده و در صورت نیاز آن را بازگردانی کند.

به‌منظور ذخیره پروژه فعلی می‌توانید از گزینه‌های save و یا save as در منوی File استفاده کنید. توجه داشته باشید که در این حالت پروژه به‌صورت یک فایل idb ذخیره می‌شود.

جستجو در کد

همان‌طور که ذکر شد نرم‌افزارهای Disassembler امکانات و قابلیت‌های متعددی را به‌منظور تسهیل روند بررسی کدها از طریق دنبال کردن فراخوانی‌ها، پرش‌ها و ارجاع‌های متعدد موجود در کد اسمبلی ارائه می‌کنند. IDA نیز امکانات وسیعی را در این زمینه ارائه می‌دهد که در زیر به برخی از آنها اشاره خواهیم کرد.

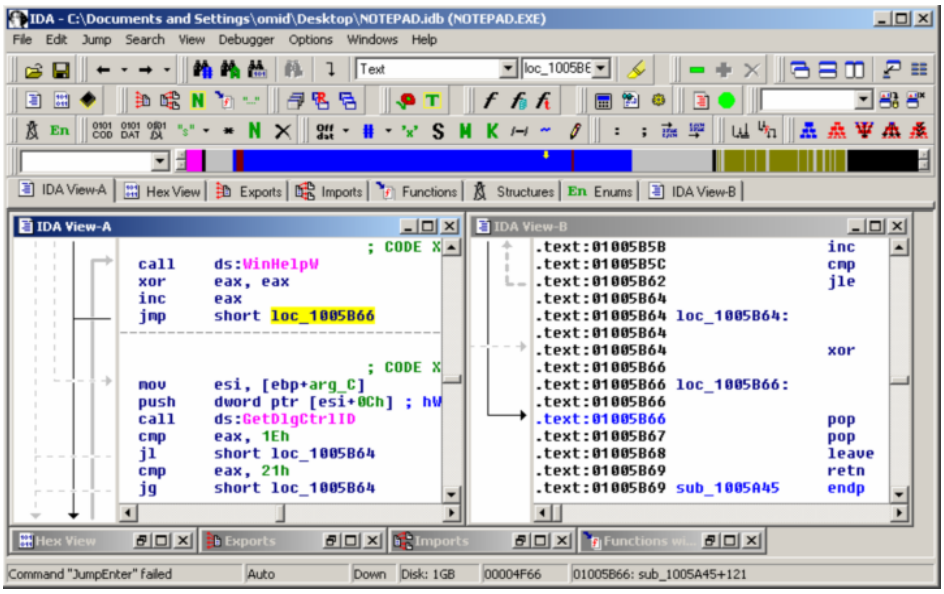
۱- به‌منظور انتقال صفحه Disassembler به آدرس مجازی خاصی از فایل اجرایی می‌توانید گزینه jump to address از منوی JUMP را انتخاب کرده و یا کلید G را فشار دهید با این کار پنجره Jump to address مانند شکل (۳-۱۷) نمایش داده شده و اجازه درج آدرس مجازی موردنظر را به کاربر می‌دهد.



شکل (۳-۱۷)

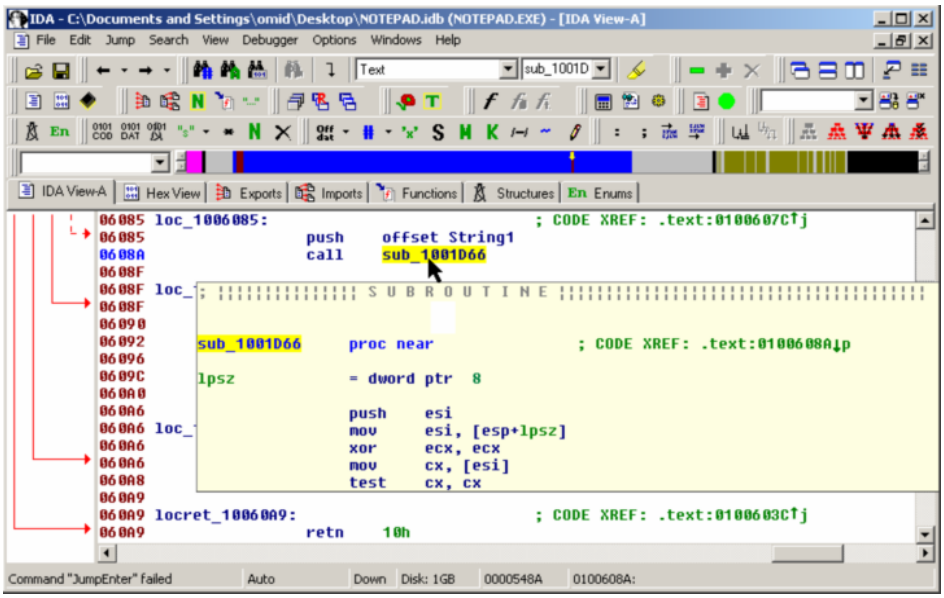
۲- برای دنبال کردن آدرس‌های استفاده شده در دستورات اسمبلی مانند JMP، call و یا سایر دستورات می‌توانید با انتخاب آدرس استفاده شده در دستور موردنظر و فشردن کلید Enter به آدرس مذکور منتقل شوید در این صورت برای بازگشت به محل اول می‌توانید از کلید Esc استفاده کنید.

در صورت نیاز می‌توانید آدرس مذکور را در صفحه جدیدی نمایش دهید. به این منظور همانند روند فوق عملکرده و کلید Alt + Enter را فشاردهید. با این عمل پنجره جدیدی باز شده و آدرس موردنظر را در خود نمایش خواهد داد. این امر باعث جلوگیری از سردرگمی و تسریع در دنبال کردن آدرس‌ها می‌گردد. در مثال شکل (۳-۱۸)، آدرس استفاده شده در دستور Jmp Short Loc_1005B66 به روش مذکور دنبال شده و آدرس مجازی 1005B66 در نمای جدیدی با نام View-B نمایش داده شده است.



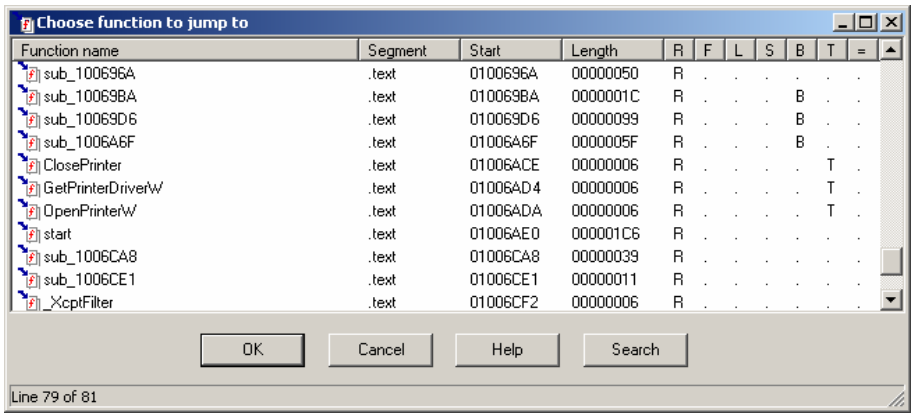
شکل (۳-۱۸)

به‌منظور ایجاد قابلیت بازبینی سریع آدرس‌ها در کدهای اسمبلی، این نرم‌افزار امکانات بسیار خوبی را درنظر گرفته است. به عنوان مثال برای مشاهده و دنبال کردن آدرس‌های مجازی استفاده شده در کدهای اسمبلی، تنها کافی است که اشاره گر ماوس را روی آنها نگه دارید. با این عمل پنجره کوچکی همانند شکل (۳-۱۹) در پایین اشاره گر ظاهر شده و کدهای اسمبلی را در آدرس مذکور به نمایش خواهد گذاشت. در صورت نیاز برای تغییر محدوده دید این پنجره می‌توانید از کنترل‌های Scroll ماوس خود استفاده کنید.



شکل (۳-۱۹)

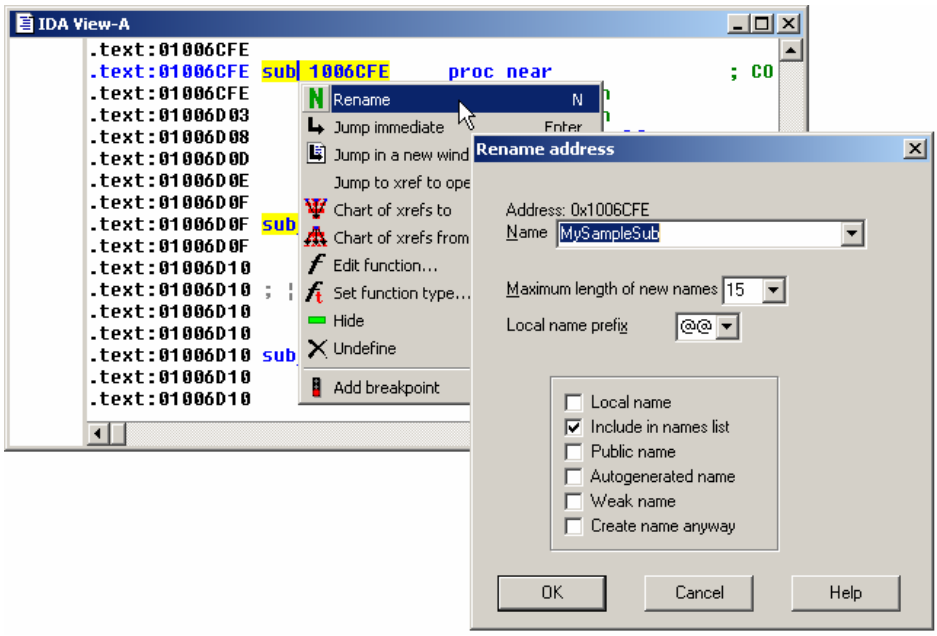
۳- در صورت نیاز به بررسی توابع موجود در کد اسمبلی، می توانید از گزینه Jump to functions از منوی Jump استفاده کرده ویا کلیدهای Ctrl + P را فشار دهید. با این عمل پنجره Jump همانند شکل (۳-۲۰) ظاهر شده و لیستی از توابع موجود را به نمایش می‌گذارد.



شکل (۳-۲۰)

با Double Click کردن برروی هر یک از توابع موجود در لیست می‌توانید به محل تعریف آن تابع در کد اسمبلی منتقل شوید. به‌طور پیش‌فرض، IDA در صورت عدم شناسایی، از آدرس ورودی

تابع به‌منظور ساخت نام آن استفاده می‌کند ولی در صورت نیاز می توانید نام توابع را به دلخواه تغییر دهید. این امر باعث خوانایی بیشتر و تسریع در دنبال کردن توابع و ارجاع‌ها در کد اسمبلی می‌گردد. برای این منظور می‌توانید برروی نام تابع در محل تعریف آن کلیک راست کرده و از منو، آیتم Rename را انتخاب کنید. با این عمل پنجره Rename address مطابق شکل (۳-۲۱) نمایش داده شده و به شما اجازه تعیین نام دلخواه برای تابع موردنظر را می‌دهد.

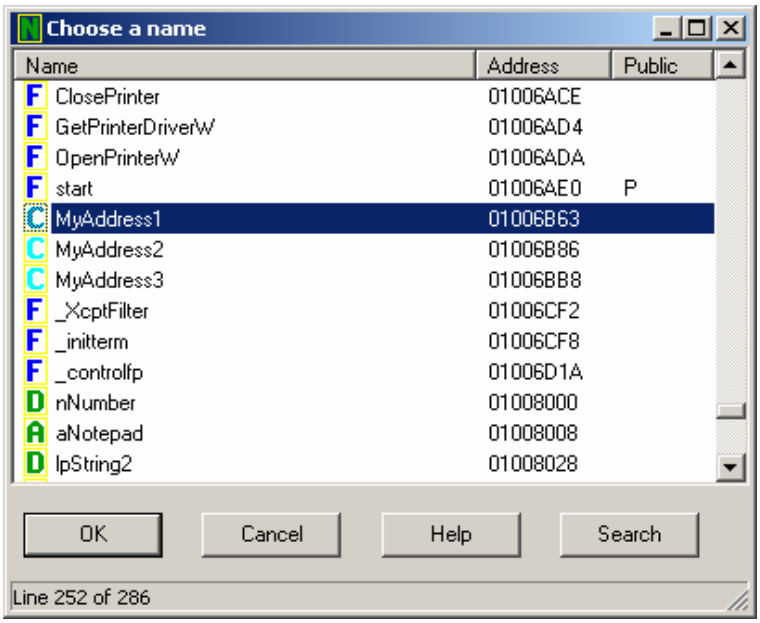


شکل (۳-۲۱)

همان‌طور که مشاهده می‌کنید در این صفحه گزینه‌هایی نیز برای نام گذاری خودکار و نحوه رفع تداخل در نام‌گذاری‌ها وجود دارد که در صورت نیاز می‌توانید از آنها استفاده کنید.

۴- در صورت نیاز می‌توانید برای آدرس‌های مجازی دلخواه از کد اسمبلی نام‌هایی را تعیین کرده و در صورت لزوم به آنها رجوع کنید. این امر باعث ایجاد تسریع در روند شناسایی اجزاء و نحوه علمکرد یک فایل اجرایی می‌گردد. به این منظور ابتدا آدرس موردنظر را انتخاب کرده و همانند مثال قبل با کلیک راست برروی آن و انتخاب گزینه Rename از منو، نام موردنظر را به آن آدرس نسبت دهید.

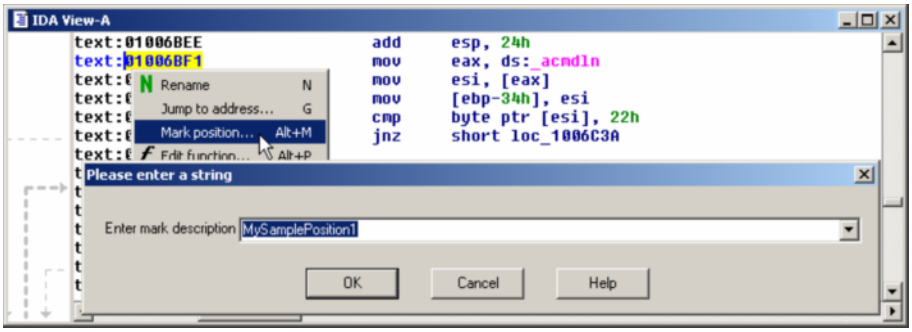
در صورت نیاز برای ارجاع به آدرس مذکور می‌توانید گزینه Jump by name از منوی Jump را انتخاب کرده ویا کلیدهای Ctrl + L را فشار دهید. با این عمل پنجره Choose a Name همانند شکل (۳-۲۲) ظاهر شده و به شما اجازه انتخاب و مراجعه به آدرس‌های نام گذاری شده را می‌دهد.



شکل (۳-۲۲)

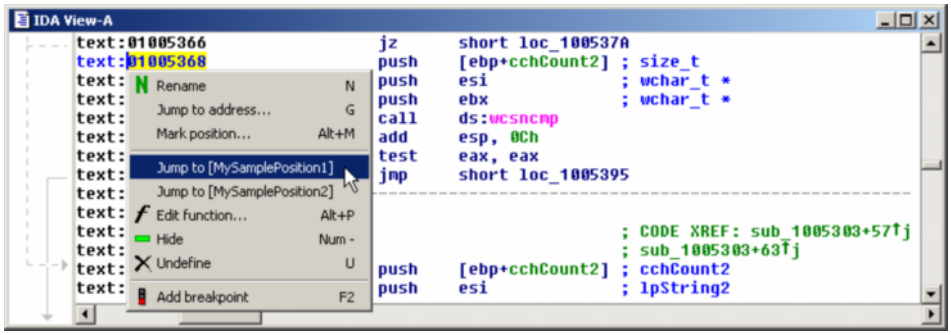
همان‌طور که در مشاهده می‌کنید، IDA نیز به نوبه خود با توجه به تحلیل‌هایی که برروی کدها انجام می‌دهد، نام‌های مفیدی را برای قسمت‌های مختلف کد و داده ازقبیل توابع، دستورالعمل‌ها، رشته‌ها و توابع ورودی در نظر می‌گیرد.

به‌منظورارجاع و بررسی سریع‌تر آدرس‌های متداول در کد اسمبلی ابتدا آدرس موردنظر را انتخاب کرده، برروی آن کلیک راست کنید و از منوی نمایش داده شده، گزینه Mark Position را انتخاب کنید. با این عمل پنجره‌ای مطابق شکل (۳-۲۳) ظاهر شده و به شما امکان تعیین نام موردنظر را برای آدرس مذکور می‌دهد.



شکل (۳-۲۳)

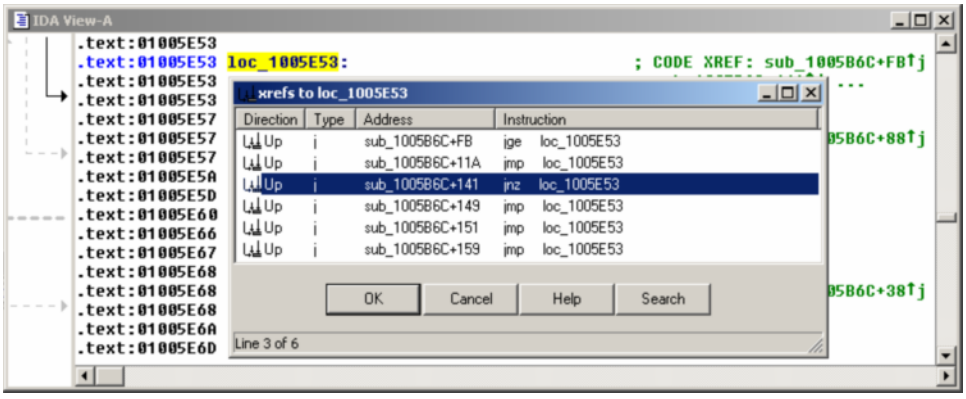
در صورت نیاز برای مراجعه به آدرس مذکور می‌توانید برروی آدرس دلخواهی کلیک راست کرده و از منوی نمایش داده شده، نام موردنظر را همانند شکل (۳-۲٤) انتخاب کنید. با این عمل پنجره Disassembler به آن آدرس منتقل می‌شود.



شکل (۳-۲٤)

بررسی ارجاع‌ها

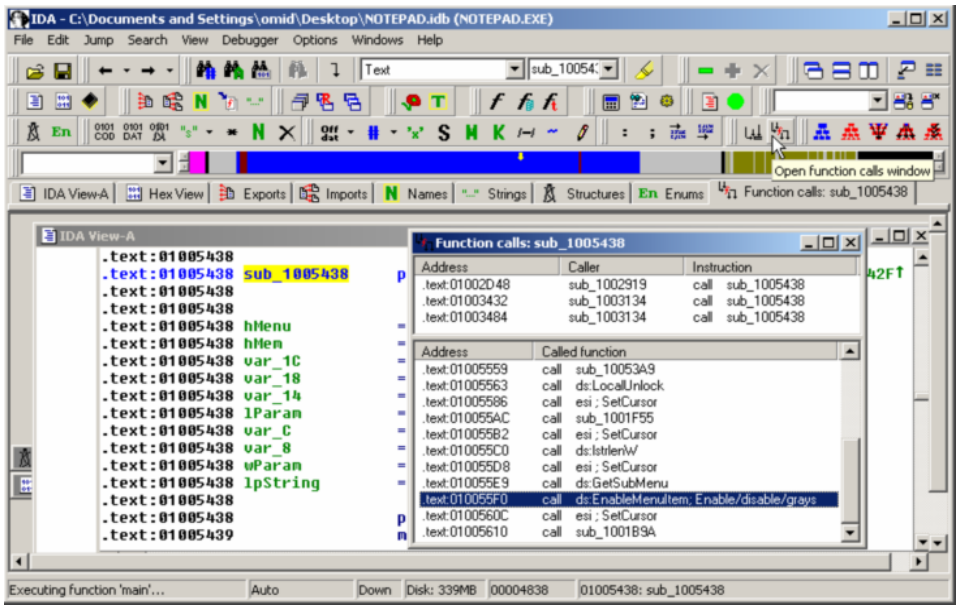
به‌منظور بررسی ارجاع‌های مستقیم انجام شده به یک آدرس خاص، ابتدا آن آدرس را انتخاب کرده و سپس کلید X را فشار دهید. با این عمل پنجره xrefs مشابه شکل (۳-۲۵) باز شده و لیستی از ارجاع‌های انجام شده به آدرس موردنظر را نمایش می‌دهد. با Double Click کردن برروی هر یک می‌توانید به محل آن ارجاع در کد اسمبلی منتقل شوید.



شکل (۳-۲۵)

در صورت نیاز می‌توانید لیستی از فراخوانی‌های انجام شده توسط یک تابع و یا فراخوانی‌های انجام شده از آن را دریافت کنید. به این منظور ابتدا تابع موردنظر را انتخاب کرده و سپس برروی دکمه Open Function Calls واقع در نوار ابزار کلیک کنید. با این عمل پنجره Function Call همانند

شکل (۳-۲۶) نمایش داده شده و لیستی از فراخوانی‌های انجام شده توسط تابع مذکور را به همراه فراخوانی‌های انجام شده از آن به نمایش می‌گذارد.



شکل (۳-۲۶)

در قسمت بعد از امکانات ترسیمی و گراف‌های تولید شده توسط این نرم‌افزار به‌منظور بررسی و تحلیل بهتر روند اجرایی برنامه‌ها و دنبال کردن ارجاع‌های انجام شده استفاده خواهیم کرد.

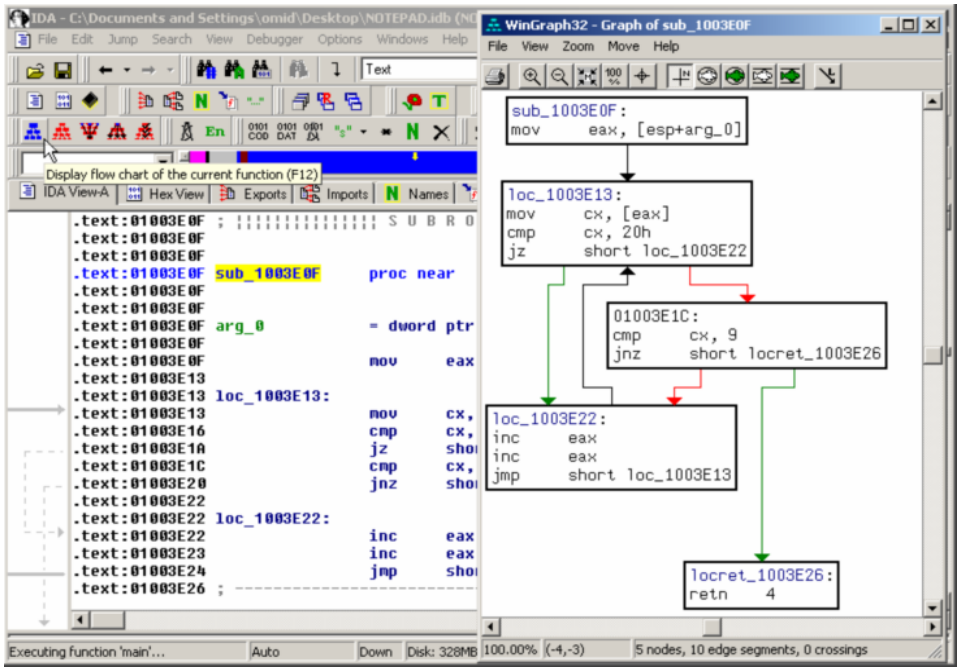
گراف‌ها و نمودارها

یکی از امکانات بسیار مفید این نرم‌افزار، توانایی آن در ایجاد گراف‌ها و نمودارهایی به‌منظور نمایش هر چه بهتر فراخوانی‌ها، ارجاع‌ها و روند اجرایی فایل اجرایی است. تحلیل و بررسی گراف‌ها و نمودارهای ایجاد شده می‌تواند ما را در دستیابی به طرحی جامع و کامل از نحوه عملکرد یک نرم‌افزارو اجزاء آن یاری کند.

۱- فلوچارت‌ها

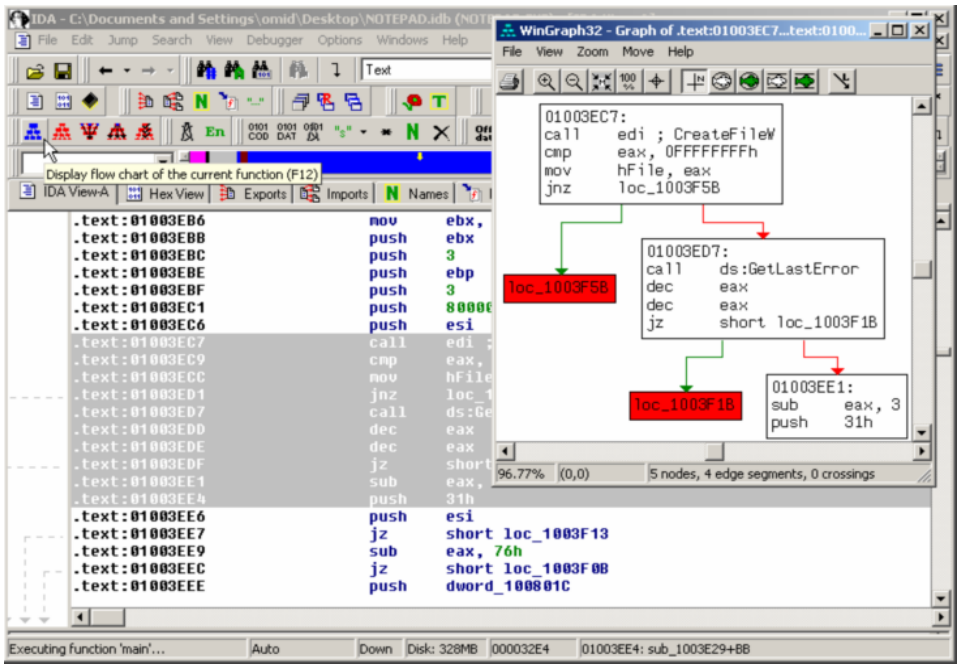
همان‌طور که می‌دانید از فلوچارت‌ها به‌منظور نمایش بهتر نحوه عملکرد یک برنامه استفاده می‌شود. این عمل معمولاً به وسیله دنبال کردن و نمایش پرش‌ها و شرط‌ها انجام می‌گیرد. بی‌تردید دنبال کردن و بررسی روند اجرایی و نحوه عملکرد یک فایل اجرایی به وسیله فلوچارت‌ها بسیار ساده‌تر از حالت معمول بوده و به صرف وقت کمتری نیز نیاز دارد.

به‌منظور ایجاد فلوچارت، ابتدا آدرس شروع را برای رسم آن انتخاب کرده و سپس گزینه Flow Chart را از منوی View انتخاب کنید. همان‌طور که در شکل (۳-۲۷) مشاهده می‌کنید، با این عمل، نرم‌افزار Wingraph32 اجرا شده و روند اجرایی را از آدرس انتخاب شده تا پایان آن به‌صورت فلوچارت به نمایش می‌گذارد.



شکل (۳-۲۷)

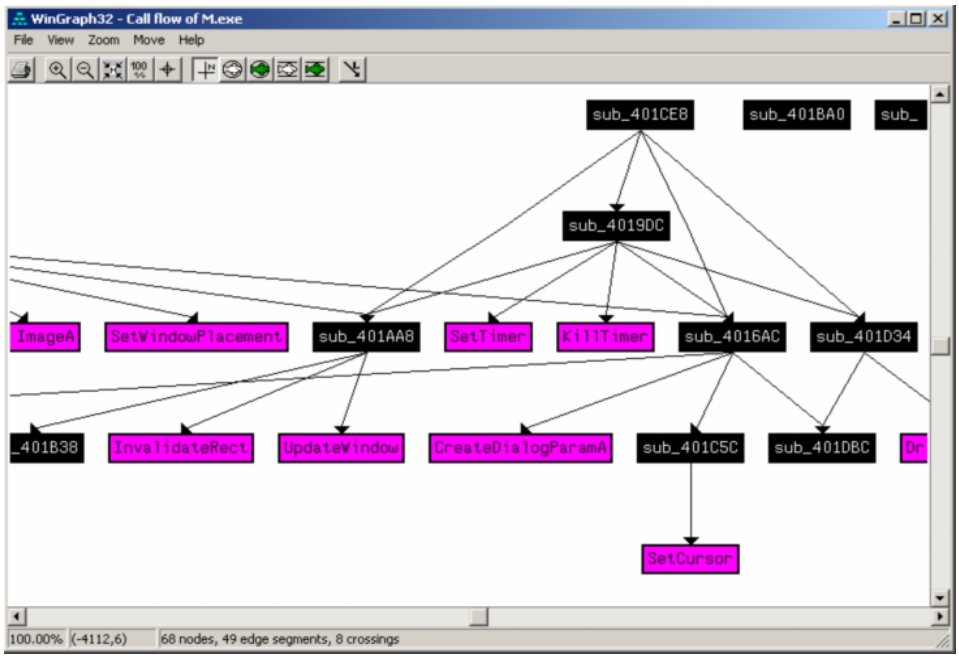
در صورت نیاز می‌توانید عملیات رسم فلوچارت را برروی قطعه مشخصی از کد اسمبلی نمایش داده شده انجام دهید. به این منظور ابتدا همانند شکل (۳-۲۸) محدوده موردنظر را انتخاب کرده و سپس از گزینه Flow Chart واقع در منوی View استفاده کنید. همان‌طور که مشاهده می‌کنید با این روش نرم‌افزار wingraph 32 تنها محدوده مشخص شده را مورد بررسی قرار می دهد.



شکل (۳-۲۸)

۲- نمودار کلی فراخوانی‌ها

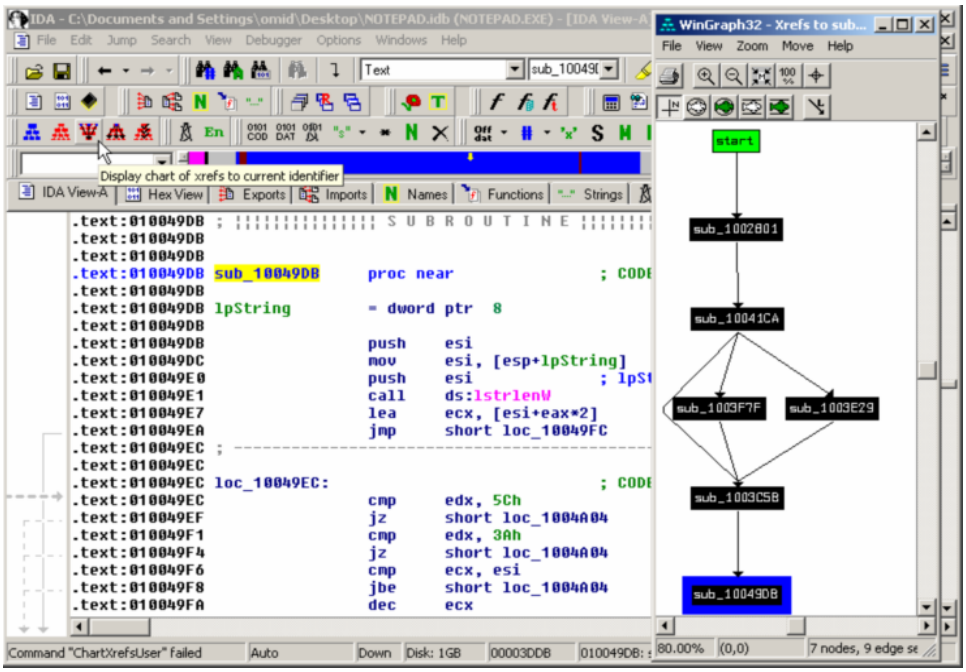
در صورت نیاز می‌توانید نموداری از کلیه فراخوانی‌های انجام شده در فایل اجرایی را مشاهده کنید. به این منظور گزینه Function Calls را از منوی View انتخاب کنید. همان‌طور که در شکل (۳-۲۹) مشاهده می‌کنید. با این عمل نرم‌افزار wingraph32 نموداری از کلیه فراخوانی‌های انجام شده در کد اسمبلی را به نمایش خواهد گذاشت.



شکل (۳-۲۹)

۳- نمودار فراخوانی‌ها از توابع

در صورت نیاز می‌توانید نموداری سلسله مراتبی از فراخوانی‌های انجام شده از شروع فایل اجرایی تا توابع خاصی را به‌صورت نموداری مشاهده کنید. به این منظور می‌توانید آدرس شروع یک تابع را انتخاب کرده واز گزینه xrefs to واقع در منوی View استفاده کنید. با این عمل، نرم‌افزار Wingraph32 همانند شکل (۳-۳۰) نموداری از فراخوانی‌های انجام شده از مبدا تا تابع موردنظر را نمایش خواهد داد.

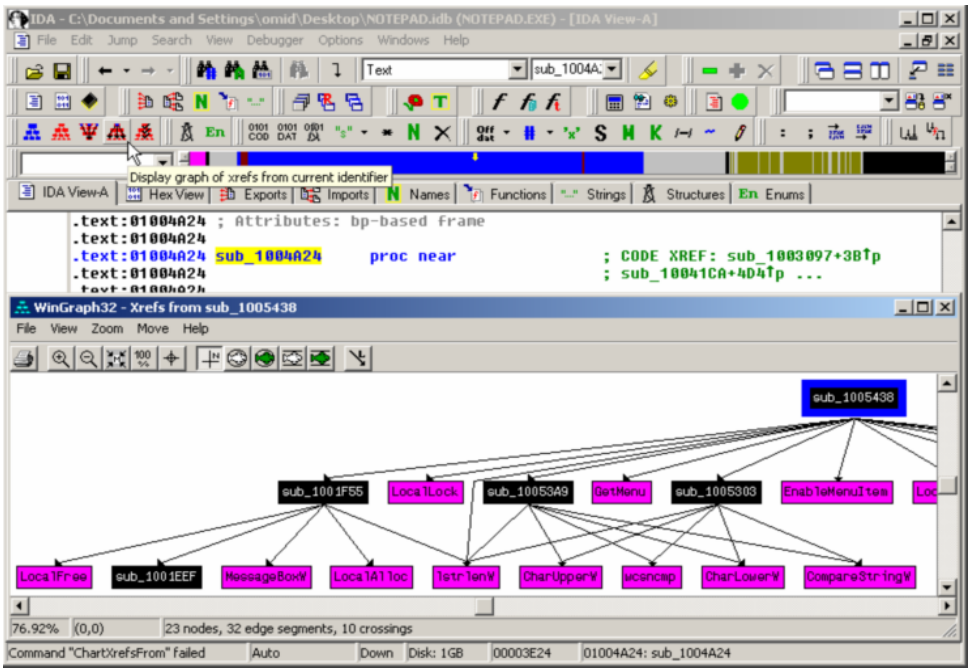


شکل (۳-۳۰)

در صورت نیاز با استفاده از روش ذکر شده در قسمت فلوچارت‌ها، می‌توانید چندین تابع را در نمودار ترسیم شده مورد بررسی قرار دهید.

۴- نمودار فراخوانی‌های انجام شده به وسیله توابع

در صورت نیاز می‌توانید نموداری سلسله مراتبی از فراخوانی‌های انجام شده توسط تابع و یا توابع موردنظر را مشاهده کنید. به این منظور می‌توانید آدرس شروع تابع موردنظر را انتخاب کرده و از گزینه Xrefs From واقع در منوی View استفاده کنید. با انجام این عمل نرم‌افزار Wingraph32 مشابه شکل (۳-۳۱) نموداری سلسله مراتبی از فراخوانی‌های انجام شده توسط تابع موردنظر را به نمایش خواهد گذاشت.



شکل (۳-۳۱)

در صورت نیاز با استفاده از روش ذکر شده در قسمت فلوچارت‌ها، می‌توانید چندین تابع را در نمودار ترسیم شده مورد بررسی قراردهید.

فصل چهارم

Decompiler ها

فصل چهارم

Decompiler ها

درحقیقت Decompilerها ابزارهایی پرقدرت و سودمند جهت تجزیه و تحلیل کدهای Disassemble شده هستند. هدف نهایی تمام آنها تبدیل این کدها به برنامه‌هایی قابل درک برای انسان است. با توجه به‌ساختارهای گوناگونی که درکامپایلرهای مختلف جهت کامپایل‌کردن برنامه‌ها به‌زبان ماشین مربوطه به کار گرفته می‌شود، هریک از این ابزارها نیز جهت تفسیر و Decompile کردن نوع و یا نواع خاصی از کامپایلرها طراحی و پیاده‌سازی می‌شوند. میزان موفقیت Decompilerها تا حدود زیادی بستگی به روش‌های به کار گرفته شده در کامپایلر مربوطه برای کامپایل کردن کدها دارد. هر چه فایل‌های کامپایل شده از نظر سرعت اجرا بهینه‌تر بوده و در مراحل کامپایل دستخوش تغییرات بیشتری شده باشند، عملیات لازم برای Decompile کردن آنها پیچیده‌تر بوده و از درصد موفقیت کمتری نیز برخوردار است. در حقیقت بین پیچیدگی کامپایلر و درصد موفقیت Decompilerهای مربوطه معمولاً رابطه معکوس برقرار است.

در این بخش برخی از Decompilerهای موفق موجود برای کامپایلرهای امروزی و نحوه عملکرد هر یک را مورد بررسی قرار خواهیم داد.

C / C++ Decompliers

همان‌طور که می‌انید امروزه از زبان‌های C/C++ به‌طور بسیار وسیعی در طراحی و ساخت نرم‌افزارهای گوناگون برای سیستم‌های عامل و سخت‌افزارهای مختلف استفاده می‌شود به دلیل پیچیدگی‌های فروان کامپایلرهای این زبان‌ها و نیز تغییرات بسیار زیادی که به‌منظور ساخت کدهای ماشین سریع‌تر و کارآمدتر برروی برنامه‌های نوشته شده انجام می‌دهند، ابزارهای موجود برای Decompile کردن این کدها بسیار معدود بوده و از کارآیی کافی نیز برخوردار نیستند. با این وجود این ابزارها می‌توانند با تشخیص برخی ساختارهای متداول مورد استفاده در این زبان‌ها از قبیل شرط‌ها، حلقه‌های تکرار، فراخوانی‌ها، توابع و متغیرهای محلی، کدهایی با خوانایی بسیار بالاتر نبست به کدهای اسمبلی تولید کنند. این کدها تا حدود زیادی به برنامه‌های اصلی نوشته شده نزدیک است. این کدهای تولید شده می‌توانند نقش کلیدی در تجزیه و تحلیل نحوه عملکرد نرم‌افزارها ایفا کنند.

نرم‌افزار REC (Reverse Engineering Compiler)

به جرأت می‌توان گفت این Decompiler تنها ابزار موفق برای Decompile کردن کدهای ماشین تولید شده توسط کامپایلرهای C/C++ محسوب می‌شود. این ابزار با توجه به پشتیبانی از انواع گوناگون فایل‌های اجرایی، سیستم‌های عامل و پردازنده‌ها، از عملکرد بسیار بالایی برخوردار است. کدهای تولید شده توسط این نرم‌افزار از خوانایی بسیار خوبی برخوردار بوده و از راهنماهای مفیدی نیز بهره می‌برند و می‌توانند به‌منظور ایجاد درک دقیق‌تر و بهتر از نحوه عملکرد کدهای کامپایل شده توسط این گونه کامپایلرها به کار گرفته شوند. این ابزار از فایل‌های اجرایی استاندارد ویندوز (PE) و کامپایلرهای C/C++ ارائه شده برای این سیستم عامل پشتیبانی می‌کند.

نسخه 1.6 این نرم‌افزار در CD ضمیمه موجود می‌باشد

Tools\REC

برای شروع بهتر است به یک مثال عملی از این نرم‌افزار نگاهی بیاندازیم.

C++ Code

```
int FindMax(int Ar[],int Size){
    int i,Max;
    i=0;
    Max=0;

    do{
        if (Max < Ar[i]) Max=Ar[i];
        i++;
    }while (i < Size);

    return Max;
}

void Main()
{
    int Array[10];
    int Max;
    Max=FindMax(Array,10);
    MessageBox(0,"Max Found","Test Caption",0);

    return 0;
}
```

Disassembled Code

```
L00401000: 55                   push ebp
           8b ec                mov ebp,esp
           83 ec 08             sub esp,+0x8
           c7 45 f8 00 00 00 00 mov dword [ebp-0x8],0x0
           c7 45 fc 00 00 00 00 mov dword [ebp-0x4],0x0
L00401014: 8b 45 f8             mov eax,[ebp-0x8]
           8b 4d 08             mov ecx,[ebp+0x8]
           8b 55 fc             mov edx,[ebp-0x4]
           3b 14 81             cmp edx,[ecx+eax*4]
           7d 0c                jnl 0x40102e
           8b 45 f8             mov eax,[ebp-0x8]
           8b 4d 08             mov ecx,[ebp+0x8]
           8b 14 81             mov edx,[ecx+eax*4]
           89 55 fc             mov [ebp-0x4],edx
L0040102e: 8b 45 f8             mov eax,[ebp-0x8]
           83 c0 01             add eax,+0x1
           89 45 f8             mov [ebp-0x8],eax
           8b 4d f8             mov ecx,[ebp-0x8]
           3b 4d 0c             cmp ecx,[ebp+0xc]
           7c d5                jl 0x401014
           8b 45 fc             mov eax,[ebp-0x4]
           8b e5                mov esp,ebp
           5d                   pop ebp
           c3                   ret
L00401046: 55                   push ebp
           8b ec                mov ebp,esp
           83 ec 2c             sub esp,+0x2c
           6a 0a                push +0xa
           8d 45 d8             lea eax,[ebp-0x28]
           50                   push eax
           e8 a9 ff ff ff       call 0x401000
           83 c4 08             add esp,+0x8
           89 45 d4             mov [ebp-0x2c],eax
           6a 00                push +0x0
           68 30 60 40 00       push dword 0x406030
           68 40 60 40 00       push dword 0x406040
           6a 00                push +0x0
           ff 15 9c 50 40 00    call [0x40509c]
           33 c0                xor eax,eax
           8b e5                mov esp,ebp
           5d                   pop ebp
           c2 10 00             ret 0x10
```

REC Output (without symbolic information)

```
/*  Procedure: 0x00401000 - 0x00401045
 *  Argument size: 0
 *  Local size: 8
 *  Save regs size: 0
 */
L00401000(A8, Ac)
/* unknown */ void  A8;
/* unknown */ void  Ac;
{
    /* unknown */ void  Vf8;
    /* unknown */ void  Vfc;

    Vf8 = 0;
    Vfc = 0;
    do {
        if(Vfc < *(A8 + Vf8 * 4)) {
            Vfc = *(A8 + Vf8 * 4);
        }
        Vf8 = Vf8 + 1;
    } while(Vf8 < Ac);
    return(Vfc);
}

/*  Procedure: 0x00401046 - 0x00401078
 *  Argument size: -28
 *  Local size: 44
 *  Save regs size: 0
 */

L00401046()
{
    /* unknown */ void  Vfd4;
    /* unknown */ void  Vfd8;

    Vfd4 = L00401000( & Vfd8, 10);
    *__imp__MessageBoxA(0, "Max Found", "Test Caption", 0);
    return(0);
}
```

حال که با مثال بالا به طرحی کلی از نحوه عملکرد این نرم‌افزار رسیده‌اید بهتر است نحوه استفاده از آن را مورد بررسی قرار دهیم.

این نرم‌افزار دارای واسط کاربر استاندارد برای ویندوز نیست و تنها از طریق کنسول DOS و خط فرمان، ورودی‌های خود را دریافت کرده و با کاربر ارتباط برقرار می‌کند.

ساده‌ترین روش استفاده از این نرم‌افزار به شکل زیر است:

```
> REC.exe sample.exe
```

توجه داشته باشید که sample.exe در مثال بالا می‌تواند یکی از انواع فایل‌های اجرایی حمایت شده توسط نرم‌افزار باشد که عبارتند از (AOUT , ELF, PE , COFF) در این صورت این نرم‌افزار ابتدا نواحی کد و داده فایل اجرایی موردنظر را تشخیص داده و سپس آنها را مورد تحلیل و بررسی قرار می‌دهد. اطلاعات خروجی این پردازش‌ها در فایلی با پسوند REC و همنام با فایل اجرایی موردنظر ذخیره خواهد شد.

به‌منظور مشخص کردن اطلاعات دقیق‌تر راجع به فایل اجرایی موردنظر که منجر به تحلیل دقیق‌تر و کدهای خروجی با خوانایی بیشتر می‌شود این نرم‌افزار از یک فایل واسط به نام فایل فرمان استفاده می‌کند. این فایل می‌تواند حاوی اطلاعاتی دقیق‌تری راجع به فایل ورودی باشد. فایل‌های فرمان دارای پسوند cmd هستند.

در مثال زیر یک نمونه از این فایل‌ها و اجزاء آن را مشاهده می‌کنید.

```
#!wrec

option: +hexconst
option: -doloops

types: winuser.o
types: winbase.o

file: file.exe

region: 0x80000400 0x80001600 0x400 text
region: 0x80001600 0x80002000 0x1600 data

symbol: 0x80107fe0, 0x80108077 T CrearImage()
symbol: 0x80108078, 0x801080d7 T LoadImage(char *, int, int)
symbol: 0x801080d8, 0x8010813b T StoreImage()
symbol: 0x8010813c, 0x801081ff T MoveImage(char *, int, int)

patterns: libmips.pat
```

همان‌طور که مشاهده می‌کنید فایل‌های Cmd با شناسه wrec!# شروع شده و در ادامه هر سطر حاوی یک دستور و آرگومان‌های آن است که با کاراکتر ":" از هم مجزا می‌شوند. در صورت نیاز به

درج توضیحات در فایل می‌توانید از کاراکتر # در ابتدای سطر موردنظر استفاده کنید. بدیهی است که ادامه چنین سطری مورد بررسی قرار نخواهد گرفت. حال به بررسی دستورات استاندارد قابل استفاده در این فایل‌ها خواهیم پرداخت.

Option

هر سطر از این دستور مشخص کننده یکی از تنظیمات در نظر گرفته شده برای این نرم‌افزار است. در صورت نیاز می‌توانید جزئیات آنها را در مستندات نرم‌افزار مشاهده کنید.

Types

این دستور فایل‌هایی را مشخص می‌کنند که حاوی اطلاعاتی راجع به انواع داده‌ها، ساختارها و توابع استاندارد مورد استفاده در فایل اجرایی به همراه پیش تعریف آنها است. به عنوان مثال اگر فایل اجرایی موردنظر شما از توابع API موجود در کتابخانه user32.dll استفاده می‌کند، می‌توانید فایل پیش تعریف توابع موجود در این کتابخانه را که winuser.o نام دارد به لیست این فایل‌ها در فایل cmd مربوطه اضافه کنید. در جدول زیر لیستی از فایل‌های پیش تعریف استاندارد را که همراه با این نرم‌افزار عرضه شده‌اند مشاهده می‌کنید. در ستون Related File می‌توانید فایل هدر مربوطه را که توسط کامپایلرهای C/C++ عرضه می‌شود مشاهده کنید. برای توابع API ویندوز، فایل کتابخانه‌ای مربوطه نیز ذکر شده است.

Type File	Related File	Description
FCNTL.O	FCNTL.H	This file defines constants for the file control options used by the _open() function.
STDIO.O	STDIO.H	This file defines the structures, values, macros, and functions used by the level 2 I/O ("standard I/O") routines.
STDLIB.O	STDLIB.H	This include file contains the function declarations for commonly used library functions which either don't fit somewhere else, or, cannot be declared in the normal place for other reasons.
STRING.O	STRING.H	This file contains the function declarations for the string manipulation functions.
mmsystem.o	MMSYSTEM.H (mmsystem.dll)	Include file for Multimedia API's.
shellapi.o	SHELLAPI.H (shell32.dll)	SHELL32.DLL functions, types, and definitions.

Type File	Related File	Description
winbase.o	WINBASE.H (kernel32.dll)	This module defines the 32-Bit Windows Base APIs.
wingdi.o	WINGDI.H (gdi32.dll)	GDI procedure declarations, constant definitions and macros.
winreg.o	WINREG.H (advapi32.dll)	This module contains the function prototypes and constant, type and structure definitions for the Windows 32-Bit Registry API.
winuser.o	WINUSER.H (user32.dll)	USER procedure declarations, constant definitions and macros

در صورت نیاز می‌توانید فایل‌های پیش تعریف دلخواه خود را ایجاد و از آنها استفاده کنید. به‌منظور دریافت اطلاعات دقیق‌تر راجع به نحوه ساخت این فایل‌ها می‌توانید به مستندات این نرم‌افزار رجوع کنید.

File

این دستور فایل اجرایی موردنظر را برای عملیات Decompile مشخص می‌کند. این فایل می‌تواند یکی از انواع فایل‌های اجرایی استانداردی باشد که قبلاً ذکر شده است.

Region

پارامترهای مورد استفاده در این دستورات نواحی مختلف فایل اجرایی را از نظر نوع، آدرس شروع و پایان در هنگام بارگذاری و آفست ناحیه موردنظر در فایل مشخص می‌کنند. توجه داشته باشید که نوع درنظر گرفته شده برای یک ناحیه برروی نحوه تحلیل و بررسی آن تأثیر مستقیم می‌گذارد. نواحی text به‌منظور عملیات Decompile و نواحی data به‌منظور بررسی رشته‌ها و اشاره‌گرهای استفاده شده در کد مورد بررسی قرار خواهند گرفت. توجه داشته باشید که در صورت نیاز می‌توانید با معرفی کردن نواحی دلخواه خود از Decompile شدن سایر نواحی در فایل خروجی جلوگیری کنید. این امر باعث ایجاد سرعت بیشتر در عملیات Decompile شدن و خوانایی بهتر فایل خروجی می‌گردد. در زیر نحوه استفاده از این دستور را به همراه توضیحات کوتاهی در مورد عملکرد هر پارامتر مشاهده می‌کنید.

	Start Address	End Address	File Offset	Region Type
region:	0x8001000	0x80109b4	0x8FF	text

Symbol

این دستورات و پارامترهای آنها آدرس شروع، پایان، پارامترها و نام دلخواهی را برای توابع موجود در فایل اجرایی تعیین می‌کنند. این نام‌ها می‌توانند باعث خوانایی بهتر فایل خروجی شده و نتیجه را هر چه بیشتر به کد اصلی نزدیک کنند. در پارامترهای این دستور، مشخص کردن آدرس پایان تابع ضرورتی ندارد زیرا در مراحل بعد به‌طور خودکار توسط REC شناسایی شده و به کار گرفته می‌شود.

Patterns

فایل‌های Pat مورد استفاده در این دستورات الگوهای خاصی را برای جایگزینی یک سری داده‌های باینری خاص با یک نام سمبولیک تعیین می‌کنند. این نام‌ها در مراحل بعد توسط REC به کار گرفته شده و باعث ایجاد خوانایی بهتر فایل خروجی می‌شوند. در مثال زیر ساختار این فایل‌های الگو را مشاهده می‌کنید.

```
MyFunction() size: 16
A0 00 0A 24 08 00 40 01
00 00 09 24 00 00 00 00
;
MyData size: 14
B5 A7 0A 24 08 2D 01 00 09 24 00 AA 00 00
;
```

همان‌طور که متوجه شده‌اید هر چه اطلاعات اولیه بیشتری در مورد فایل اجرایی در اختیار این ابزار قرار گیرد، فایل اجرایی نهایی شباهت بیشتری به برنامه اولیه داشته و از خوانایی بهتری نیز برخوردار خواهد بود. در مثال زیر یک نمونه از خروجی این نرم‌افزار را با استفاده از اطلاعات کافی در فایل Cmd مربوطه مشاهده می‌کنید.

```
hexdump(char * fname)
{
    unsigned char  buff[16];
    unsigned long  offset;
    struct _IO_FILE* fp;
    struct stat st;
    int cnt;

    if(stat(fname, & st) != 0) {
        fp = fopen(fname, "rb");
        if(fp != 0) {
            offset = 0;
L08048867:
            if(st.st_size > offset) {
                cnt = fread( & buff, 1, 16, fp);
                if(cnt != 0) {
                    dumpline( & buff, offset, cnt);
                    offset = offset + cnt;
                    goto L08048867;
                }
            } else {
            }
            fclose(fp);
            eax = 0;
        } else {
            perror(fname);
            eax = 1;
        }
    } else {
        perror(fname);
        eax = 1;
    }
}
```

JAVA Decompilers

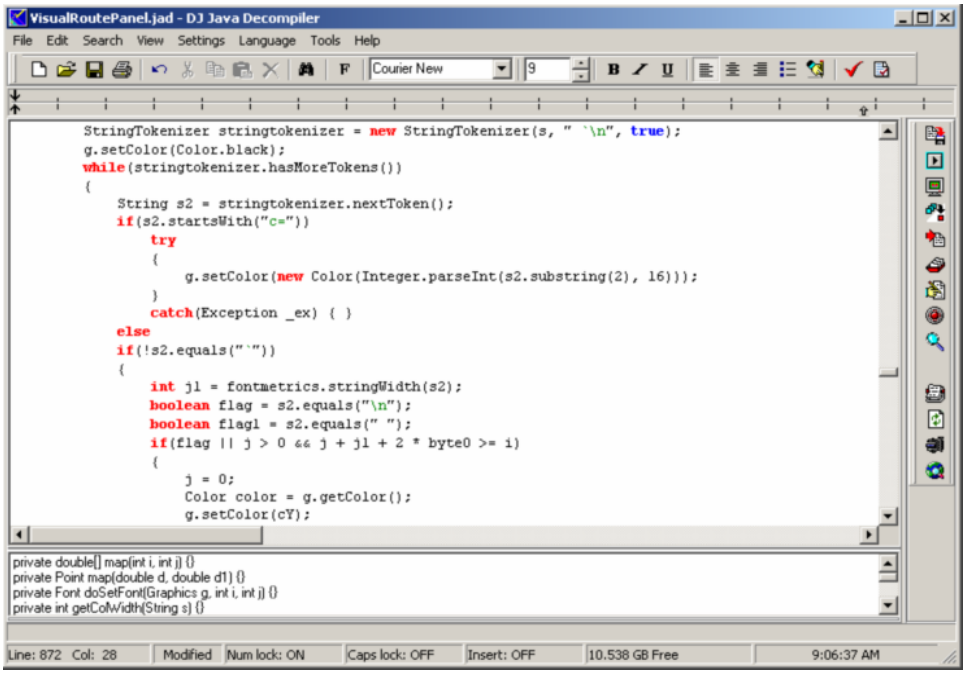
نرم‌افزارهای نوشته شده به زبان JAVA به‌طور گسترده‌ای به‌صورت appletها در اینترنت و یا سایر نرم‌افزارهای کاربردی دیگر به کار گرفته می‌شوند. کامپایلرهای این زبان برنامه‌های نوشته شده را به زبان ماشین ترجمه نمی‌کنند بلکه این برنامه‌ها به یک زبان میانی به نام Byte-Codeها ترجمه شده و معمولاً با پسوند Class ذخیره می‌شوند. وظیفه ترجمه این Byte-Code به زبان ماشین مربوطه و اجرای آنها برعهده ماشین مجازی JAVA (JAVA Virtual Machine) است. به کارگیری این روش سبب می‌شود که این برنامه‌ها بتوانند مستقل از سخت‌افزار اجرا شوند. با توجه به مراحل کامپایل بسیار ساده و نیز استفاده از ساختارهای ثابت و استاندارد توسط کامپایلرهای JAVA که به‌منظور کامپایل کردن برنامه‌های نوشته شده به Byte-Codeها صورت می‌گیرد، Decompiler های این زبان نیز به سادگی طراحی و پیاده‌سازی شده و از موفقیت بسیار بالایی نیز برخوردار هستند. موفقیت این Decompilerها تا حدی است که نتایج خروجی آنها می‌تواند مستقیماً به‌منظور کامپایل مجدد توسط کامپایلرهای JAVA به کار گرفته شود. در ادامه نحوه عملکرد یکی از موفق‌ترین Decompilerهای JAVA را مورد بررسی قرار خواهیم داد.

نرم‌افزار DJ JAVA Decompiler (JAD)

این ابزار یکی از معروفترین Decompilerهای طراحی شده برای JAVA است. نرم‌افزار DJ JAVA Decompiler در حقیقت به عنوان پوسته‌ای برای برای ابزار JAD عمل کرده و رابط کاربر استانداردی را برای استفاده بهتر از آن فراهم می‌آورد. علت ایجاد این ابزار این است که JAD در کنسول DOS اجرا شده و فاقد رابط کاربر استاندارد ویندوز است. استفاده از این ابزارها بسیار ساده بوده و همان‌طور که ذکر شد فایل خروجی آنها می‌تواند بدون نیاز به تصحیح به‌منظور کامپال مجدد توسط کامپایلرهای JAVA به کارگرفته شود. در شکل (٤-١) صفحه اصلی نرم‌افزار DJ JAVA Decompiler را در حال نمایش کدهای Decompile شده مشاهده می‌کنید.

نسخه 3.7 این نرم‌افزار در CD ضمیمه موجود می‌باشد

Tools\ DJJavaDecompiler



شکل (٤-١)

Visual Basic Decompilers

Visual Basic تا قبل از نسخه‌های ٤ برنامه‌های خروجی خود را به یک زبان واسط به نام P-Code تبدیل می‌کرد. این کدها توسط یک فایل dll که در حقیقت نقش ماشین مجازی VB را بازی می‌کرد به زبان ماشین تبدیل شده و اجرا می‌شدند. در نسخه‌های جدیدتر امکان کامپایل شدن برنامه‌های نوشته شده به زبان ماشین وجود دارد با این وجود این فایل‌های اجرایی کامپایل شده نیز به شدت به فایل dll مذکور وابسته هستند در حقیقت این فایل‌های اجرایی از قسمت اندکی کد ماشین به‌علاوه فراخوانی‌های متعدد از فایل dll مذکور تشکیل می‌شوند که همین امر باعث کند شدن سرعت اجرایی آنها می‌شود.

با توجه به عدم استفاده این کامپایلر از ساختارها و دستورالعمل‌های استاندارد ماشین در فایل اجرایی، Decompilerهای آن نیز بسیار معدود بوده و از موفقیت بسیار محدودی برخوردار هستند و عملکرد اکثر آنها بیشتر به تشخیص و بازیابی رابط کابر استفاده شده در فایل‌ها اجرایی محدود می‌شود. با این وجود این Decompilerها می‌توانند اطلاعات مفیدی را راجع به آدرس‌های شروع توابع، نام و مشخصات آنها و ساختار سلسله مراتبی اشیاء استفاده شده در فایل‌های اجرایی

مشخص کنند. این عمل گامی بسیار مهم در شناسایی ساختار کلی فایل اجرایی موردنظر و عملکرد آن محسوب می‌شود.

در ادامه این بحث به بررسی یکی از موفق‌ترین Decompilers های VB خواهیم پرداخت.

نرم‌افزار VB Reformer

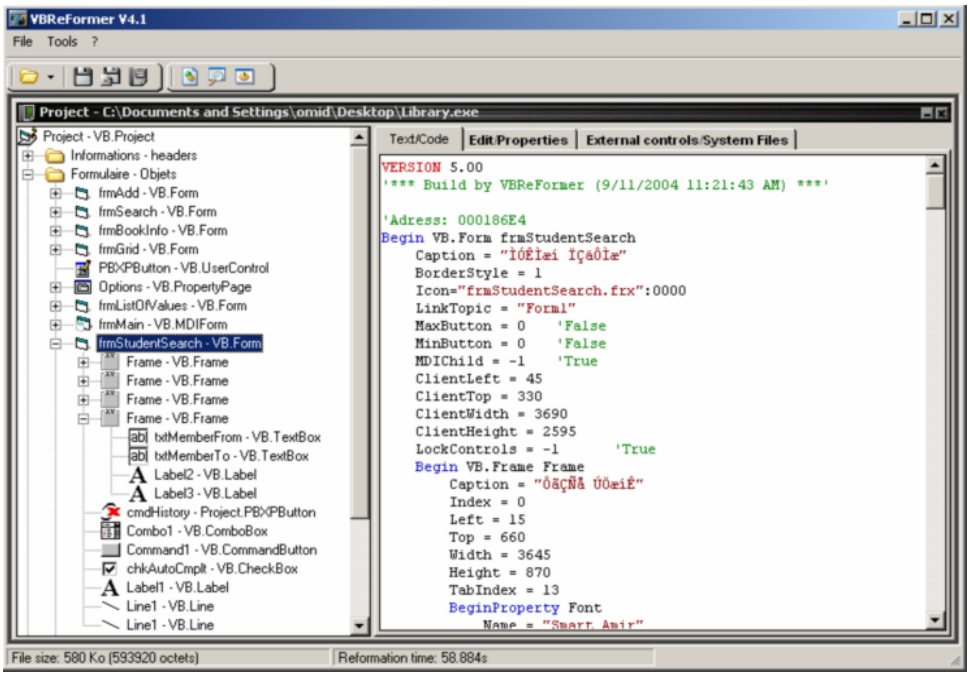
این ابزار می‌تواند به‌منظور بررسی رابط کاربر، توابع و اشیاء استفاده در فایل‌های اجرایی کامپایل شده به وسیله Visual Basic به کارگرفته شود. می‌توان گفت عملکرد اصلی آن بازیابی یک پوسته خالی از فایل اجرایی موردنظر است. این پوسته خالی شامل فایل‌های مربوط به فرم‌ها، ماژول‌ها و کلاس‌ها است که از بین آنها تنها فایل‌های مربوط به رابط کاربر به‌طور کامل بازیابی می‌شوند و برای بقیه آنها تنها فایل‌های بلا استفاده‌ای به وجود آورده می‌شود که صرفاً جنبه نمادین دارند.

نسخه 4.1 این نرم‌افزار در CD ضمیمه موجود می‌باشد

Tools\VBReformer

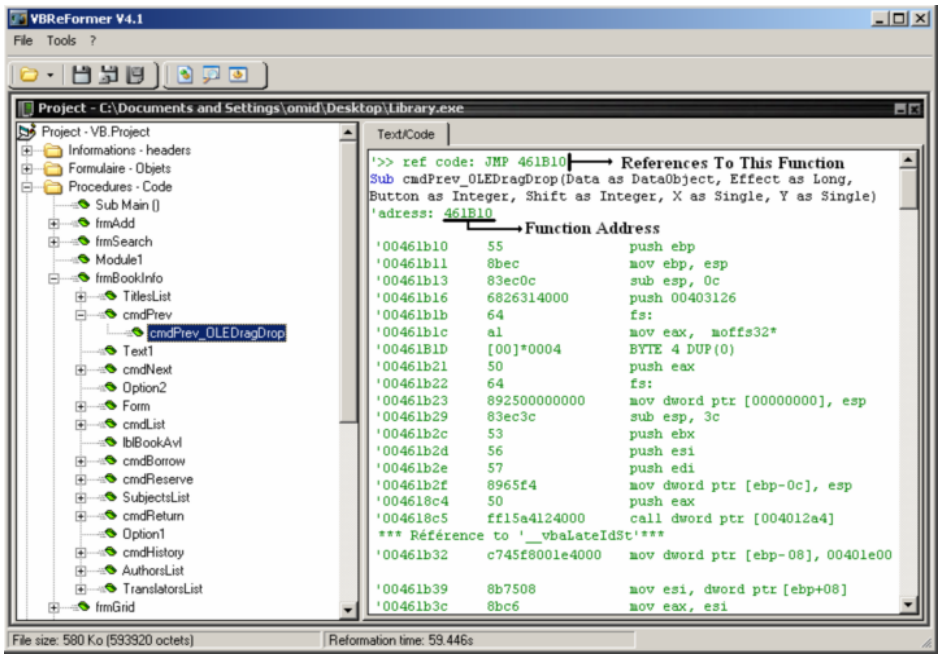
نحوه کار با این ابزار بسیار ساده است که در ادامه با چند مثال به آن خواهیم پرداخت.

در شکل (٤-۲) صفحه اصلی این نرم‌افزار را در حال نمایش سلسله مراتبی فرم‌ها و اشیاء بصری موجود در آنها مشاهده می‌کنید.



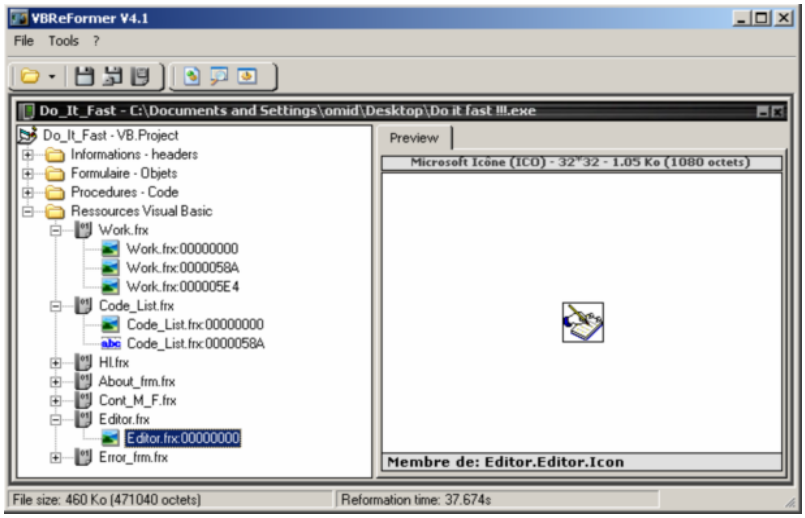
شکل (٤-٢)

به‌منظور دریافت اطلاعات در مورد توابع موجود در برنامه و محل هر یک می‌توانید از قسمت Procedures-code از نمودار درختی استفاده کنید. همان‌طور که در شکل (٤-٣) مشاهده می‌کنید، با انتخاب هر تابع، اطلاعات مفیدی راجع به آدرس ارجاع‌ها، آدرس شروع و کدهای اسمبلی مربوطه در قسمت راست پنجره نمایش داده می‌شود.



شکل (٤-٣)

درصورت نیاز می‌توانید همانند مثال زیر با استفاده از قسمت Ressources Visual Basic به بررسی منابع استاندارد VB در فایل اجرایی موردنظر بپردازید.



شکل (٤-٤)

C++ Builder / Delphi Decompilers

کامپایلرهای C++ Builder و Delphi ساخت شرکت Borland هستند و فایل‌های اجرایی آنها ساختار مشابهی دارند. برنامه‌های نوشته شده به این زبان‌ها در هنگام کامپایل به‌طور کامل به کد ماشین تبدیل می‌شوند ولی وجود ساختارهای ثابت و از پیش تعریف شده در آنها که به‌منظور درج کلاس‌ها و اشیاء، درنظر گرفته شده‌اند، سبب شده است که Decompiler های آنها از موفقیت نسبتاً خوبی برخوردار بوده و داده‌های خروجی آنها خوانایی قابل قبولی داشته باشند. توجه به این نکته جالب است که نام‌ها و تعاریف مورد استفاده برای کلاس‌ها و اشیاء که در هنگام طراحی نرم‌افزارها در این زبان‌ها توسط سازنده تعیین شده‌اند، در فایل کامپایل شده نهایی نیز به‌همان صورت وجود دارند. این نام‌ها به‌نوبه خود سبب خوانایی بیشتر کدهای Decompile شده می‌گردند. این کامپایلرها نیز مانند VB به‌صورت پیش‌فرض از ساختارهای خاص خود برای ذخیره کردن و مدیریت منابع و اشیاء مربوط به رابط کاربر برنامه استفاده می‌کنند. این داده‌ها در قسمتی به نام RC Data از منابع فایل اجرایی ذخیره می‌شوند.

حال که با برخی از خصوصیات فایل‌های اجرایی این کامپایلرها آشنایی پیدا کردید، به سراغ یکی از قوی‌ترین Decompilerهای موجود برای این کامپایلرها خواهیم رفت و نحوه عملکرد آن را به‌طور کامل مورد بررسی قرار خواهیم داد.

نرم‌افزار DeDe (Delphi Decompiler)

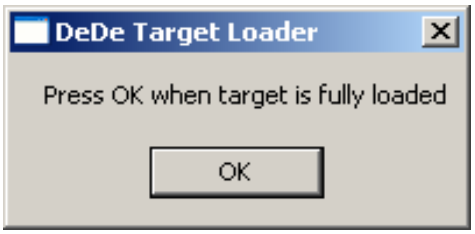
این نرم‌افزار یکی ازکامل‌ترین Decompilerهای موجود برای کامپایلرهایDelphi , C++ Builder ساخت شرکت Borland است و می‌تواند به‌منظور Decompile کردن فایل‌های کامپایل شده به وسیله نسخه‌های 7~ 2 این کامپایلرها به کار گرفته شود.

نسخه 3.50.03 این نرم‌افزار در CD ضمیمه موجود می‌باشد

Tools\DeDe

حال نحوه کار با این ابزار را مورد بررسی قرار می‌دهیم.

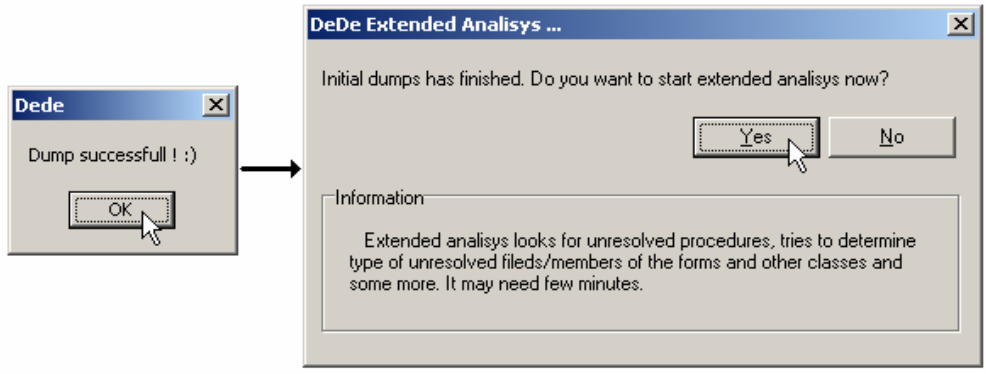
برای شروع، فایل اجرایی موردنظر را باز کرده و دکمه Process را فشار دهید. با انجام این کار فایل اجرایی مذکور توسط DeDe اجرا می‌شود و پیغامی مشابه پیغام زیر نمایش داده شده و از شما می‌خواهد که تا جرای کامل فایل اجرایی صبر کنید و سپس دکمه OK را فشار دهید. با این عمل DeDe فایل اجرایی موردنظر را از حافظه بازخوانی کرده (Dump) و سپس آن را مورد تحلیل و بررسی قرار می‌دهد.



شکل (٤-٥)

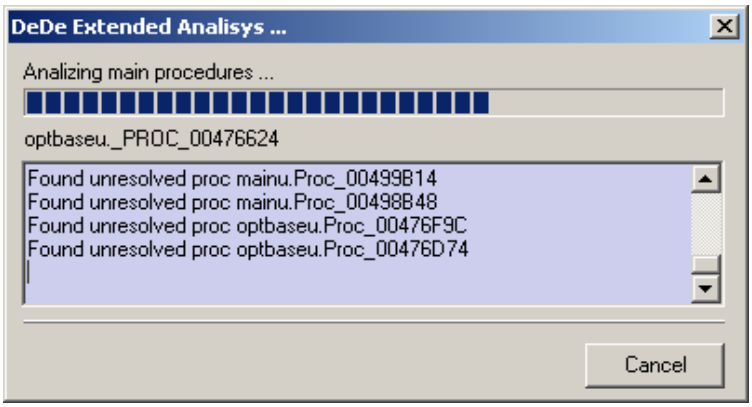
به علت استفاده برخی از فایل‌های اجرایی از متدهای محافظتی، DeDe ترجیح می‌دهد که به جای استفاده از فایل اجرایی موردنظر نسخه‌ای از آن را که به حافظه بارگذاری می‌شود، مورد استفاده قرار دهد زیرا تمام فایل‌های اجرایی محافظت شده در هنگام بارگذاری به حافظه و اجرا مجبورند که از حالت کد شده خارج شوند.

در صورت موفقیت‌آمیز بودن مرحله بازخوانی پیغامی مشابه شکل (٤-٦) نمایش داده شده و موفقیت این عملیات را اعلام می‌کند. با تایید این پیغام، پنجره Extended Analyses مطابق شکل نمایش داده می‌شود.



شکل (٤-٦)

در بخش Extended Analyses بررسی‌های دقیق‌تری به‌منظور کشف و نام گذاری برخی از ساختارها، متدها و کلاس‌هایی که در مرحله قبل ناشناخته مانده‌اند صورت می‌گیرد. با تایید بررسی Extended Analyses پنجره‌ای مانند شکل (٤-٧) باز شده و روند جستجو و شناسایی را نشان خواهند داد.

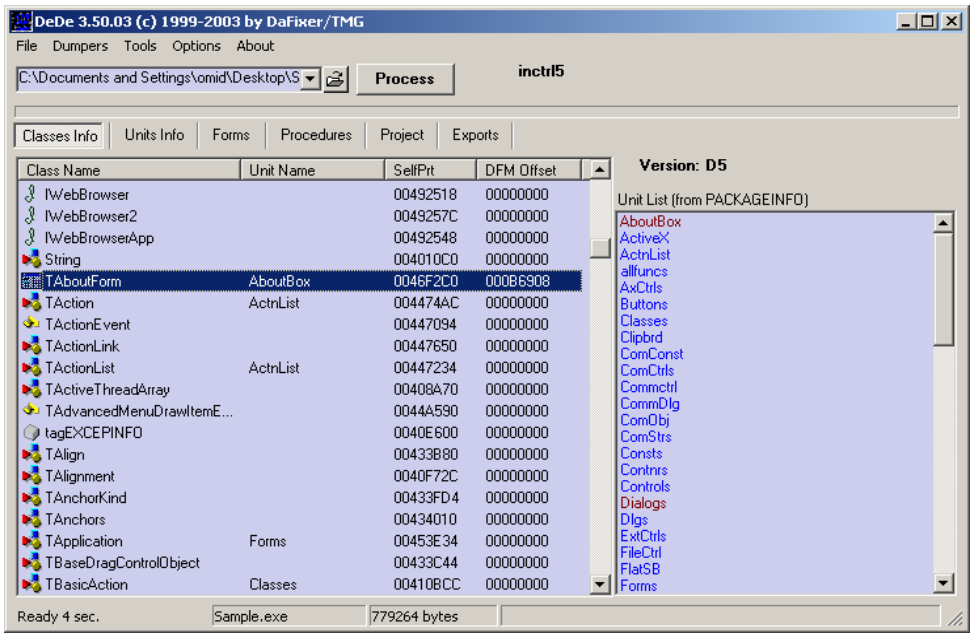


شکل (٤-٧)

پس از اتمام عملیات Extended Analyses پنجره اصلی DeDe نمایش داده شده و نتیجه تحلیل و بررسی‌های انجام شده را به نمایش می‌گذارد. به‌منظور خوانایی بهتر، اطلاعات مذکور برحسب موضوع به بخش‌های مختلفی تقسیم شده‌اند که در ادامه به بررسی آنها خواهیم پرداخت.

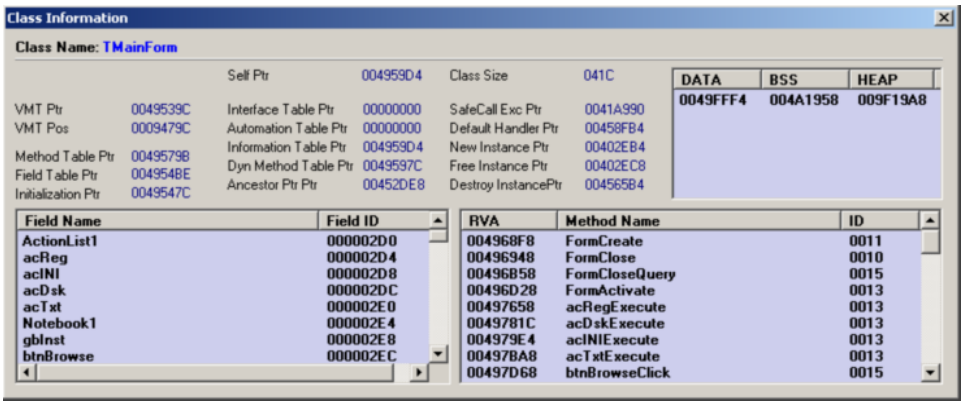
Classes Info

در این قسمت لیستی از کلیه کلاس‌ها و اشیاء مورد استفاده در فایل اجرایی به همراه اطلاعات کاملی راجع به هر یک به نمایش گذاشته می‌شود. همان‌طور که در شکل (٤-٨) مشاهده می‌کنید، برخی از این کلاس‌ها، کلاس‌های استاندارد کامپایلرهای C++Bulder/Delphi هستند که به‌طور پیش‌فرض در تمام فایل‌های اجرایی ساخته شده توسط این کامپایلرها قرار داده می‌شوند. به همین دلیل است که فایل‌های اجرایی ساخته شده توسط کامپایلرها از حجم بالایی برخوردار هستند.



شکل (٤-٨)

در صورت نیاز می‌توانید به‌منظور دریافت اطلاعات دقیق‌تر راجع به کلاس موردنظر، برروی نام آن در لیست کلاس‌ها Double click کنید. با این عمل پنجره Class Information همانند شکل (٤-٩) ظاهر شده و لیستی از اعضاء و متدهای کلاس مذکور را به همراه اطلاعات دقیقی راجع به شناسه‌ها و آدرس‌های شروع هر یک در فایل اجرایی به نمایش می‌گذارد.

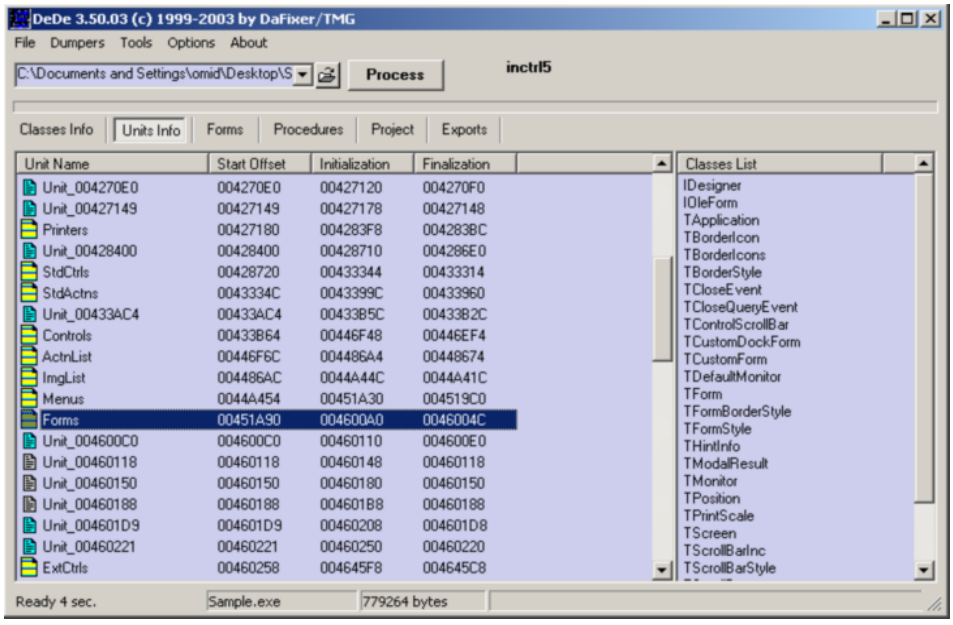


شکل (٤-٩)

همان‌طور که در شکل بالا مشاهده می‌کنید در این پنجره اطلاعات بسیار مفیدی نیز درباره توابع و ساختارهای مربوط به مدیریت کلاس و آدرس شروع هر یک نیز وجود دارد که از آن جمله می‌توان به تابع سازنده و مخرب کلاس، تابع مدیریت خطاها و رکوردهای مربوط به متدها و خواص کلاس اشاره کرد.

Units Info

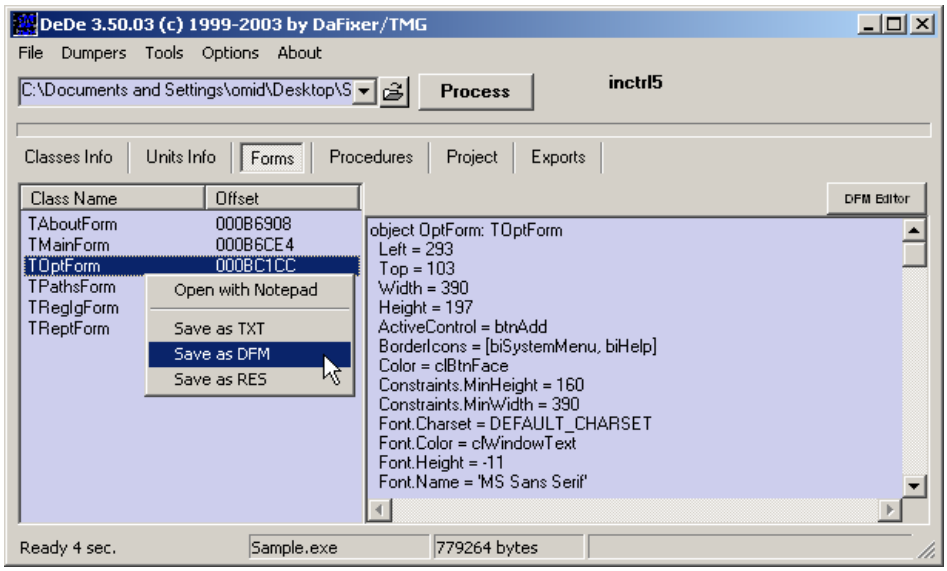
همان‌طور که می‌دانید فایل‌های حاوی کد در کامپایلرهای C++ Builder/Delphi , unit نامیده می‌شوند. این فایل‌ها می‌توانند حاوی تعاریف کلاس‌ها، اشیاء، توابع، ساختارها و... باشند. در قسمت Units Info لیستی از کلیه Unitهای استفاده شده برای تولید فایل اجرایی موردنظر به‌همراه لیستی از کلاس‌های تعریف شده در هر یک به نمایش گذاشته شده است. همان‌طور که در شکل (٤-١٠) مشاهده می‌کنید به علت عدم شناسایی، برای برخی از Unitها از نام‌های مستعار مانند Unit_00480100 استفاده شده است که معمولاً قسمت دوم از این نام‌ها آدرس شروع unit موردنظر در فایل اجرایی هستند.



شکل (٤-١٠)

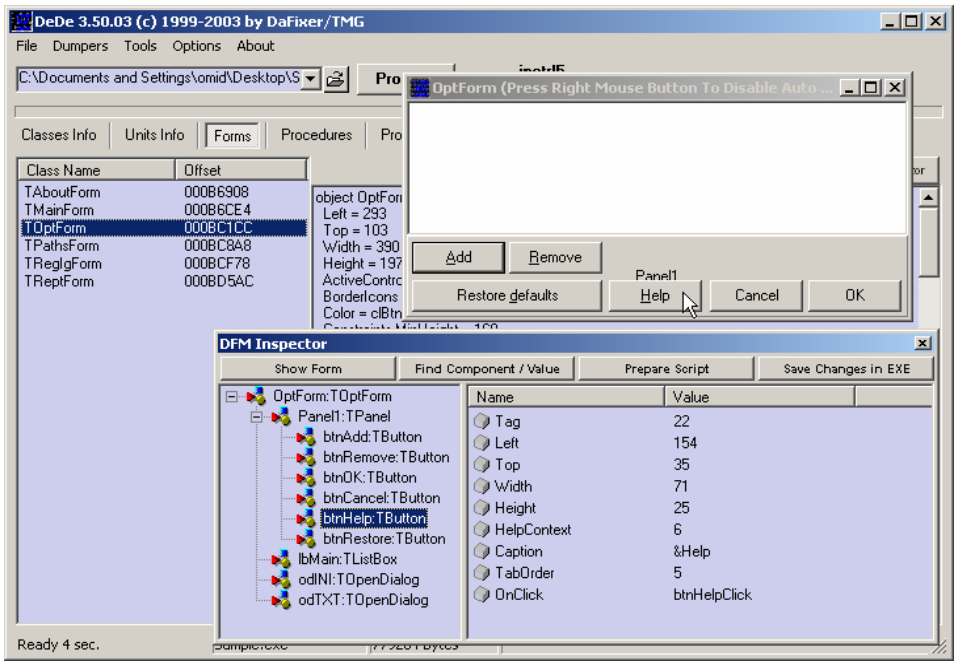
Forms

در این قسمت لیستی از کلیه پنجره‌های استاندارد استفاده شده در فایل اجرایی موردنظر به نمایش گذاشته می‌شود. در صورت نیاز می‌توانید آنها را به‌صورت فایل‌های استاندارد کامپایلرهای مذکور ذخیره کرده و از آنها استفاده کنید.



شکل (٤-١١)

به‌منظور مشاهده شکل ظاهری فرم موردنظر می‌توانید برروی آن Double Click کنید. با این عمل پنجره DFM Inspector به همراه پنجره موردنظر به نمایش گذاشته می‌شود.



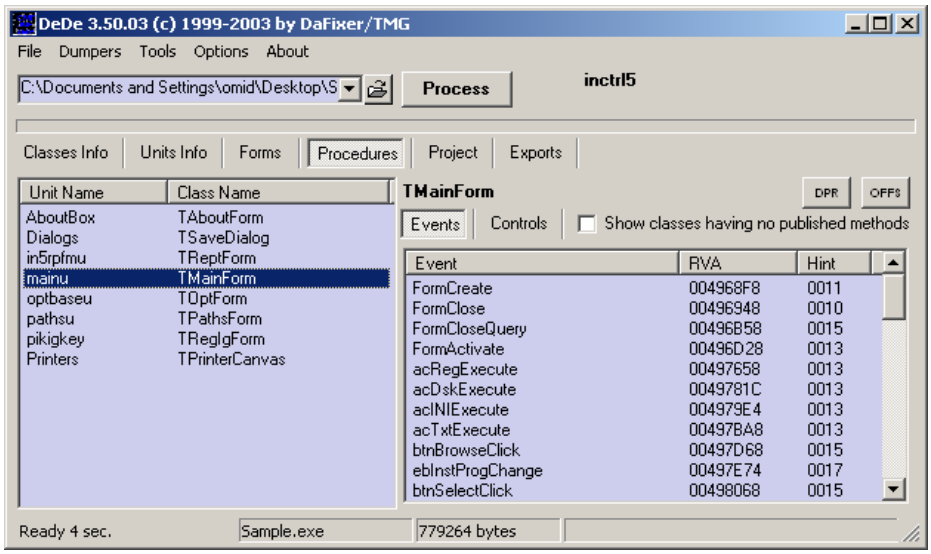
شکل (٤-١٢)

همان‌طور که مشاهده می‌کنید، با حرکت برروی کنترل‌های استفاده شده در پنجره موردنظر، رکورد متناظر با آن نیز در ساختار سلسله مراتبی نشان داده شده در پنجره DFM Inspector نمایش داده می‌شود.

متأسفانه نسخه‌های فعلی DeDe امکان ایجاد تغییرات در فایل‌های اجرایی را ندارند و دکمه‌های موجود در پنجرهٔ DFM Inspector صرفاً جنبه نمایشی داشته و غیرفعال هستند.

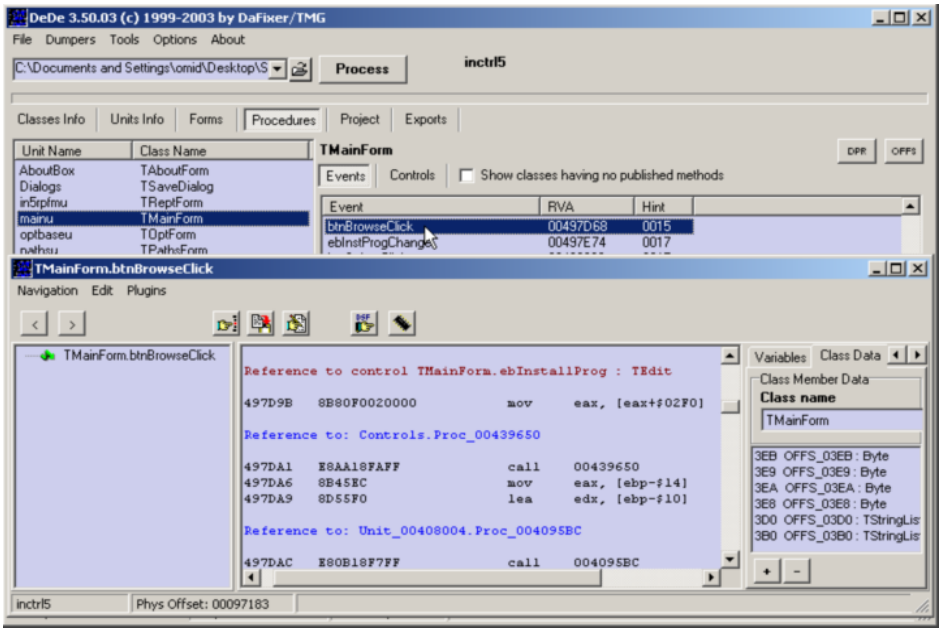
Procedures

در این قسمت اطلاعات بسیار دقیقی راجع به توابع موجود در فایل اجرایی به نمایش گذاشته می‌شود. همان‌طور که در شکل (٤-١٣) مشاهده می‌کنید توابع بر حسب Unit تقسیم‌بندی شده‌اند. در صورتی که یک تابع مسئول پاسخگویی به رویداد خاصی باشد، نام آن رویداد نیز ذکر شده است. در مورد سایر توابع از نام مستعار استفاده شده است.



شکل (٤-١٣)

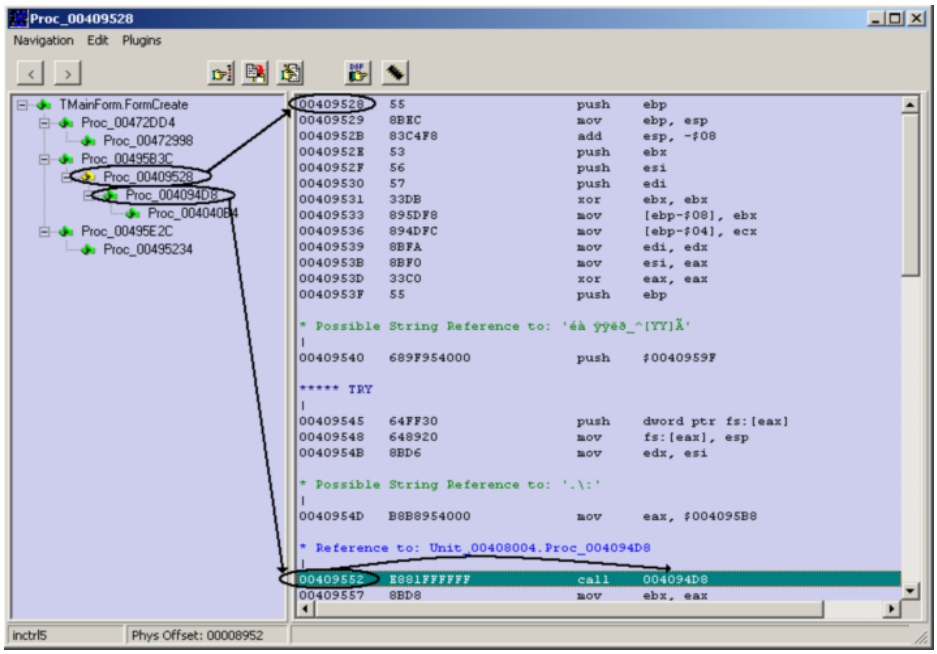
به‌منظور مشاهده کدهای مربوط به یک تابع، برروی نام آن در لیست Events، Double Click کنید. با این عمل پنجره Disassembler همانند شکل (٤-١٤) نشان داده می‌شود.



شکل (٤-١٤)

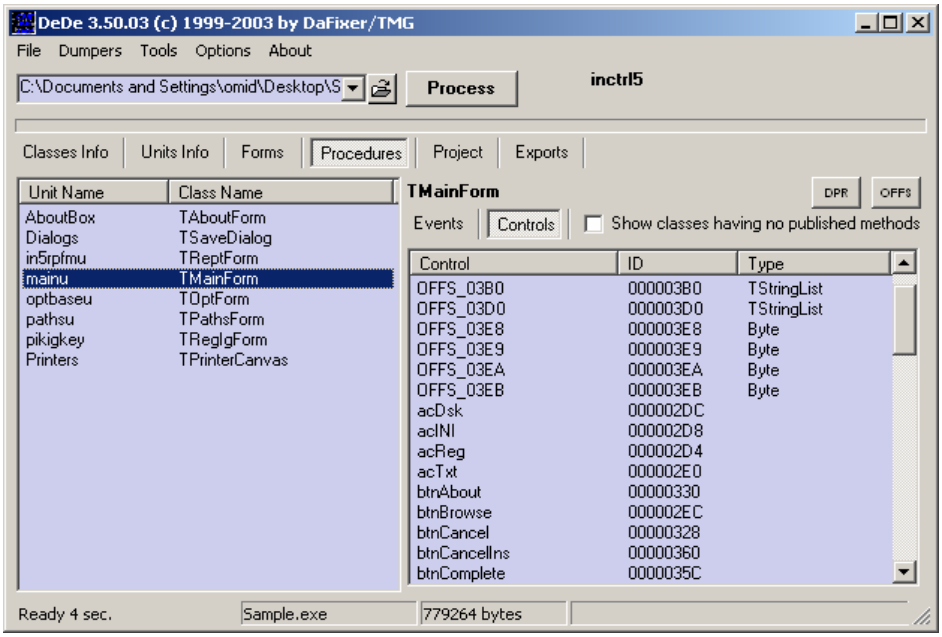
در پنجره Disassembler کدهای اسمبلی مربوط به تابع موردنظر از ابتدا تا انتها نمایش داده می‌شود. همان‌طور که در شکل بالا مشاهده می‌کنید در مورد فراخوانی‌ها و ارجاع‌ها به کلاس‌ها و Unitهای موجود در فایل اجرایی، اطلاعات مفیدی راجع به نام کلاس، نوع کلاس، نام متد و یا خاصیت مراجعه شده آورده شده است. این اطلاعات می‌تواند شما را در تجزیه و تحلیل کدهای اسمبلی و بررسی این اجاع‌ها و فراخوانی‌ها یاری دهد.

به‌منظور دنبال‌کردن پرش‌ها و ارجاع‌های انجام شده درکد اسمبلی می‌توانید برروی دستور موردنظر Double Click کنید. در صورتی که دستورالعمل مذکور یک دستور پرش قطعی باشد، سطر متناظر با آدرس پرش در لیست دستورات انتخاب می‌شود. در صورتی که دستورالعمل موردنظر، یک دستورالعمل فراخوانی تابع (call) باشد، DeDe تابع مذکور را Disassemble کرده و آن را نمایش خواهد داد. همان‌طور که در شکل (٤-١٥) می‌بینید، با دنبال کردن این فراخوانی‌ها لیستی سلسله مراتبی از آنها در سمت چپ پنجره Disassembler نمایش داده می‌شود. این لیست سلسله مراتبی می‌تواند از سردرگمی شما در هنگام دنبال کردن فراخوانی‌های انجام شده به‌صورت تو در تو جلوگیری کند، با کلیک برروی آیتم‌های این لیست کدهای اسمبلی مربوط به آن تابع در صفحه Disassembler نمایش داده خواهد شد.



شکل (٤-١٥)

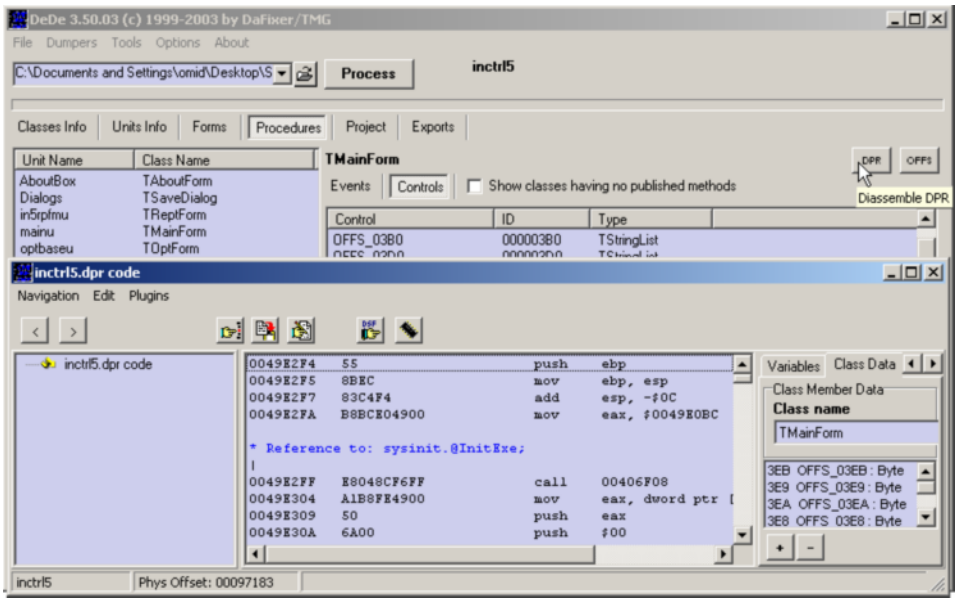
در صورت نیاز برای مشاهده کنترل‌های استفاده شده در هر unit می‌توانید همانند شکل (٤-١٦) گزینه Controls را به جای Events انتخاب کنید.



شکل (٤-١٦)

درصورت نیاز می‌توانید کدهای آغازین فایل اجرایی را که محل مقداردهی اولیه متغیرها، اشاره‌گرها، ایجاد کلاس‌ها و.... هستند مشاهده کنید. روند اجرایی در فایل‌های اجرایی ساخته شده به‌وسیله کامپایلرهای C++Builder/Delphi معمولاً به‌وسیله یک پروسیجر اصلی، شروع و خاتمه پیدا می‌کند. معمولاً کدهای مربوط به این تابع در فایل پروژه این کامپایلرها ذخیره می‌شود. در کامپایلر Delphi پسوند آنها dpr است. همان‌طورکه ذکر شد وظیفه‌اصلی این تابع آماده‌سازی پیش‌زمینه‌های اولیه لازم برای اجرای فایل موردنظر است. به‌طور معمول روند اجرایی فایل‌های ساخته شده به‌وسیله این کامپایلرها از آدرس این تابع آغاز می‌شود.

به‌منظور مشاهده کدهای مربوط به این تابع می‌توانید برروی دکمه DPR در صفحه procedures کلیک کنید. با این عمل پنجره Disassembler که همانند شکل (٤-١٧) باز شده و کدهای اسمبلی مربوط به این تابع را نمایش خواهد داد.


DeDe 3.50.03 (c) 1999-2003 by DaFixer/TMG
File Dumpers Tools Options About
C:\Documents and Settings\omid\Desktop\S
Process
inctrl5
Classes Info Units Info Forms Procedures Project Exports
Unit Name Class Name
AboutBox TAboutForm
Dialogs TSaveDialog
in5rptmu TReptForm
mainu TMainForm
optbaseu TOptForm
TMainForm
Events Controls Show classes having no published methods
DPR OFFS
Diassemble DPR
Control ID Type
OFFS_03B0 000003B0 TStringList
inctrl5.dpr code
Navigation Edit Plugins
inctrl5.dpr code
0049E2F4 55 push ebp
0049E2F5 8BEC mov ebp, esp
0049E2F7 83C4F4 add esp, -$0C
0049E2FA B8BCE04900 mov eax, $0049E0BC
* Reference to: sysinit.@InitExe;
0049E2FF E8048CF6FF call 00406F08
0049E304 A1B8FE4900 mov eax, dword ptr
0049E309 50 push eax
0049E30A 6A00 push $00
Variables Class Data
Class Member Data
Class name
TMainForm
3EB OFFS_03EB : Byte
3E9 OFFS_03E9 : Byte
3EA OFFS_03EA : Byte
3E8 OFFS_03E8 : Byte
inctrl5
Phys Offset: 00097183

شکل (٤-١٧)

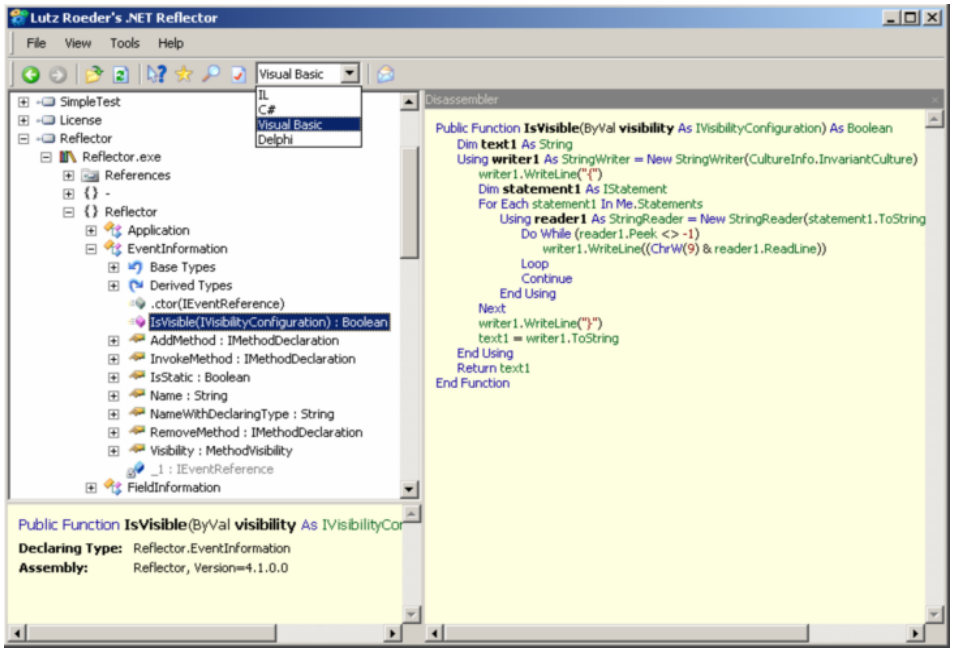
.NET Decompilers

همان‌طور که ذکر شد فایل‌های اجرایی تولید شده توسط کامپایلرهای Net. به یک زبان واسط (IL) ترجمه شده و توسط یک ماشین مجازی اجرا می‌شوند. از لحاظ نحوه عملکرد و ساختار کلی این کامپایلرها همانند کامپایلرهای JAVA هستند. Decompilerهای متعددی وجود دارند که می‌توانند به‌منظور ایجاد کدهای اولیه از روی کدهای IL به کار گرفته شوند. همان‌طور که در ادامه خواهید دید، کدهای Decompile شده توسط این ابزارها از خوانایی بسیار خوبی برخوردار بوده و می‌توانند به سادگی و بدون نیاز به تغییر برای کامپایل مجدد به کار گرفته شوند.

در شکل (۱۸-٤) صفحه اصلی نرم‌افزار Reflector را درحال نمایش کدهای Decompile شده فایل اجرایی خود مشاهده می‌کنید.

نسخه 4.1 این نرم‌افزار در CD ضمیمه موجود می‌باشد

Tools\Reflector



شکل (۱۸-٤)

فصل پنجم

ها Debugger

فصل پنجم

Debugger ها

مقدمه

در مراحل ایجاد نرم‌افزارها با زبان‌های سطح بالا، به‌طور قطع از امکانات Debuggerهای تدارک دیده شده برای این زبان‌ها به‌منظور کشف و رفع خطاهای احتمالی برنامه‌های خود استفاده کرده‌اید. همان‌طور که می‌دانید از این ابزارها به‌منظور کنترل روند اجرایی برنامه و ردیابی خطاها و... استفاده می‌شود. معمولاً این Debuggerها به‌منظور کنترل کدهای سطح بالا همانند کدهای C++ , Pascal و... به کار گرفته می‌شوند.

انواع دیگری از Debuggerها نیز وجود دارند که امکانات آنها برروی کنترل، تحلیل و بررسی کدهای کامپایل شده و فایل‌های اجرایی نهایی متمرکز شده است. از این ابزارها به‌منظور انجام بررسی‌های دقیق برروی نحوه عملکرد و جزئیات مربوط به روند اجرایی فایل‌های اجرایی استفاده می‌شود. به‌طورکلی این Debuggerها به دو دسته سیستمی و کاربر تقسیم می‌شوند. از Debugger های سیستمی معمولاً به‌منظور انجام بررسی‌ها برروی کدهای سیستمی و درایورهای سخت‌افزاری استفاده می‌شود. Soft Ice یکی از معروف‌ترین این Debuggerها محسوب می‌شود.

دسته دیگری از Debuggerها به‌منظور انجام بررسی‌ها برروی روند اجرایی نرم‌افزارهای مد کاربر به کار گرفته می‌شوند. با توجه به استفاده‌های گسترده از این Debuggerها و نیز سهولت نحوه کار با آنها، در این فصل یکی از پرکاربردترین و قدرتمندترین آنها را معرفی کرده و امکانات و قابلیت‌های آن را به‌طور کامل مورد بررسی قرار خواهیم داد.

نرم‌افزارOllyDbg

یکی از قوی‌ترین دیباگرهای مد کاربر است که امکانات بسیار وسیعی در زمینه تحلیل، بررسی و تغییر فایل‌های اجرایی استاندارد ویندوز (Win32 PE) دارد. در این بخش سعی خواهیم کرد قسمت‌های مختلف این نرم‌افزار را مورد بررسی قرار داده و امکانات و قابلیت‌های آن را تا حد امکان توضیح دهیم. بدیهی است که با توجه به وسعت بسیار زیاد نکات و امکانات این نرم‌افزار، توضیح کامل آن از حوصله این کتاب خارج است در نتیجه سعی خواهیم کرد بیشتر به نکات و قابلیت‌های کلیدی این نرم‌افزار اشاره کنیم.

ابتدا بیایید نگاهی گذرا به برخی از این قابلیت‌ها و خصوصیات این ابزار بیاندازیم.

- دیباگ کردن برنامه‌های Multi Thread.
- سرعت بسیار بالا در تحلیل و بررسی کدهای اسمبلی.
- توانایی بررسی خطاهایی که در هنگام اجرای فایل اجرایی رخ می‌دهد.
- توانایی دیباگ کردن فایل‌های dll به‌صورت مجزا و فراخوانی توابع داخلی آنها.
- توانایی ذخیره کردن تغییرات و تنظیمات اعمال شده برروی فایل‌های اجرایی و dll ها در هنگام عملیات دیباگ و استفاده دوباره از آنها در صورت نیاز.
- توانایی تشخیص پارامترهای ارسالی به توابع استاندارد API در کدهای اسمبلی.
- پشتیبانی از استانداردهای UNICODE و ASCII برای رشته‌های استفاده شده در فایل اجرایی.
- پشتیبانی از انواع گوناگونی از عملیات جستجو در کدهای اسمبلی.
- پشتیبانی از انواع مختلف نقاط توقف شرطی و غیرشرطی در هنگام دیباگ کردن فایل اجرایی.
- ارائه روش‌های بسیار سریع و دقیق برای دنبال کردن دستورات اجرا شده در مراحل دیباگ.
- امکان ایجاد تغییرات در کدهای اسمبلی در مراحل اجرا و دیباگ، بدون نیاز به اجرای دوباره فایل اجرایی.
- تشخیص برخی از ساختارهای متداول مورد استفاده توسط کامپایلرهای امروزی.

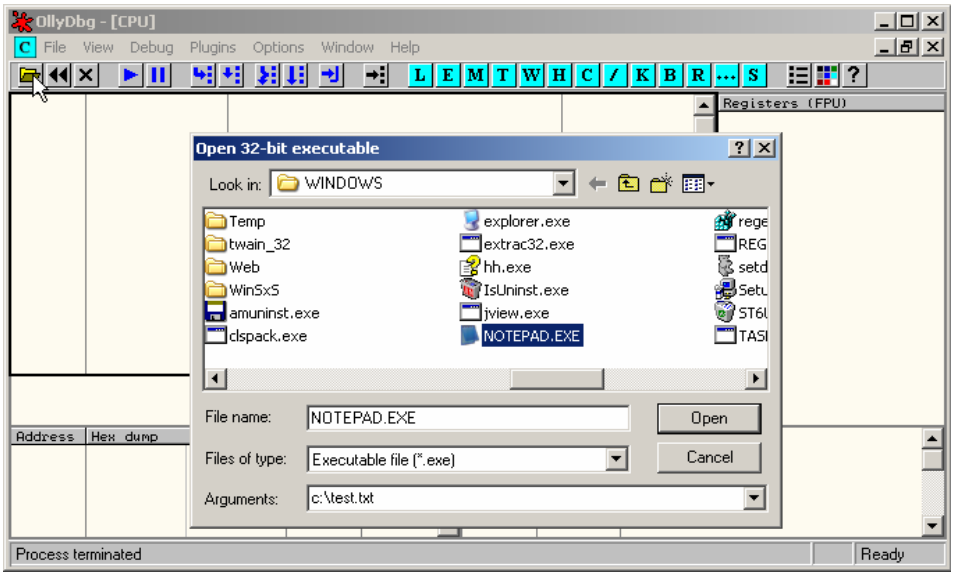
نسخه 1.10 این نرم‌افزار در CD ضمیمه موجود می‌باشد
Tools\OllyDbg

شروع عملیات دیباگ

در این ابزار به سه روش عملیات دیباگ آغاز می‌شود. کاربر می‌تواند برحسب نوع نیاز یکی از آنها را انتخاب کرده و عملیات دیباگ را برای یک فایل اجرایی یا dll آغاز کند.

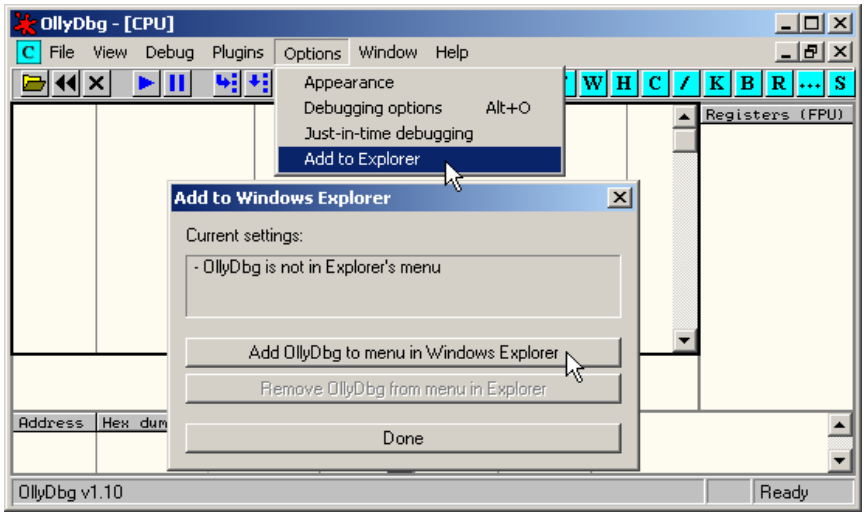
اجرای فایل اجرایی در حالت دیباگ

درصورت نیاز می‌توانید فایل اجرایی موردنظر را در حالت دیباگ اجرا کنید. به این منظور به سادگی فایل موردنظر را باز کرده و در صورتی که پارامترهای خط فرمان خاصی برای اجرای آن باید آورده شود آنها را همانند شکل (۵-۱) در قسمت Arguments ذکر کنید.



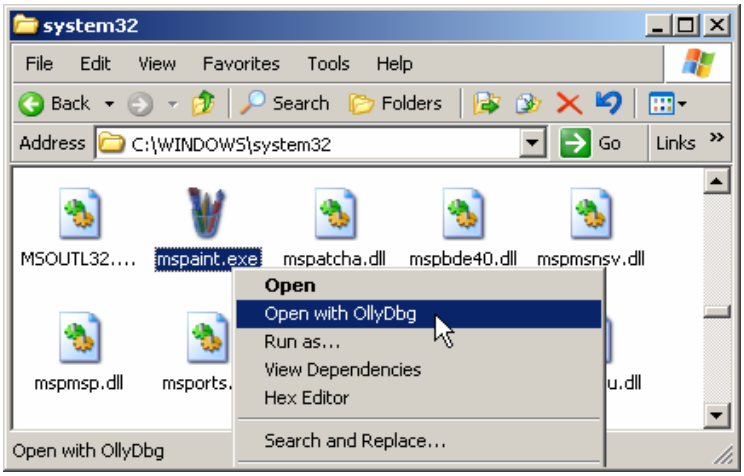
شکل (۵-۱)

به‌منظور سادگی کار می‌توانید گزینه‌ای را برای بازکردن فایل‌های exe و dll به منوی Explorer ویندوز اضافه کنید به این منظور همانند شکل (۵-۲) ابتدا گزینه Add to Exphorer از منوی Options را انتخاب کرده و سپس در پنجره ظاهر شده این گزینه را فعال کنید.



شکل (۵-۲)

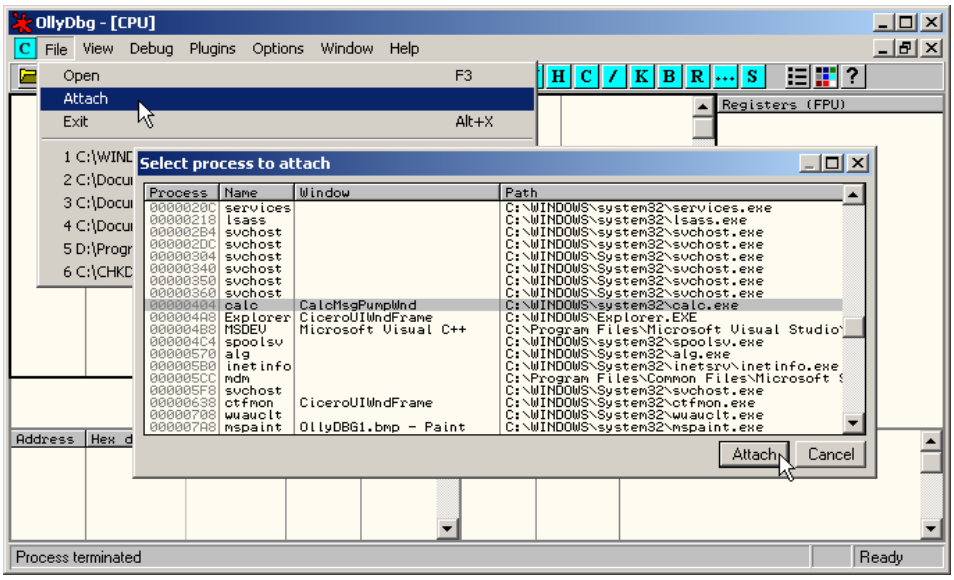
با فعال شدن این گزینه قادر خواهید بود که فایل‌های اجرایی را هنگام مرور کردن آنها در Explorer با استفاده از منوی موجود، به درون OllyDbg بارگذاری کرده و عملیات دیباگ را آغاز کنید.



شکل (۵-۳)

اتصال به فایل‌های اجرایی در حین اجرا

همان‌طور که می‌دانید توابع API استاندارد ویندوز به دیباگرها این امکان را می‌دهند که به‌برنامه‌های در حال اجرا متصل شده و آنها را به مد دیباگ (Single Step) ببرند. به این منظور گزینه Attach را از منوی File انتخاب کنید. با این عمل پنجره Select Process همانند شکل (٥-٤) باز شده و لیستی از کلیه برنامه‌های در حال اجرا در سیستم را به همراه اطلاعات مختصری در مورد هر یک به نمایش می‌گذارد. با استفاده از این لیست برنامه موردنظر را که قصد اتصال به آن را دارید، انتخاب کرده و دکمه Attach را کلیک کنید.



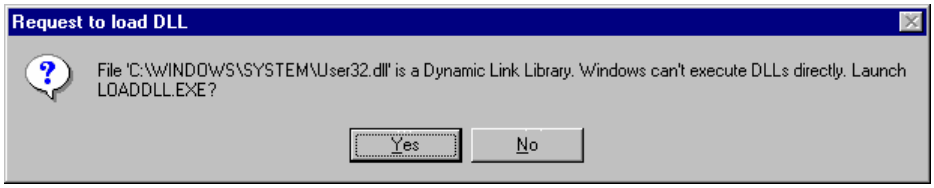
شکل (٥-٤)

اجرای توابع داخلی فایل‌های dll در حالت دیباگ

همان‌طور که می‌دانید اجرای توابع داخلی فایل‌های dll به‌صورت مستقیم ممکن نیست. در نتیجه OllyDbg به‌منظور ایجاد امکان دیباگ برای این نوع فایل‌ها، از یک فایل اجرایی کمکی با نام loaddll.exe استفاده می‌کند. این فایل اجرایی در ابتدا درون فایل اجرایی اصلی این نرم‌افزار قرار دارد. در صورتی که کاربر قصد دیباگ کردن فایل dllای را داشته باشد OllyDbg آن را از داخل فایل اجرایی خود به دیسک کپی کرده و از آن به‌منظور بارگذاری و دیباگ کردن فایل‌های dll استفاده می‌کند. حال با یک مثال نحوه فراخوانی توابع داخلی فایل‌های dll را مورد بررسی قرار خواهیم داد.

در این مثال قصد داریم یکی از توابع داخلی فایل user32 .dll ویندوز را با نام MessageBoxW فراخوانی کنیم.

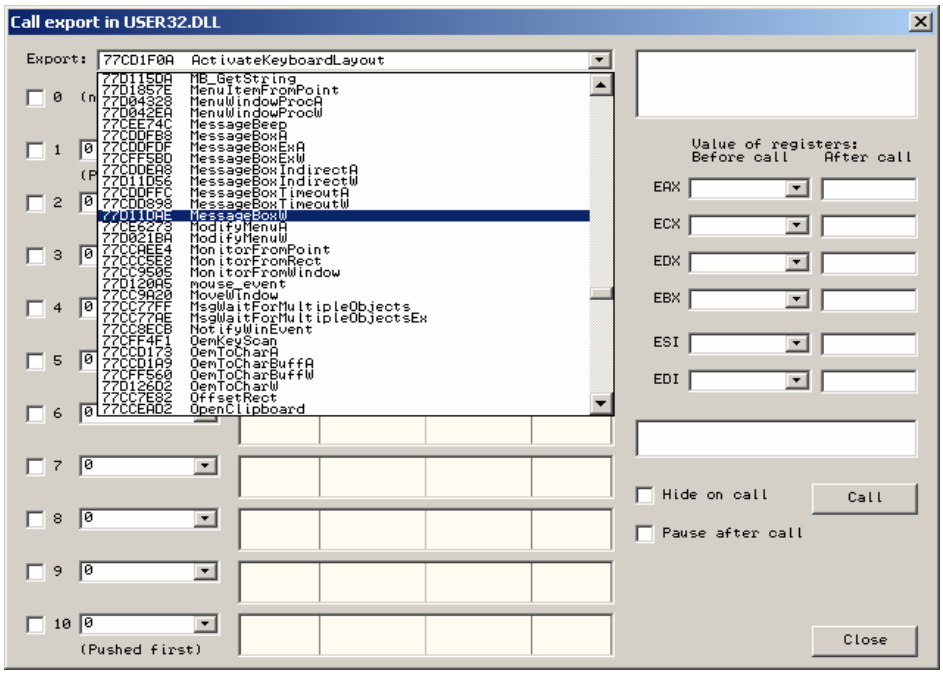
در ابتدا فایل dll مذکور را همانند سایر فایل‌های اجرایی باز می‌کنیم. در صورتی که برای اولین بار این عمل را انجام داده‌اید با پیغام زیر مواجه خواهید شد.



شکل (۵-۵)

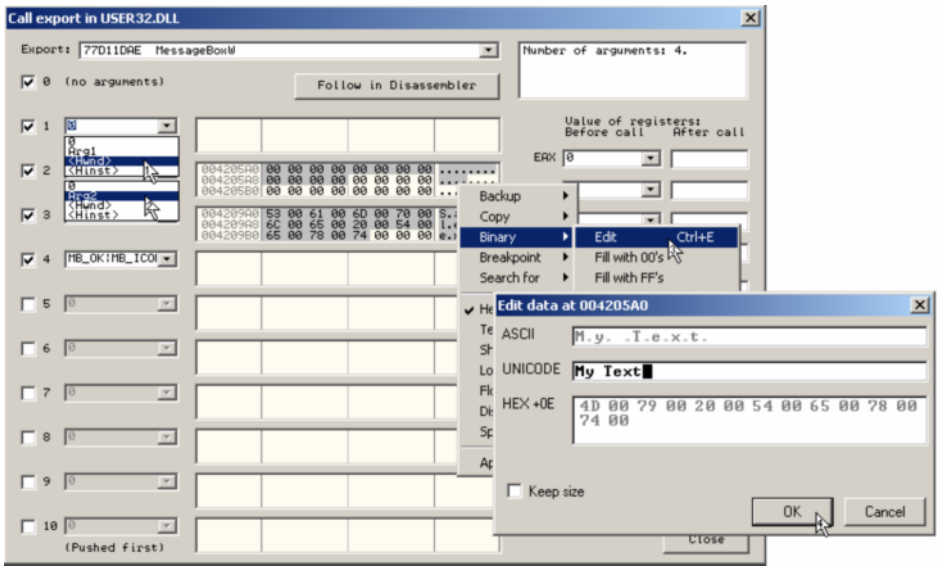
با تأیید عملیات، OllyDbg به‌وسیله فایل اجرایی loaddll.exe فایل dll موردنظر را بارگذاری کرده و عملیات تحلیل و بررسی کدها و دیباگ را آغاز می‌کند. توجه داشته باشید که کدهای مربوط به تابع آماده‌سازی اولیه dll که توسط سازنده آن نوشته شده است به‌طور اتوماتیک در هنگام بارگذاری فایل موردنظر، اجرا خواهد شد. عملیات دیباگ به‌طور معمول کنترلی برروی اجرای این قسمت از کد فایل dll ندارد و این عملیات پس از اجرای این قسمت آغاز می‌شود.

به‌منظور فراخوانی توابع صادر شده توسط dll مذکور، گزینه Call Dll Export را از منوی Debug انتخاب کنید با این عمل پنجره Call Export همانند شکل (۵-۶) نمایش داده می‌شود. به‌منظور فراخوانی تابع MessageBoxW، ابتدا آنرا از لیست Export انتخاب کنید.



شکل (۵-۶)

پس از انتخاب تابع موردنظر، در صورتی که این تابع از توابع API استاندارد ویندوز باشد، تعداد پارامترهای ورودی آن تشخیص داده می‌شود. در مرحله بعد پارامترهای ارسالی به تابع موردنظر تعیین می‌شوند. در صورت نیاز می‌توانید مقدار خاصی را برای یک پارامتر در نظر بگیرید و یا اینکه از گزینه‌های پیش‌فرض استفاده کنید. در شکل (۵-۷) روش آماده‌سازی پارامترهای ورودی برای تابع MessageBoxW را مشاهده می‌کنید.



شکل (۵-۷)

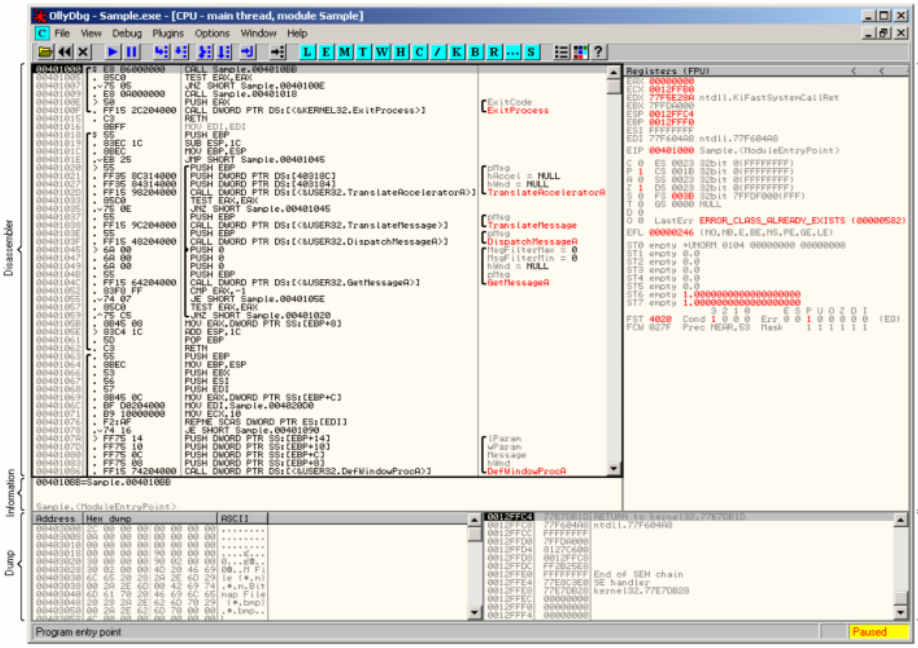
حال که مقادیر مورد نیاز برای پارمترهای تابع آماده شده است می‌توانید دکمه Call را کلیک کرده و تابع MessageBoxW را فراخوانی کنید.

در شکل (۵-۸) نتیجه فراخوانی این تابع را با استفاده از پارامترهای ارسالی مشاهده می‌کنید.

شکل (۵-۸)

پنجره اصلی Olly Dbg (CPU window)

پس از پایان عملیات تحلیل و بررسی و بارگذاری فایل اجرایی، پنجره اصلی ظاهر شده و عملیات دیباگ آغاز می‌گردد. در شکل (۵-۹) این صفحه را مشاهده می‌کنید.



شکل (۵-۹)

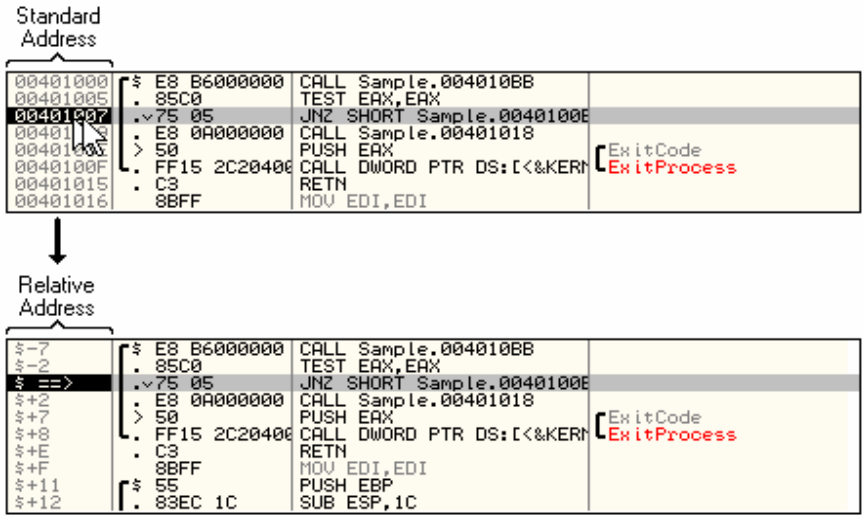
همان‌طور که مشاهده می‌کنید این صفحه به پنج قسمت تقسیم شده است. هر یک از این قسمت‌ها وظیفه خاصی داشته و اطلاعاتی را راجع به جزئیات عملیات دیباگ، وضعیت فعلی ثبات‌ها و.... به نمایش می‌گذارند. با استفاده از کلید Tab می‌توانید برروی این قسمت‌ها حرکت داشته باشید. در ادامه نحوه عملکرد قسمت‌های مختلف این صفحه را مورد بررسی قرار خواهیم داد.

۱- Disassembler

دراین قسمت کدهای Disassemble شده فایل اجرایی موردنظر به‌نمایش گذاشته می‌شود. همان‌طور که مشاهده کردید قسمت Disassembler به چهار ستون تقسیم می‌شود که هر یک اطلاعات خاصی را به نمایش گذاشته و وظایف خاصی را نیز انجام می‌دهند.

Address

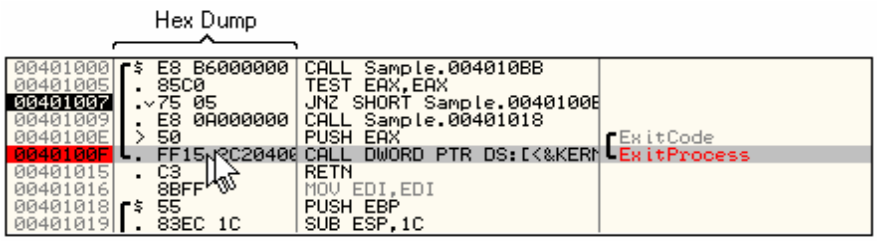
در این ستون آدرس مجازی سطرها در فایل اجرایی نمایش داده می‌شود. با Double Click کردن برروی آدرس‌های این ستون، آدرس‌های مجازی به آدرس‌های نسبی، نسبت به سطر فعلی تبدیل می‌شوند.



شکل (۵-۱۰)

Hex Dump

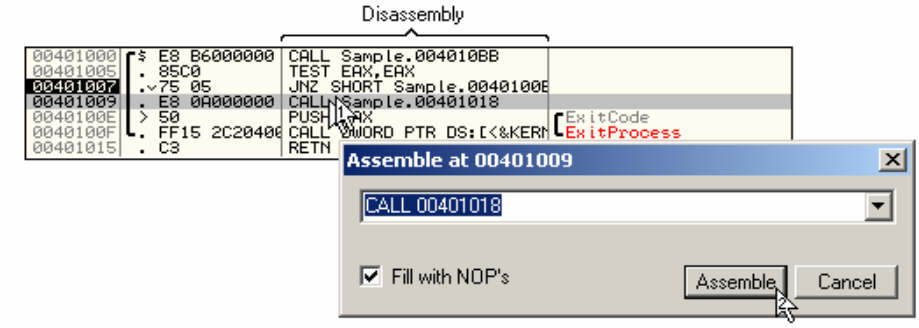
در این قسمت اطلاعات خام Disassemble نشده برای دستوالعمل موردنظر به‌صورت Hex نمایش داده می‌شود. همان‌طور که در شکل (۵-۱۱) مشاهده می‌کنید، علاوه بر اطلاعات مذکور این ستون حاوی برخی کاراکترهای کمکی نیز هست. این کارکترها اطلاعات بسیار مفیدی را راجع به‌آدرس‌های شروع و پایان توابع، مبداء و مقصد دستورالعمل‌های پرشی و ارجاع‌ها به توابع مشخص می‌کنند.



شکل (۵-۱۱)

Disassembly

در این ستون کدهای اسمبلی به نمایش گذاشته می‌شوند. در صورت نیاز می‌توانید در این کدها تغییراتی را اعمال کنید. توجه داشته باشید که تغییرات اعمال شده تنها در طول مراحل دیباگ معتبر هستند و تأثیری برروی فایل اجرایی موردنظر ندارند. به این منظور برروی دستوالعمل موردنظر در ستون Disassemble، Double Click کنید. با این کار پنجره Assemble همانند شکل (۵-۱۲) نمایش داده می‌شود.

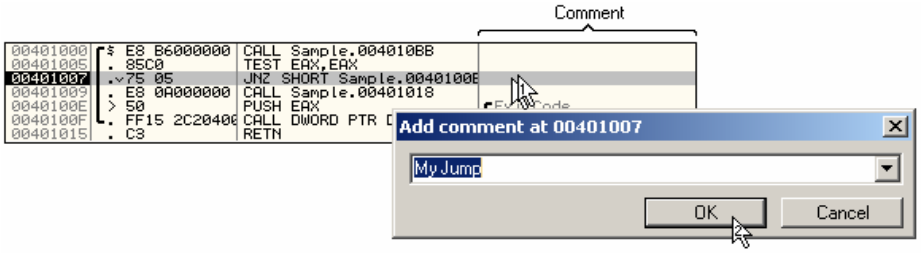


شکل (۵-۱۲)

در این پنجره می‌توانید دستورالعمل موجود را با دستورالعمل موردنظر خود جایگزین کنید. در صورتی که گزینه Fill with Nop's فعال باشد، اگر فضای اشغالی دستورالعمل جدید کمتر از فضای اشغالی دستوالعمل قبلی باشد، فضای اضافی توسط دستورالعمل‌های بی‌اثر NOP پرخواهد شد. در صورتی که فضای اشغالی دستورالعمل جدید بیشتر از فضای اشغالی دستورالعمل قبلی باشد، فضای اضافی مورد نیاز از دستورالعمل‌های بعدی گرفته خواهد شد که باعث تخریب آنها می‌شود. در نتیجه همیشه دقت داشته باشید که فضای اشغالی دستورالعمل جدید کوچکتر یا مساوی دستورالعمل قبلی بوده و باعث تخریب دستورالعمل‌های بعدی نگردد.

Comment

در این ستون اطلاعات بسیار مفیدی راجع به فراخوانی‌های API، رشته‌ها و سایر ساختارهای شناسایی شده در مراحل تحلیل و بررسی آورده شده است. در صورت نیاز می‌توانید توضیحات دلخواه خود را نیز به سطر موردنظر اضافه کنید. به این منظور برروی ستون comment متناظر با سطر موردنظر Double click کنید. با این عمل پنجره Add Comment همانند شکل (۵-۱۳) نمایش داده می‌شود.



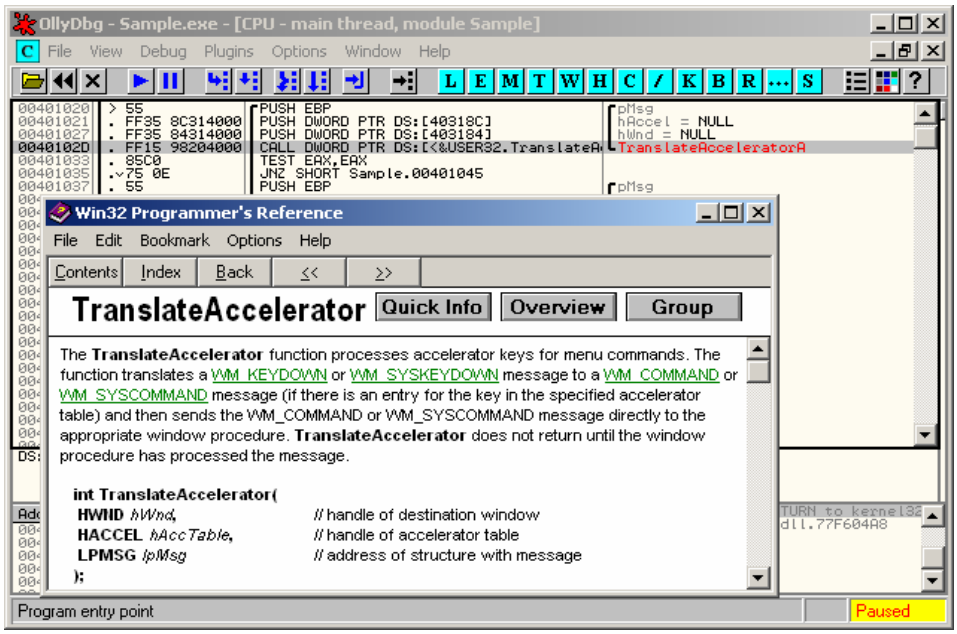
شکل (۵-۱۳)

در این پنجره می‌توانید توضیحات دلخواه را به سطر موردنظر اضافه کرده و یا توضیحات موجود را تغییر دهید در صورتی که دستور موردنظر یک فراخوانی از توابع API استاندارد باشد، به منظور دریافت اطلاعات دقیق‌تر راجع به تابع مذکور ابتدا آن را انتخاب کرده و سپس از کلیدهای Ctrl + F1 استفاده کنید. OllyDbg از فایل راهنمای استاندارد ویندوز برای توابع API به این منظور استفاده می‌کند که win32.hlp نام دارد. این فایل را باید به‌طور جداگانه تهیه کرده و با استفاده از گزینه Select API Help از منوی Help آنرا به این نرم‌افزار معرفی کنید.

این فایل راهنما در CD ضمیمه موجود می باشد

Tools\Win32.hlp

پس از معرفی این فایل راهنما می‌توانید با استفاده از کلیدهای Ctrl + F1 اطلاعات دقیقتری راجع به توابع API فراخوانی شده بدست آورید.



شکل (۵-۱۴)

۲- Information

در این قسمت از پنجره اصلی اطلاعات مفیدی راجع به دستورالعمل فعلی، آرگومان‌های ارسال شده به آن و مقادیر هر یک به نمایش گذاشته می‌شود. این اطلاعات در هنگام اجرای خط به خط فایل اجرایی، راهنمای بسیار خوبی است. این قسمت گزینه‌های مختلفی را نیز برای دنبال کردن آدرس‌ها و مقادیر نمایش داده شده تدارک دیده است. در ادامه به توضیح آنها خواهیم پرداخت.

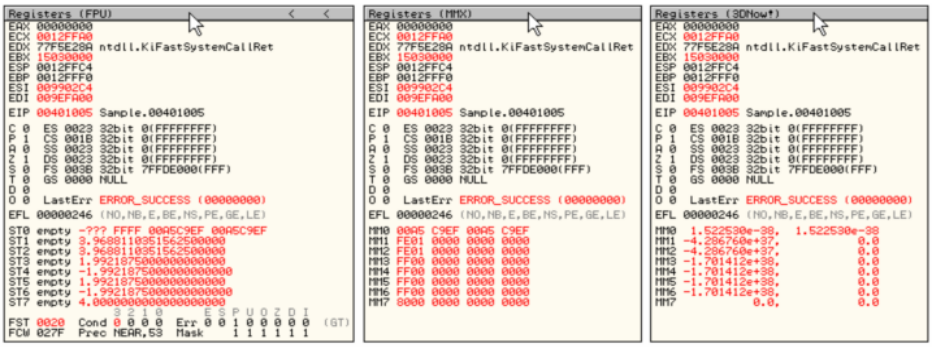
۳- Dump

در این قسمت داده‌های خام قسمت‌های مختلف فایل و یا حافظه با استفاده از فرمت‌های استاندارد از قبیل Hex , Byte , Text , Integer , Float و ... به نمایش گذاشته می‌شود. علاوه براین در قسمت Dump گزینه‌های مختلفی به‌منظور جستجو و حرکت برروی این داده‌ها نیز درنظر گرفته شده است. در ادامه به توضیح آنها خواهیم پرداخت.

۴- Registers

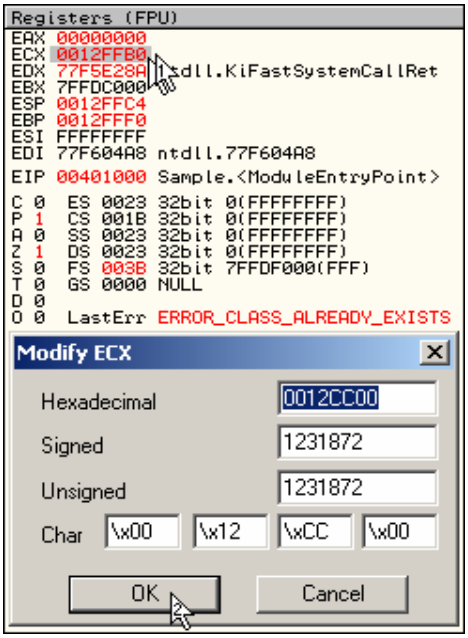
در این قسمت مقادیر فعلی ثبات‌های سیستم به نمایش گذاشته شده و قابلیت ایجاد تغییرات در این مقادیر در صورت امکان برای کاربر ایجاد می‌گردد. در صورت نیاز می‌توانید با کلیک برروی نوار

عنوان این قسمت، مقادیر موجود در ثبات‌های MMX , 3DNOW و یا FPU را نیز مورد بررسی قرار داده و یا تغییر دهید.



شکل (۵-۱۵)

در صورت نیاز، به‌منظور تغییر مقدار فعلی ثبات‌ها می‌توانید برروی مقدار موجود در ثبات موردنظر Double Click کنید. با این عمل پنجره Modify همانند شکل (۵-۱٦) ظاهر شده و امکان ایجاد تغییرات را به صورت‌های مختلف فراهم می‌آورد.



شکل (۵-۱٦)

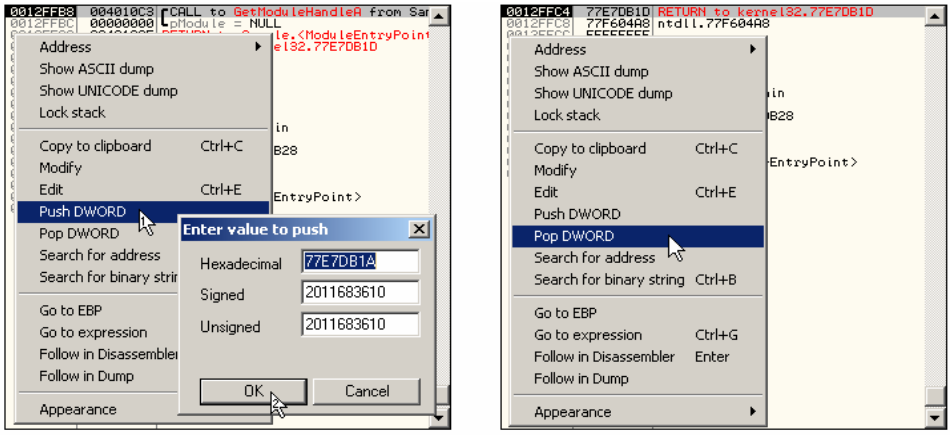
به‌منظور ایجاد سریع‌تر این تغییرات می‌توانید مقدار ثبات موردنظر را انتخاب کرده و از کلیدهای + و – برای افزایش و یا کاهش این مقدار به میزان یک واحد استفاده کنید.

در این قسمت گزینه‌های مختلفی نیز برای دنبال کردن آدرس‌ها و مقادیر نمایش داده شده در سایر قسمت‌ها از جمله Disassembler , Stack , Dump درنظر گرفته شده است. در ادامه به توضیح آنها خواهیم پرداخت.

Stack -۵

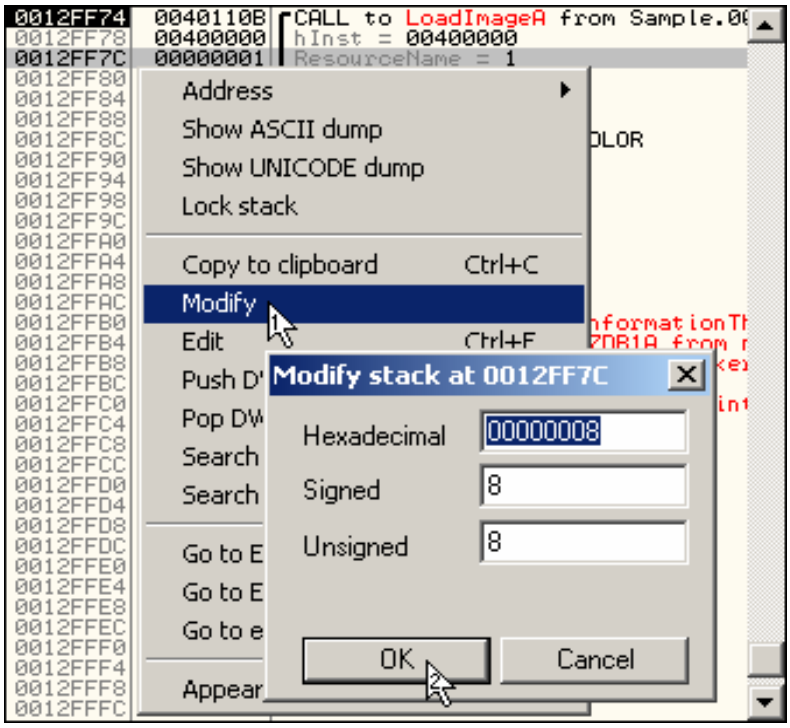
در این قسمت فضای stack و پارامترهای موجود در آن به نمایش گذاشته شده و امکانات وسیعی به‌منظور جستجو و دنبال کردن آدرس‌ها و مقادیر موجود، در سایر نواحی صفحه اصلی از جمله Disassembler , Dump در نظر گرفته شده است.

همان‌طور که می‌دانید معمولاً از stack به‌منظور ذخیره آدرس‌های بازگشت در فراخوانی‌ها و نیز به‌منظور ارسال پارامترها به توابع استفاده می‌شود. در صورت نیاز می‌توانید مقادیر جدیدی را به stack اضافه کرده (push) و یا آخرین عنصر را از آن خارج کنید (pop). به این منظور پس از کلیک راست، از گزینه‌های Pop DWORD, Push DWORD مطابق شکل (۵-۱۷) استفاده کنید.



شکل (۵-۱۷)

در صورت نیاز می‌توانید در مقادیر و داده‌های موجود در Stack، تغییرات موردنظر خود را اعمال کنید. به این منظور پس از انتخاب عنصر موردنظر و کلیک راست، گزینه Modify را انتخاب کنید. با این عمل پنجره Modify همانند شکل (۵-۱۸) نمایش داده شده و امکان ایجاد تغییرات را در اختیار شما قرار می‌دهد.


0012FF74 0040110B CALL to LoadImageA from Sample.0
0012FF78 00400000 hInst = 00400000
0012FF7C 00000001 ResourceName = 1
Address
Show ASCII dump
Show UNICODE dump
Lock stack
Copy to clipboard Ctrl+C
Modify
Edit Ctrl+E
Modify stack at 0012FF7C
Hexadecimal 00000008
Signed 8
Unsigned 8
OK
Cancel
1
2

شکل (۵-۱۸)

نقاط توقف (Breakpoints)

OllyDbg از انواع گوناگون نقاط توقف پشتیبانی می‌کند. هر یک از این نقاط توقف کاربردهای خاصی دارند. تسلط به آنها، نحوه به کارگیری و شرایط استفاده از هر یک می‌تواند در مراحل دیباگ بسیار کارگشا بوده و باعث ایجاد تسریع در این عملیات گردد. در نقطه مقابل، تشخیص نادرست شرایط و به کارگیری نادرست آنها می‌تواند عملیات دیباگ را بسیار کند کرده و حتی با شکست مواجه کند. می‌توان گفت که موفقیت شما در این عملیات با انتخاب نقاط توقف مناسب بستگی مستقیم دارد و این امر جز با تمرین فراوان دست یافتنی نیست.

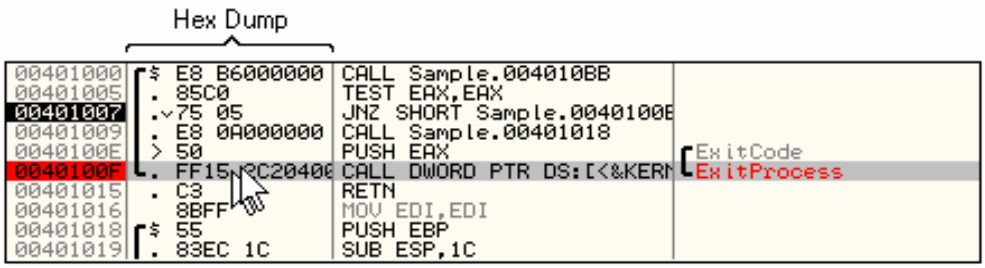
در ادامه انواع نقاط توقف در OllyDbg و نحوه به کارگیری هریک را مورد بررسی قرار خواهیم داد.

۱- نقاط توقف معمولی

ساده‌ترین و پرکاربردترین نقاط توقف هستند. پس از قراردادن یکی از این نقاط در آدرس موردنظر، روند اجرایی برنامه با رسیدن به دستورالعمل متناظر با آن آدرس متوقف شده و کنترل اجرا به Debugger منتقل می‌شود. از این مرحله به بعد کاربر می‌تواند بررسی‌ها و یا تغییرات دلخواه خود را انجام داده، به اجرای خط به خط برنامه بپردازد و یا روند اجرایی را ادامه دهد. این نقاط توقف با درج دستورالعمل وقفه دیباگ (INT3) در دستورالعمل‌ها ایجاد می‌شوند. تعداد این‌گونه نقاط توقف محدودیت خاصی نداشته و می‌توانید هر تعداد از آنها را ایجاد کرده و از آنها استفاده کنید. توجه داشته باشید که این‌گونه نقاط توقف فقط در قسمت کد معتبر بوده و استفاده از آنها در قسمت‌های data هیچگونه عملکرد خاصی نداشته و تنها باعث تخریب آنها می‌گردد.

OllyDbg نقاط توقف تعیین شده در هنگام مراحل دیباگ را در فایل‌هایی ذخیره می‌کند. این فایل‌ها برای هر فایل باز شده به‌وسیله این نرم‌افزار به‌طور جداگانه ایجاد شده و مورد استفاده قرار می‌گیرد. در صورتی که فایل موردنظر دوباره توسط OllyDbg بارگذاری شود، این نقاط توقف به‌طور خودکار دوباره اعمال می‌گردند.

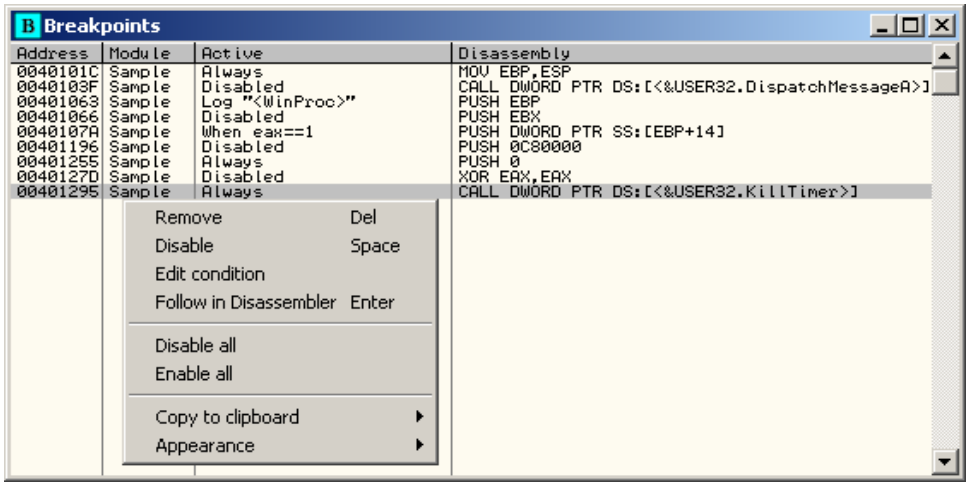
ساده‌ترین روش ایجاد این نقاط توقف، انتخاب آدرس موردنظر در قسمت Disassembler و فشردن کلید F2 است. Double Click کردن در قست Hex Dump از آدرس موردنظر نیز اثر مشابهی خواهد داشت.



شکل(٤-١٩)

به‌منظور غیرفعال کردن نقاط توقف ایجاد شده می‌توانید عملیات فوق را تکرار کنید.

به‌منظور مشاهده و مدیریت نقاط توقف ایجاد شده می‌توانید از پنجرهٔ Breakpoints استفاده کنید. این پنجره از طریق گزینهٔ Breakpoints از منوی View قابل دسترسی است. همان‌طور که در شکل (۵-۲۰) مشاهده می‌کنید در این پنجره لیست کاملی از نقاط توقف به همراه توضیحات کوتاهی در مورد هر یک نمایش داده شده و امکان ایجاد تغییرات و غیرفعال کردن آنها به کاربر داده شده است.



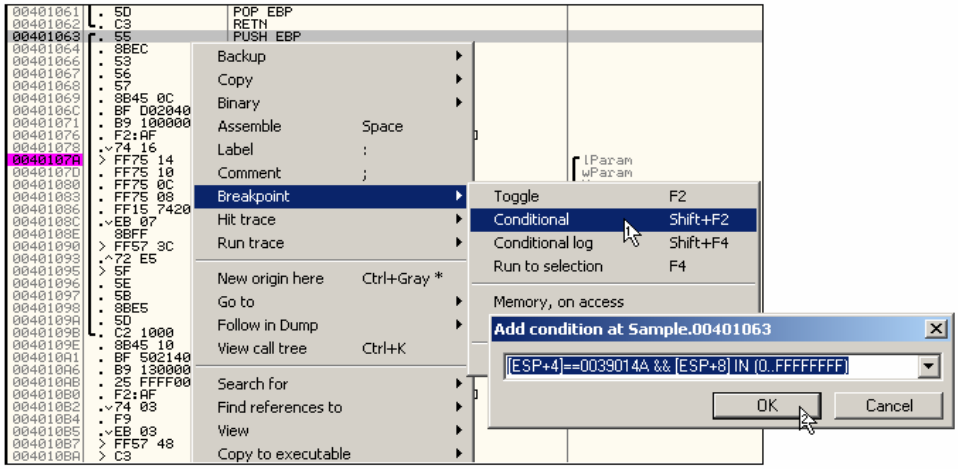
شکل (۵-۲۰)

۲- نقاط توقف شرطی

این‌گونه نقاط توقف دقیقاً همانند نقاط توقف معمولی هستند با این تفاوت که OllyDbg در هنگام مواجه شدن با آنها شرط خاصی را که توسط کاربر تعیین شده است ، بررسی کرده و در صورت برقراری شریط مذکور روند اجرایی توسط دستورالعمل INT3 (وقفهٔ دیباگ) متوقف می‌شود. وجود

این‌گونه نقاط توقف باعث کندتر شدن عملیات دیباگ می‌گردد. یکی از پرکاربردترین موارد استفاده از این نقاط، توقف در دیباگ کردن پروسیجرهای پنجره برحسب پیغام‌های رسیده است.

به‌منظور ایجاد اینگونه نقاط توقف ابتدا آدرس موردنظر را در قسمت Disassembler انتخاب کرده و سپس کلیدهای Shift+F2 را فشار دهید. با این عمل پنجرهٔ Add Condition همانند شکل (۵-۲۱) نمایش داده شده و امکان تعیین شرط یا شروط موردنظر را به کاربر می‌دهد.



شکل (۵-۲۱)

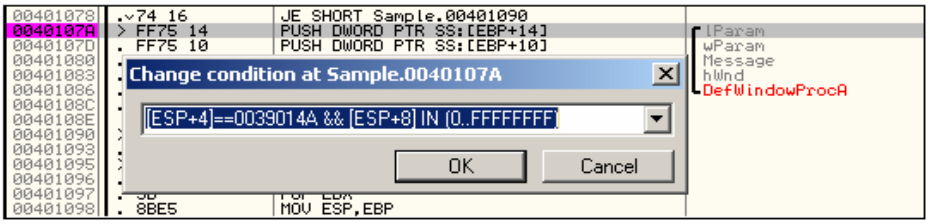
OllyDbg گرامر خاصی را برای تعیین شرطها درنظر گرفته است که علاوه بر خوانایی خوب، امکان تعریف شرطهای بسیار پیچیده را نیز دارا می‌باشد. در ادامه با استفاده از مثال‌هایی به بررسی نحوهٔ تعریف شرطها در این نرم‌افزار خواهیم پرداخت. به‌منظور بررسی دقیق‌تر و کامل‌تر گرامر مورد استفاده می‌توانید به مستندات این نرم‌افزار مراجعه کنید. در جدول (۵-۱) با استفاده از مثال‌هایی، انواع شرطهای معتبر دسته‌بندی و توضیح داده شده است.

Category	Sample	Description
Numeric Systems	10	constant 0x10 (unsigned). All integer constants are assumed hexadecimal unless followed by a decimal point;
	10.	decimal constant 10 (signed);
Constants	WM_PAINT	Window Paint Message (Value = HF)
Characters	'A'	character constant 0x41

Category	Sample	Description
Registers	EAX	contents of register EAX, interpreted as unsigned number.
	EAX.	contents of register EAX, interpreted as signed number.
Memory Addresses	[123456]	contents of unsigned doubleword at address 123456. By default, OllyDbg assumes doubleword operands.
	DWORD PTR [123456]	same as above. Keyword PTR is optional.
	[SIGNED BYTE 123456]	contents of signed byte at address 123456. OllyDbg allows both MASM- and IDEAL-like memory expressions.
	[[123456]]	doubleword at address that is stored in doubleword at address 123456.
Strings	STRING [123456]	ASCII zero-terminated string that begins at address 123456. Square brackets are necessary because you display the contents of memory
Conditions	EAX.<0.	0 if EAX is in range 0..0x7FFFFFFF and 1 otherwise. Notice that constant 0 is also signed. When comparing signed with unsigned, OllyDbg always converts signed operand to unsigned.
	EAX<0	always 0 (false), because unsigned numbers are always positive.
	(EAX>0 && EAX<9) || ECX==2	
	[STRING 123456]=="Brown fox"	true if memory starting from address 0x00123456 contains ASCII string "Brown fox", "BROWN FOX JUMPS", "brown fox???" or similar. The comparison is case-insensitive and limited in length to the length of text constant.
	EAX=="Brown fox"	same as above, EAX is treated as a pointer.
	UNICODE [EAX]=="Brown fox"	OllyDbg treats EAX as a pointer to UNICODE string, converts it to ASCII and compares with text constant.
	[ESP+8]==WM_PAINT	in expressions, you can use hundreds of symbolic constants from Windows API.

جدول (۵-۱)

در صورت نیاز به‌منظور تغییر شرط‌های اعمال شده می‌توانید از پنجرهٔ Breakpoints استفاده کرده و یا در آدرس موردنظر همانند مراحل قبل کلیدهای Shift+F2 را فشار دهید. با این عمل پنجرهٔ Change Condition همانند شکل (۵-۲۲) نمایش داده شده و امکان ایجاد تغییرات در شرط‌ها را به کاربر می‌دهد.

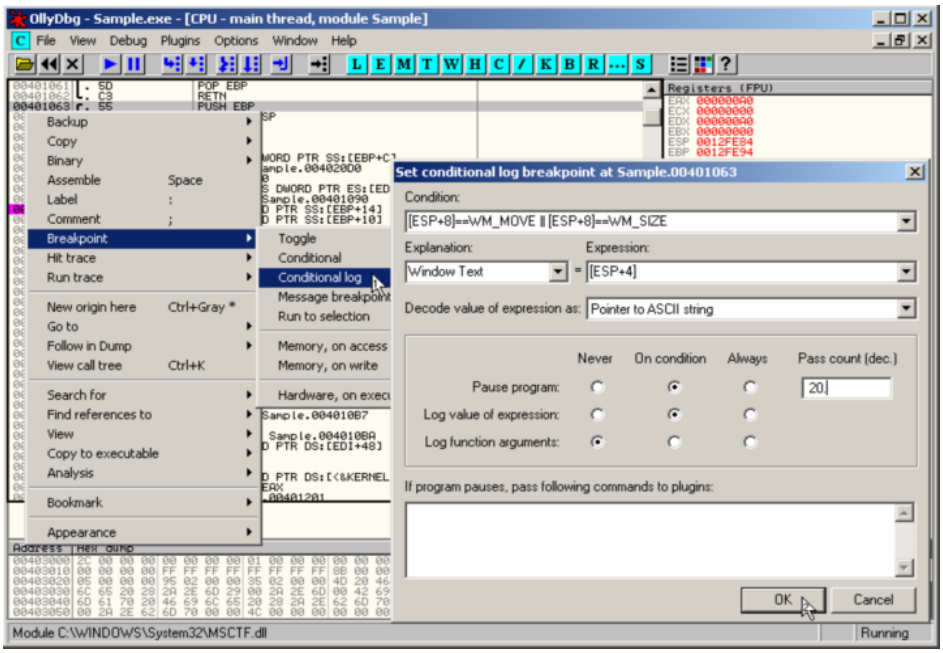


شکل (۵-۲۲)

۳- نقاط توقف شرطی همراه با گزارش

این‌گونه نقاط توقف دقیقاً همانند نقاط توقف شرطی معمولی هستند با این تفاوت که می‌توانند علاوه بر عملکرد یک نقطهٔ توقف، عملیات گزارش‌گیری را نیز به‌صورت‌های مختلف انجام دهند. این گزارش‌ها می‌توانند شامل اطلاعات موجود در ثبات‌ها و یا آدرس‌های حافظه با فرمت‌های دلخواه باشند. همان‌طور که در ادامه اشاره خواهیم کرد، خروجی این گزارش‌ها می‌تواند در فایل ذخیره شده و یا در پنجرهٔ Log مشاهده شود.

به‌منظور ایجاد این‌گونه نقاط توقف ابتدا آدرس موردنظر را در قسمت Disassembler انتخاب کرده و سپس کلیدهای Shift+F4 را فشار دهید. با این عمل پنجرهٔ Set Conditional Log Breakpoint همانند شکل (۵-۲۳) نمایش داده می‌شود.



شکل (۵-۲۳)

همان‌طور که مشاهده می‌کنید این پنجره دارای قسمت‌های مختلفی است که در ادامه به بررسی عملکرد آنها خواهیم پرداخت.

Condition

دراین قسمت شرط مورد نظر تعیین می‌شود که از گرامر ذکر شده در جدول (۵-۱) پیروی می‌کند. در صورت خالی بودن این فیلد، هیچ‌گونه شرطی اعمال نمی‌شود.

Explanation

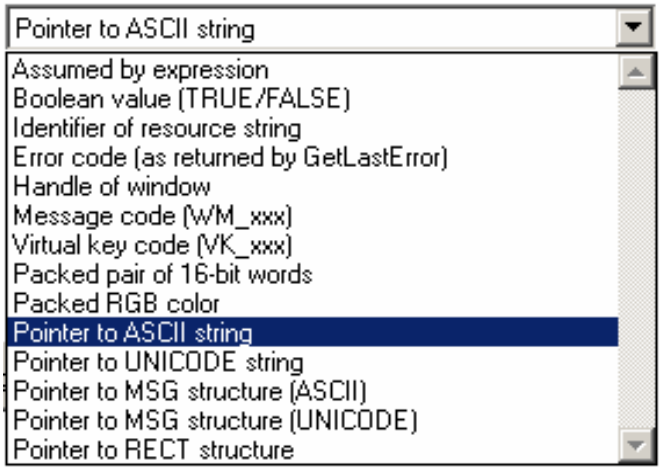
در این فیلد می‌توانید توضیحات کوتاهی را در مورد گزارش خود ذکر کنید. در صورت ایجاد چندین نمونه از این نقاط توقف، این توضیحات می‌تواند شناسه‌ای برای گزارش مربوطه در لیست گزارش‌ها باشد.

Expression

در این قسمت اطلاعات موردنظر که قصد نمایش آنها را در گزارش دارید مشخص می‌شود. این اطلاعات می‌تواند مقادیر موجود در ثبات‌ها و یا حافظه باشد که از قوانین جدول (۵-۱) پیروی می‌کند.

Decode Value of expression as

در این قسمت فرمت مورد نظر برای نمایش اطلاعات ذکر شده در قسمت Expression مشخص می‌شود که می‌تواند یکی از فرمت‌های مشخص شده در شکل (۵-۲٤) باشد.



شکل (۵-۲٤)

Pause Program

در این قسمت گزینه‌هایی به‌منظور توقف برنامه در مراحل دیباگ وجود دارد. در صورت انتخاب گزینهٔ On Condition، نقطهٔ توقف تنها در صورت برقراری شرط ذکر شده باعث توقف (INT3) می‌گردد. در فیلد Pass Count می‌توانید تعداد ملاقات‌های نقطهٔ توقف را که باید نادیده گرفته شوند مشخص کنید. در شکل (۵-۲۳)، ۲۰ ملاقات اول نقطهٔ توقف با درنظرگرفتن شرط، نادیده گرفته شده است. درنتیجه این نقطهٔ توقف در ملاقات ۲۱ ام به بعد باعث متوقف شدن روند اجرایی می‌گردد.

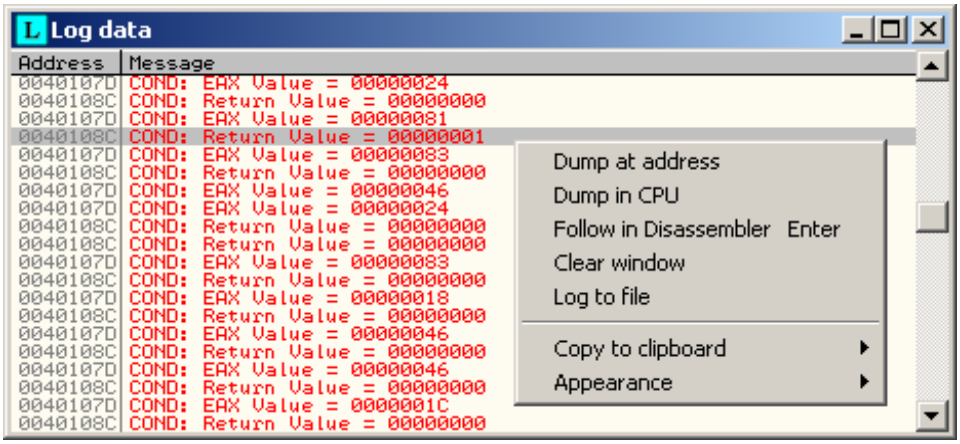
Log Value of Expression

در این قسمت شرایط گزارش‌گیری مشخص می‌شود که گزینه‌هایی مشابه گزینه‌های قسمت قبل دارد.

Log Function Arguments

در صورتی که تابع و پارامترهای آن برای OllyDbg مشخص شده باشد، (به عنوان مثال فراخوانی‌های API و یا پروسجیرهای پنجره) این گزینه‌ها می‌توانند گزارشی از کلیهٔ پارامترهای ارسالی به تابع موردنظر را در هنگام فراخوانی تهیه کنند.

پس از مشخص کردن جزئیات و ایجاد نقطهٔ توقف به روش مذکور، در صورت اجرای فایل موردنظر به‌وسیلهٔ OllyDbg، گزارشی از نحوهٔ عملکرد این‌گونه نقاط توقف در پنجرهٔ Log Data نمایش داده خواهد شد. برای مشاهدهٔ این پنجره از کلیدهای Alt+L استفاده کنید. با این عمل پنجره Log Data همانند شکل (۵-۲۵) نمایش داده می‌شود.



شکل (۵-۲۵)

همان‌طور که مشاهده می‌کنید در این پنجره گزارش‌هایی از نقاط توقف باتوجه به گزینه‌ها و تنظیمات معین شده برای هر یک به نمایش گذاشته شده است. بدیهی است که بررسی این گزارش‌ها از دنبال کردن خط به خط برنامه و مشاهدهٔ مقادیر، بسیار ساده‌تر و عملی‌تر است.

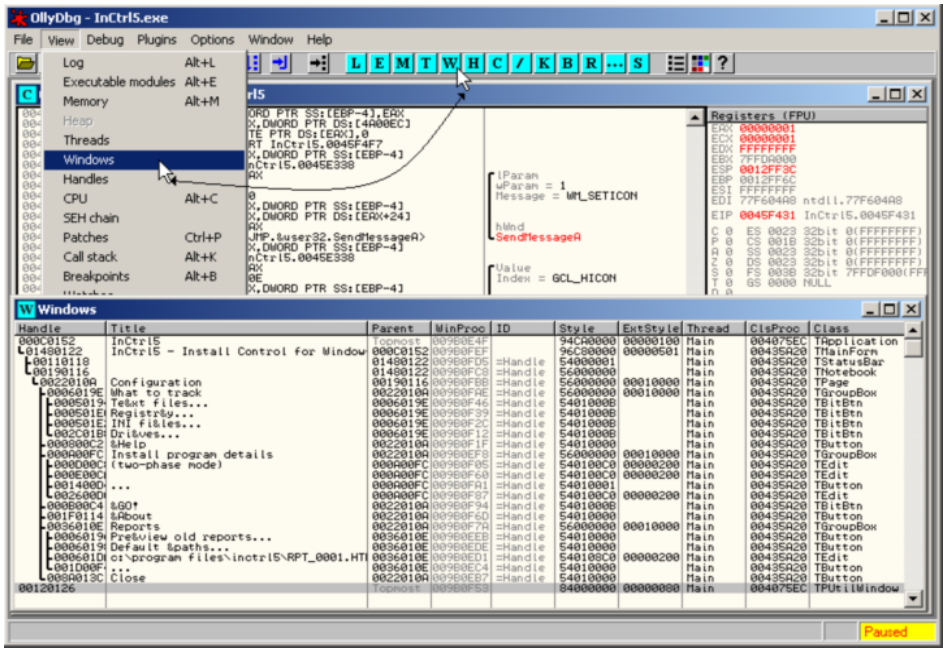
در صورت نیاز با فعال کردن گزینهٔ Log to file در منوی این پنجره و انتخاب فایل خروجی می‌توانید خروجی را به فایل موردنظر منتقل کنید.

۴- نقاط توقف برای پروسیجرهای پنجره

این‌گونه نقاط توقف دقیقاً همانند نقاط توقف بررسی شده در قسمت قبل هستند با این تفاوت که شرط‌های مورد نیاز برای آنها به طور خودکار توسط OllyDbg ایجاد می‌گردد. از این نقاط توقف به‌منظور کنترل پارامترهای ارسال شده به پروسیجرهای پنجره استفاده‌های فراوان می‌شود.

این‌گونه نقاط توقف به سادگی و از طریق پنجرهٔ Windows تعریف شده و مورد استفاده قرار می‌گیرند. در ادامه به بررسی اجزاء و نحوهٔ عملکرد این پنجره خواهیم پرداخت.

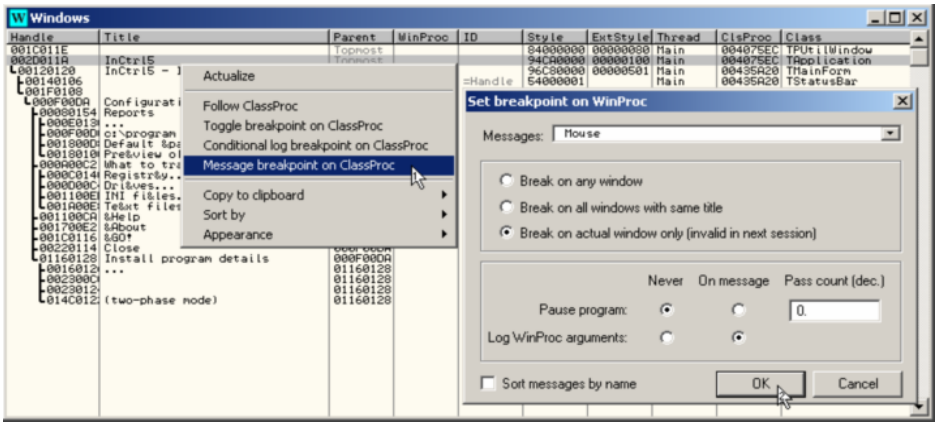
به‌منظور ایجاد این نوع از نقاط توقف ابتدا گزینهٔ Windows را از منوی View انتخاب کنید. با این عمل پنجرهٔ Windows همانند شکل (۵-۲۶) ظاهر شده و لیستی از پنجره‌های ایجاد شده توسط فایل اجرایی مورد نظر را به نمایش می‌گذارد.



شکل (۵-۲۶)

همان‌طور که می‌دانید پنجره‌های مورد نیاز برنامه‌ها در ویندوز، پس از اجرا ایجاد می‌شوند. در نتیجه قبل از اینکه بتوانید پنجره‌های مورد استفادهٔ یک برنامه را مشاهده کنید باید آن را توسط OllyDbg اجرا کرده باشید در غیراین‌صورت لیست نمایش داده شده در پنجرهٔ Windows خالی خواهد بود.

به‌منظور تعیین نقاط توقف ابتدا پنجرهٔ موردنظر را در لیست نمایش داده شده در پنجرهٔ Windows انتخاب کرده و سپس با استفاده از منوی موجود گزینهٔ Message breakpoint را انتخاب کنید. با این عمل پنجرهٔ Set Breakpoint همانند شکل (۵-۲۷) ظاهر می‌شود.



شکل (۵-۲۷)

همان‌طور که مشاهده می‌کنید این پنجره دارای قسمت‌های مختلفی است که در ادامه به توضیح آنها خواهیم پرداخت.

Messages

در این قسمت پیغام و یا نوع پیغام‌های موردنظر تعیین می‌گردد که در حقیقت به عنوان فیلتری برای گزارش‌گیری از پیغام‌ها محسوب می‌شود. با تعیین این فیلترها می‌توانید برروی پیغام‌های موردنظر خود متمرکز شوید.

همان‌طور که در این لیست مشاهده خواهید کرد، این فیلتر می‌تواند یک پیغام خاص و یا گروهی از پیغام‌های مرتبط با هم باشد. در جدول (۵-۲) این پیغام‌ها برحسب موضوع گروه‌بندی شده‌اند. در صورت نیاز می‌توانید به‌منظور دریافت اطلاعات دقیق‌تر راجع به این پیغام‌ها و روش‌های استفاده از آنها به مستندات API از قبیل Win32 API Reference مراجعه کنید که در CD ضمیمه نیز موجود می‌باشد. (Win32.hlp)

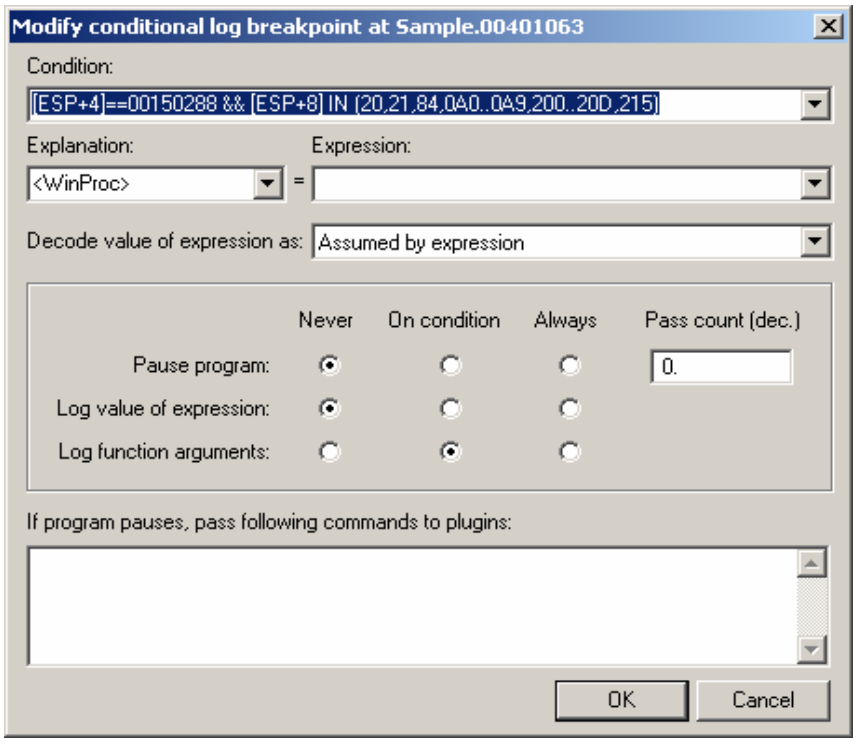
Category	Messages
Creation and destruction	WM_CREATE, WM_DESTROY, WM_CLOSE, WM_QUERYENDSESSION, WM_QUIT,WM_ENDSESSION, WM_NCCREATE, WM_NCDESTROY, WM_INITDIALOG
Window activation	WM_ACTIVATE, WM_SETFOCUS, WM_KILLFOCUS,WM_ENABLE, WM_SHOWWINDOW, WM_CHILDACTIVATE, WM_QUERYNEWPALETTE
Window position and size	WM_MOVE, WM_SIZE, WM_QUERYOPEN, WM_SHOWWINDOW, WM_GETMINMAXINFO, WM_WINDOWPOSCHANGING, WM_WINDOWPOSCHANGED, WM_NCCALCSIZE, WM_SIZING, WM_MOVING, WM_ENTERSIZEMOVE, WM_EXITSIZEMOVE
Commands and notifications	WM_MEASUREITEM, WM_COMMNOTIFY, WM_NOTIFY,WM_NOTIFYFORMAT, WM_STYLECHANGING, WM_STYLECHANGED, WM_COMMAND, WM_SYSCOMMAND, WM_ENTERIDLE, WM_PARENTNOTIFY, WM_MDIRESTORE
System	WM_SYSCOLORCHANGE, WM_WININICHANGE, WM_DEVMODECHANGE, WM_ACTIVATEAPP, WM_FONTCHANGE, WM_TIMECHANGE, WM_COMPACTING, WM_POWER, WM_USERCHANGED, WM_DISPLAYCHANGE, WM_NCACTIVATE, WM_POWERBROADCAST, WM_DEVICECHANGE, WM_PALETTEISCHANGING, WM_PALETTECHANGED
Drawing	WM_SETREDRAW, WM_PAINT, WM_ERASEBKGND, WM_PAINTICON, WM_ICONERASEBKGND, WM_DRAWITEM, WM_NCPAINT, WM_QUERYNEWPALETTE, WM_PRINT, WM_PRINTCLIENT
Scrolling	WM_HSCROLL, WM_VSCROLL, WM_CTLCOLORSCROLLBAR
Icon	WM_QUERYOPEN, WM_QUERYDRAGICON, WM_GETICON, WM_SETICON
MDI	WM_MDICREATE, WM_MDIDESTROY, WM_MDIACTIVATE, WM_MDIRESTORE, WM_MDINEXT, WM_MDIMAXIMIZE, WM_MDITILE, WM_MDICASCADE, WM_MDIICONARRANGE, WM_MDIGETACTIVE, WM_MDISETMENU
Dialog	WM_CANCELMODE, WM_NEXTDLGCTL, WM_MEASUREITEM, WM_DELETEITEM, WM_GETDLGCODE, WM_CTLCOLORMSGBOX, WM_CTLCOLORDLG
Menu	WM_MEASUREITEM, WM_HELP, WM_CONTEXTMENU, WM_INITMENU, WM_INITMENUPOPUP, WM_MENUSELECT, WM_MENUCHAR, WM_ENTERMENULOOP, WM_EXITMENULOOP, WM_NEXTMENUWM_MDIREFRESHMENU

Category	Messages
Text	WM_SETTEXT, WM_GETTEXT, WM_GETTEXTLENGTH, WM_SETFONT, WM_GETFONT
Mouse	WM_SETCURSOR, WM_MOUSEACTIVATE, WM_NCHITTEST, WM_NCMOUSEMOVE, WM_NCLBUTTONDOWN, WM_NCLBUTTONUP, WM_NCLBUTTONDBLCLK, WM_NCRBUTTONDOWN, WM_NCRBUTTONUP, WM_NCRBUTTONDBLCLK, WM_NCMBUTTONDOWN, WM_NCMBUTTONUP, WM_NCMBUTTONDBLCLK, WM_MOUSEMOVE, WM_LBUTTONDOWN, WM_LBUTTONUP, WM_LBUTTONDBLCLK, WM_RBUTTONDOWN, WM_RBUTTONUP, WM_RBUTTONDBLCLK, WM_MBUTTONDOWN, WM_MBUTTONUP, WM_MBUTTONDBLCLK, WM_MOUSEWHEEL, WM_XBUTTONDOWN, WM_XBUTTONUP, WM_XBUTTONDBLCLK, WM_CAPTURECHANGED
Keyboard	WM_VKEYTOITEM, WM_CHARTOITEM, WM_SETHOTKEY, WM_GETHOTKEY, WM_KEYDOWN, WM_KEYUP, WM_CHAR, WM_DEADCHAR, WM_SYSKEYDOWN, WM_SYSKEYUP, WM_SYSCHAR, WM_SYSDEADCHAR, WM_HOTKEY
Clipboard	WM_CUT, WM_COPY, WM_PASTE, WM_CLEAR, WM_UNDO, WM_RENDERFORMAT, WM_RENDERALLFORMATS, WM_DESTROYCLIPBOARD, WM_DRAWCLIPBOARD, WM_PAINTCLIPBOARD, WM_VSCROLLCLIPBOARD, WM_SIZECLIPBOARD, WM_ASKCBFORMATNAME, WM_CHANGECBCHAIN, WM_HSCROLLCLIPBOARD
Edit control	All EM_xxx messages
Static control	All STM_xxx messages
Button	All BM_xxx messages, WM_CTLCOLORBTN
Combo box	All CB_xxx messages, WM_COMPAREITEM
List box	All LB_xxx messages, WM_COMPAREITEM, WM_CTLCOLORLISTBOX
IME	All WM_IME_xxx messages
User-defined	All messages equal or above WM_USER

جدول (۵-۲)

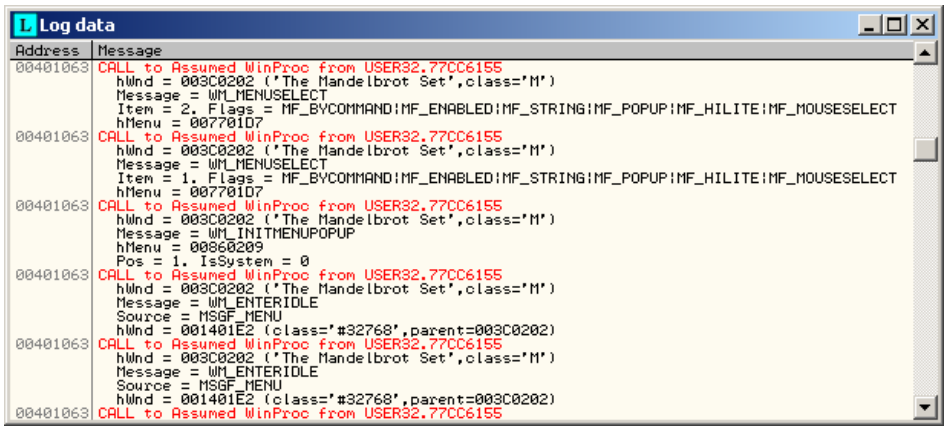
Break

در گزینه‌های این قسمت تنظیماتی برای حوزه عملکرد نقطهٔ توقف در نظر گرفته می‌شود. تنظیمات قسمت بعد دقیقاً مشابه تنظیمات بررسی شده در نقاط شرطی قسمت قبل هستند. با تأیید این تنظیمات، OllyDbg شرط‌های مورد نیاز را باتوجه به تنظیمات تعیین شده ایجاد کرده و یک نقطهٔ توقف شرطی همراه با گزارش‌گیری در آدرس شروع پروسیجر پنجرهٔ موردنظر تعریف می‌کند. از این مرحله به بعد این نقطهٔ توقف دقیقاً همانند نقاط توقف شرطی عمل می‌کند. به‌منظور ایجاد تغییرات در این نقاط توقف ابتدا آدرس نقطهٔ موردنظر را در قسمت Disassembler انتخاب کرده و سپس از کلیدهای Shift+F4 استفاده کنید. در شکل (۵-۲۸) نقطهٔ توقف شرطی ایجاد شده توسط OllyDbg را با استفاده از تنظیمات به کار گرفته شده در شکل (۵-۲۷) مشاهده می‌کنید.



شکل (۵-۲۸)

با اجرا شدن فایل اجرایی، OllyDbg گزارش دقیقی از پیغام‌های موردنظر شما را ضبط کرده و در پنجرهٔ Log Data نمایش خواهد داد. در شکل (۵-۲۹) گزارش‌های تهیه شده پس از ایجاد نقطهٔ توقف را مشاهده می‌کنید.



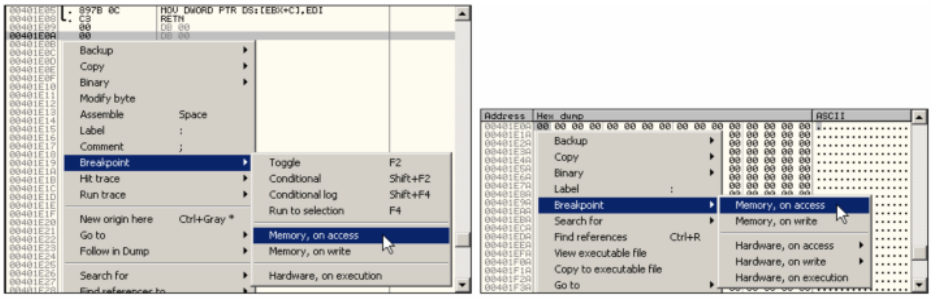
شکل (۵-۲۹)

در صورت نیاز به بررسی دقیق‌تر پروسیجرهای پنجره، نحوهٔ عملکرد و پارامترهای مورد استفادهٔ آنها می‌توانید به فصل ۸ مراجعه کنید.

۵- نقاط توقف برای دسترسی‌ها به حافظه

از این‌گونه نقاط توقف معمولاً به منظور ردیابی عملیات خواندن و نوشتن در محدودهٔ خاصی از فضای حافظهٔ برنامه استفاده می‌شود. OllyDbg اجازهٔ تعریف یک نمونه از این نقاط توقف را در کل برنامه می‌دهد. به علت محدودیت‌های سخت‌افزاری اندازهٔ این محدوده 4096 بایت و ثابت است. به علت مدیریت نرم‌افزاری درنظرگرفته شده برای این نقاط توقف، معمولاً استفاده از آنها در محدوده‌هایی با دسترسی زیاد می‌تواند کارآیی سیستم و سرعت اجرا را به شدت تحت‌تأثیر قرار دهد. استفاده از این‌گونه نقاط توقف در سیستم‌های عامل Windows 95/98 با اختلالات زیادی همراه است. عدم توانایی کاربر در تعیین محدوده موردنظر نیز معمولاً با مشکلات زیادی همراه بوده و باعث توقف‌های غیردلخواه در محدودهٔ 4096 بایتی می‌گردد. در قسمت بعد نقاط توقف سخت‌افزاری معرفی خواهد شد که می‌توانند جایگزینی دقیق و مناسب برای این‌گونه نقاط توقف باشند.

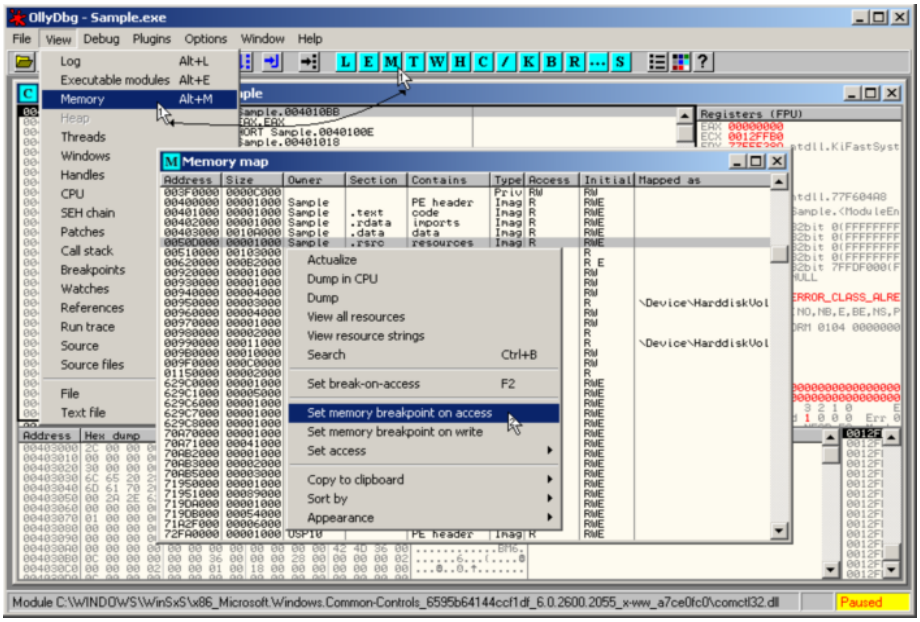
به‌منظور ایجاد این نقاط توقف ابتدا آدرس شروع موردنظر را در قسمت Disassembler و یا Dump انتخاب کرده و سپس از قسمت Breakpoints از منوی صفحه، گزینهٔ Memory on Access و یا Memory on Write را انتخاب کنید.



شکل (۵-۳۰)

نوع دیگری از اینگونه نقاط توقف توانایی پشتیبانی از یک بلوک از حافظهٔ برنامه را دارند. این بلوک‌ها معمولاً Section های فایل‌های اجرایی و یا dllهای بارگذاری شده به فضای حافظهٔ برنامه هستند. این نوع از نقاط توقف در ردیابی دسترسی‌ها به منابع فایل‌های اجرایی نقش کلیدی را ایفا می‌کنند.

به‌منظور ایجاد این نوع از نقاط توقف ابتدا گزینهٔ Memory را از منوی View انتخاب کرده و یا از کلیدهای Alt+M استفاده کنید. با این عمل پنجرهٔ Memory map همانند شکل (۵-۳۱) ظاهر شده و لیستی از بلوک‌های موجود در حافظهٔ برنامه را به همراه مشخصات هر یک به نمایش می‌گذارد. در ادامه جزئیات این پنجره و نحوهٔ عملکرد آن را مورد بررسی قرار خواهیم داد.



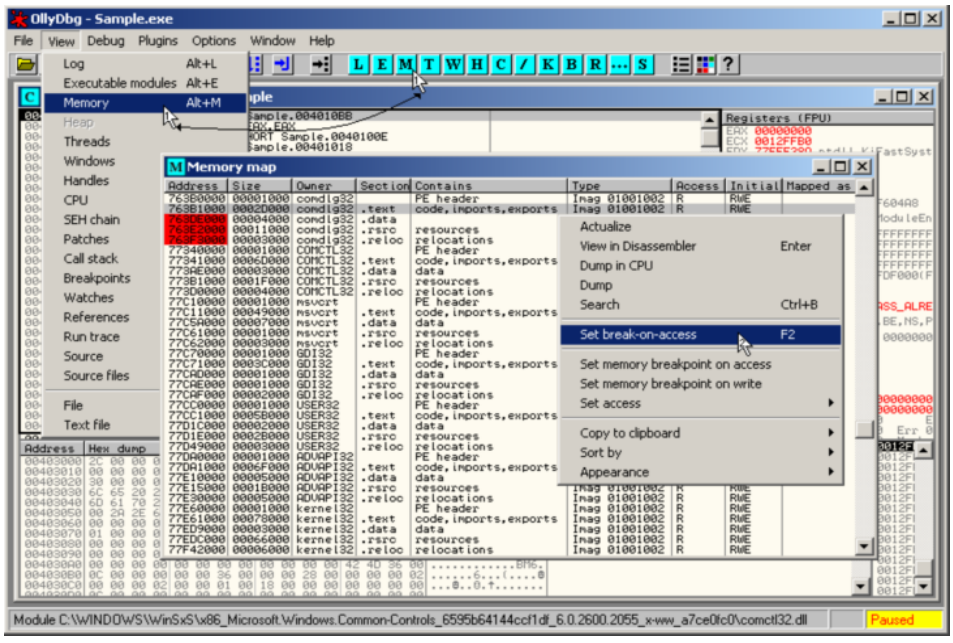
شکل (۵-۳۱)

از لیست نمایش داده شده بلوک مورد نظر را انتخاب کرده و از گزینه‌های Set memory breakpoint on access و یا Set memory breakpoint on write استفاده کنید. توجه داشته باشید که غیر از محدویت اندازهٔ ناحیه، محدودیت‌های این نوع دقیقاً همانند نوع قبل است.

۶- نقاط توقف یکبار مصرف برای بلوک‌های حافظه

عملکرد این نوع از نقاط توقف دقیقاً مشابه نوع قبل است با این تفاوت که پس از یک بار فعال شدن به‌طور خودکار از بین می‌روند. همچنین این نوع محدودیتی برای تعداد ندارند و می‌توانید برخلاف نوع قبلی به هر تعداد از آنها استفاده کنید. توجه داشته باشید که این نوع از نقاط توقف تنها در ویندوزهای خانوادهٔ NT (NT, 2000, XP, 2003) قابل تعریف و استفاده هستند. معمولاً از آنها به‌منظور کنترل فراخوانی‌ها و یا بازگشت‌های انجام شده به dllهای مورداستفادهٔ برنامه استفاده می‌شود.

به‌منظور ایجاد این نوع از نقاط توقف ابتدا گزینهٔ Memory را از منوی View انتخاب کرده و پس از انتخاب بلوک موردنظر از کلید F2 استفاده کنید.



شکل (۵-۳۲)

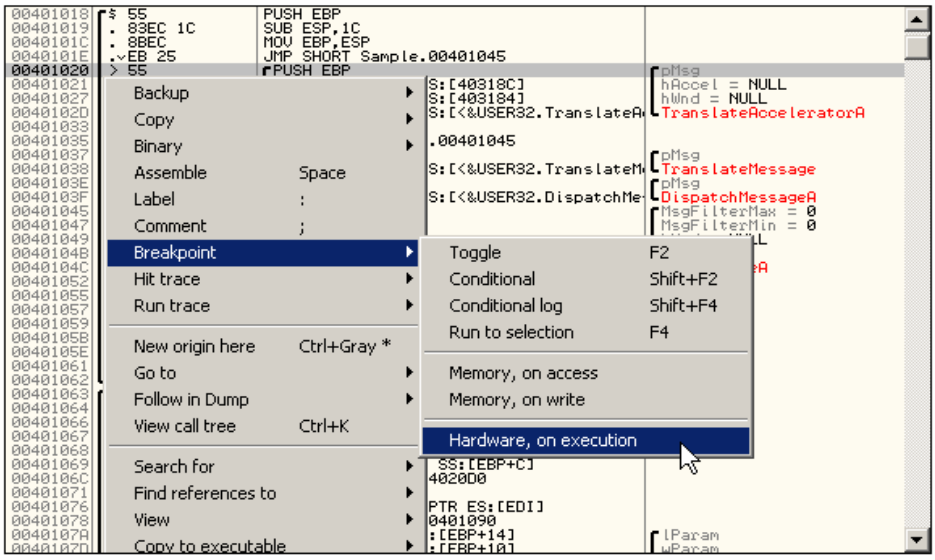
به‌منظور حذف این نقاط توقف عملیات مذکور را تکرار کنید.

۷- نقاط توقف سخت‌افزاری

این نقاط توقف مستقیماً توسط پردازنده‌های 80x86 پشتیبانی می‌شوند و می‌توانند عملکردی مشابه نقاط توقف معمولی (INT3) و یا نقاط توقف حافظه داشته باشند. معمولاً از آنها به‌منظور ردیابی عملیات خواندن، نوشتن و یا اجرای محدودهٔ خاصی از فضای حافظهٔ برنامه استفاده می‌شود. به علت محدودیت‌های سخت‌افزاری، تعداد کل نقاط توقف سخت‌افزاری مورد استفاده به تعداد ٤ عدد و اندازهٔ آنها به محدوده‌های یک بایتی (Byte)، دو بایتی (Word) و چهار بایتی(Dword) محدود می‌شود. باتوجه به پشتیبانی سخت‌افزار، استفاده از آنها باعث کند شدن سرعت اجرای برنامه نمی‌گردد.

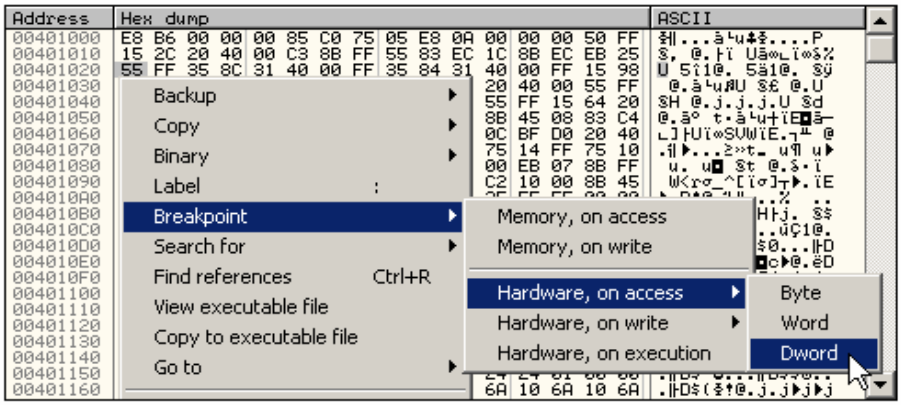
توجه داشته باشید که این‌گونه نقاط توقف تنها توسط سیستم‌های عامل Windows ME, 2000, XP, 2003 و بالاتر پشتیبانی می‌شوند و در نسخه‌های قدیمی‌تر امکان استفاده از آنها وجود ندارد.

برای ایجاد این‌گونه نقاط توقف در قسمت کد، ابتدا آدرس دستورالعمل موردنظر را در قسمت Disassembler انتخاب کرده و سپس همانند شکل (۵-۳۳) از قسمت Breakpoints از منوی Disassembler، گزینهٔ On Execution و Hardware را انتخاب کنید.



شکل (۵-۳۳)

به‌منظور ایجاد توقف برای دسترسی‌ها به حافظه و محدوده‌های data ، ابتدا آدرس موردنظر را در قسمت Dump انتخاب کرده و سپس از قسمت Breakpoint از منوی Dump نقطهٔ توقف دلخواه را انتخاب کنید. همان‌طور که در شکل (۵-۳٤) مشاهده می‌کنید انواع مختلف این‌گونه نقاط توقف عبارتند از: On access، On write و On execute.



شکل (۵-۳٤)

On write : در هنگام نوشتن داده در محدودهٔ موردنظر فعال می‌شود.

On execute : در قسمت‌های کد و درهنگام اجرای دستورالعمل در محدودهٔ موردنظر فعال می‌شود.

On access : با هرگونه دسترسی به محدودهٔ موردنظر فعال می‌شود که می‌تواند خواندن، نوشتن و یا اجرا باشد.

همان‌طور که ذکر شد این‌گونه از نقاط توقف می‌توانند محدوده‌های 1 و 2 و 4 بایتی را تحت پوشش قرار دهند که توسط منوی Dump قابل انتخاب است.

۸- نقاط توقف برای رویدادهای دیباگ

دیباگرها به‌منظور عملکرد صحیح و مطمئن خود باید از برخی از رویدادهای مهم در مراحل اجرا و بارگذاری فایل اجرایی موردنظر مطلع شده و در صورت نیاز کنترل را به دست بگیرند. در زیر برخی از این رویدادها را مورد بررسی قرار خواهیم داد.

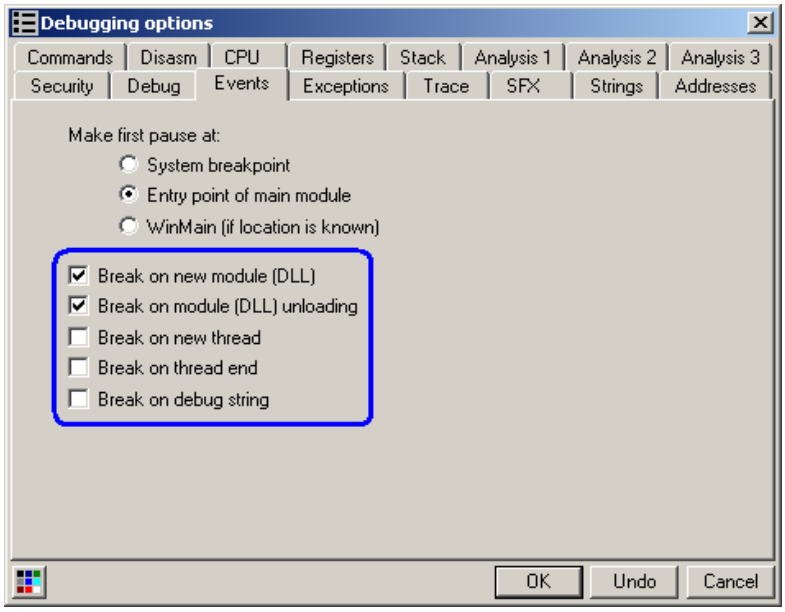
بارگذاری یک فایل dll : درهنگام بارگذاری یک فایل dll جدید به فضای حافظهٔ برنامه رخ می‌دهد.

خارج شدن یک فایل dll : درهنگام خارج شدن یک فایل dll از فضای حافظهٔ برنامه رخ می‌دهد.

ایجاد Thread : درهنگام ایجاد یک Thread جدید در برنامه رخ می‌دهد.

خاتمهٔ Thread : درهنگام خاتمهٔ روند اجرایی یک Thread رخ می‌دهد.

OllyDbg توانایی ایجاد نقاط توقف را برای رویدادهای مذکور درنظر گرفته است که می‌تواند در شرایطی بسیار حساس و مفید باشد. به این منظور گزینهٔ Debugging Options را از منوی Options انتخاب کنید. با این عمل پنجرهٔ Debugging Options همانند شکل (۵-۳۵) ظاهر می‌شود.



شکل (۵-۳۵)

همان‌طور که مشاهده می‌کنید در قسمت Events از این پنجره گزینه‌هایی به‌منظور ایجاد توقف برای رویدادهای دیباگ وجود دارد. شما می‌توانید باتوجه به نیاز خود هر کدام از آنها را انتخاب کنید.

پس از تعیین تنظیمات فوق، با رخ دادن رویداد موردنظر روند اجرایی برنامه متوقف شده و کنترل به OllyDbg منتقل می‌شود. در این مرحله شما می‌توانید بررسی‌ها و یا تغییرات مورد نظر را انجام داده و در صورت نیاز روند اجرایی را دنبال کنید.

بررسی کد

همان‌طور که خواهید دید، OllyDbg امکانات وسیعی را به‌منظور تسهیل روند بررسی کدهای اسمبلی، حافظهٔ برنامه، Threadها، پنجره‌ها و.... تدارک دیده است که می‌تواند به این‌گونه بررسی‌ها سرعت بیشتری دهد. در ادامه به بررسی این امکانات خواهیم پرداخت.

گزینه‌های حرکت در Disassembler

این امکانات و گزینه‌ها، روند حرکت کردن و بررسی آدرس‌ها را در پنجرهٔ Disassembler تسهیل می‌کنند که در ادامه به آنها اشاره خواهیم کرد.

رفتن به آدرس مشخص

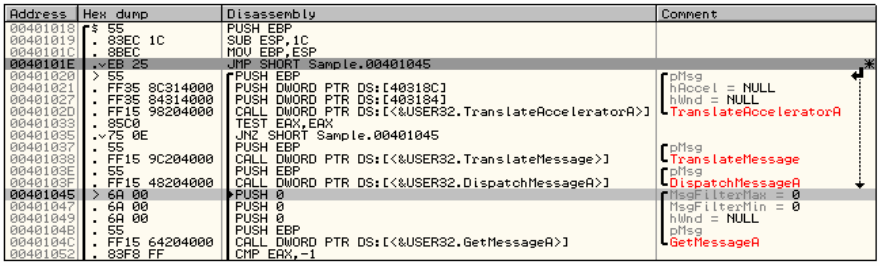
به‌منظور انتقال پنجرهٔ Disassembler به آدرس مجازی موردنظر در فضای حافظهٔ برنامه می‌توانید از کلیدهای Ctrl+G استفاده کنید. با این عمل پنجرهٔ Enter Expression همانند شکل (۵-۳٦) نمایش داده شده و می‌توانید آدرس موردنظر را در آن وارد کنید.

شکل (۵-۳٦)

با تأیید این پنجره، به آدرس مجازی موردنظر منتقل خواهید شد.

دنبال کردن آدرس‌های استفاده شده در پرش‌ها و فراخوانی‌ها

به‌منظور دنبال کردن آدرس‌های استفاده شده در دستورالعمل‌های پرش و فراخوانی‌ها، ابتدا دستورالعمل مذکور را انتخاب کرده و سپس کلید Enter را فشار دهید. با این عمل همانند شکل (۵-۳۷) به آدرس مقصد دستور پرش و یا فراخوانی موردنظر منتقل خواهید شد.

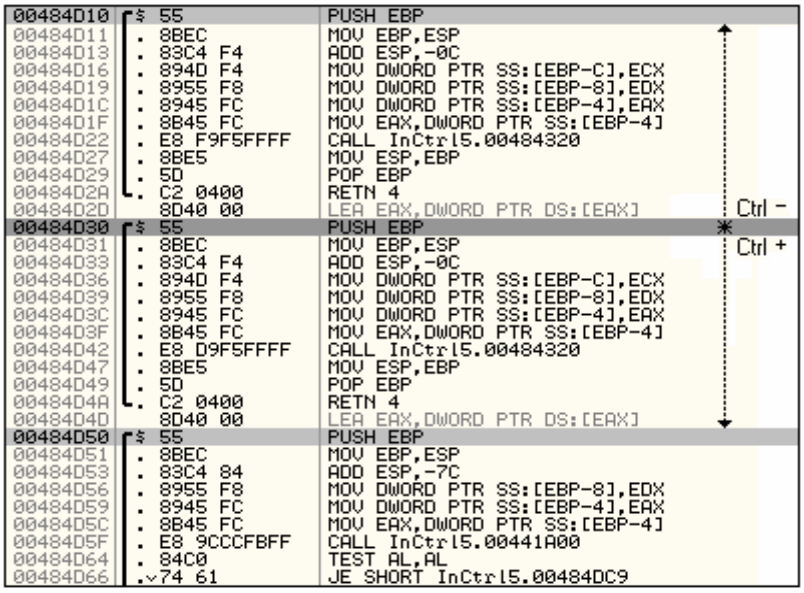


شکل (۵-۳۷)

در هنگام دنبال کردن آدرس‌ها و فراخوانی‌ها و یا حرکت در قسمت Disassembler می‌توانید با استفاده از کلید + و - به آدرس بعدی و قبلی منتقل شوید. همچنین به‌منظور انتقال به آدرس فعلی که توسط ثبات EIP مشخص شده است کلید * را به کار بگیرید.

حرکت براساس آدرس شروع توابع

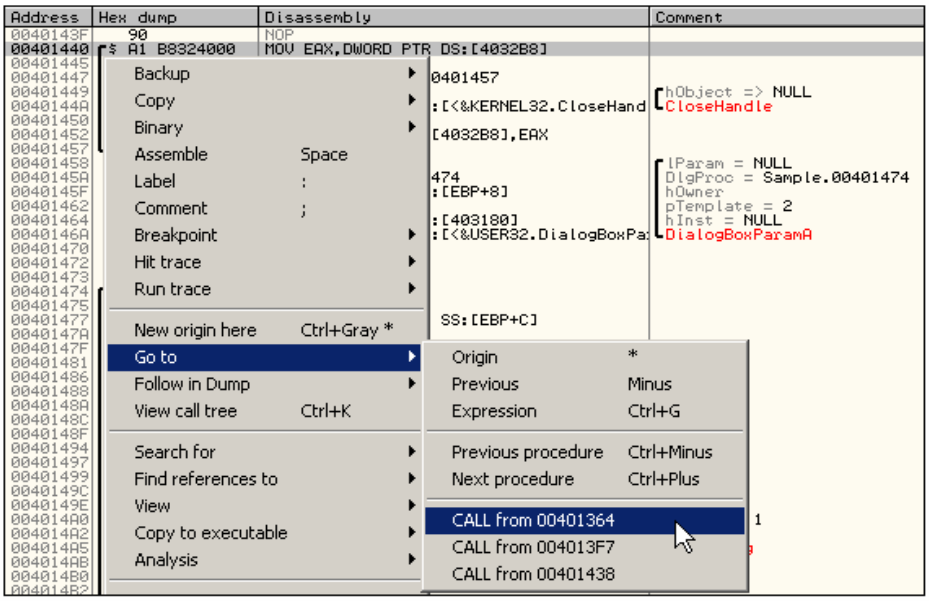
در صورت نیاز می‌توانید براساس آدرس شروع توابع در قسمت Disassembler حرکت کنید. این امر باعث تسریع در دنبال کردن توابع موجود در فایل اجرایی موردنظر می‌گردد. کلیدهای "+" + Ctrl و "-" + Ctrl به ترتیب شما را به آدرس شروع تابع بعد و قبلی در پنجرهٔ Disassembler منتقل خواهند کرد.



شکل (۵-۳۸)

دنبال کردن سریع ارجاع‌ها، پرش‌ها و فراخوانی‌ها

به‌منظور دنبال کردن پرش‌ها و یا فراخوانی‌ها به آدرس موردنظر، ابتدا آدرس مذکور را انتخاب کرده و سپس به قسمت Go to از منوی Disassembler مراجعه کنید. همان‌طور که در شکل(۵-۳۹) مشاهده می‌کنید، در بخش آخر زیر منوی Go to لیستی از ارجاع‌های انجام شده به نمایش گذاشته شده است که می‌توانید با انتخاب آنها، به آدرس فراخوانی و یا پرش موردنظر منتقل شوید.

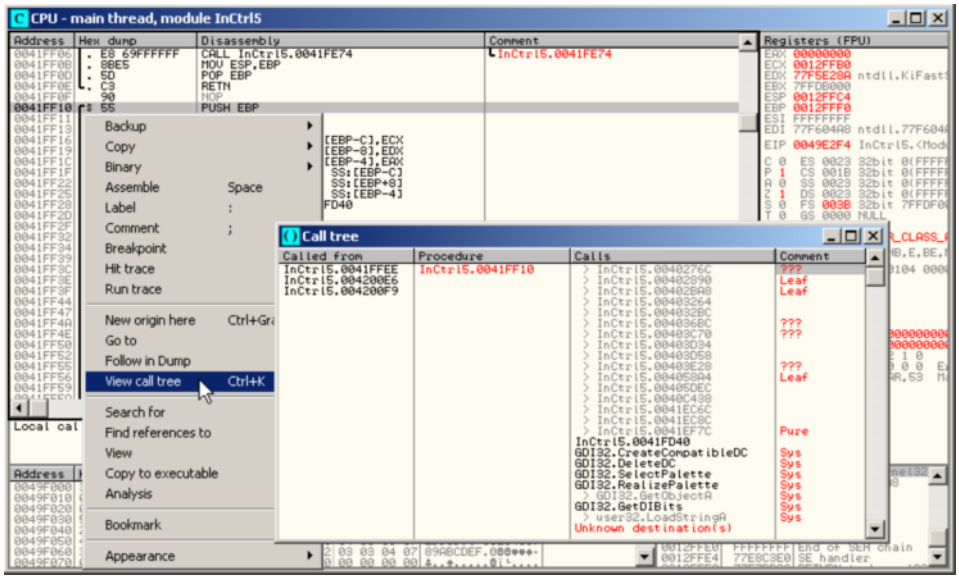


شکل (۵-۳۹)

بررسی سلسله مراتبی فراخوانی‌های انجام شده

OllyDbg امکانات بسیار مفیدی را به‌منظور دنبال کردن فراخوانی‌های انجام شده از یک تابع و نیز فراخوانی‌های مستقیم و یا غیرمستقیم انجام شده توسط آن تابع را تدارک دیده است که می‌تواند به‌منظور بررسی سلسله مراتبی فراخوانی‌ها و ارجاع‌ها به توابع به کار گرفته شود. در عمل از این امکانات به‌طور بسیار گسترده‌ای به‌منظور به دام انداختن فراخوانی‌ها API، دسترسی‌ها به توابع و نیز بررسی توابع وابسته در عملیات استخراج توابع و کد استفاده می‌شود.

به‌منظور مشاهدهٔ سلسله مراتبی فراخوانی‌های انجام شده از تابع موردنظر و نیز فراخوانی‌های انجام شده به آن تابع، ابتدا آدرس شروع آن را در قسمت Disassembler انتخاب کرده و سپس از کلیدهای Ctrl+K استفاده کنید. با این عمل پنجرهٔ Call tree همانند شکل (۵-۴۰) ظاهر می‌شود.

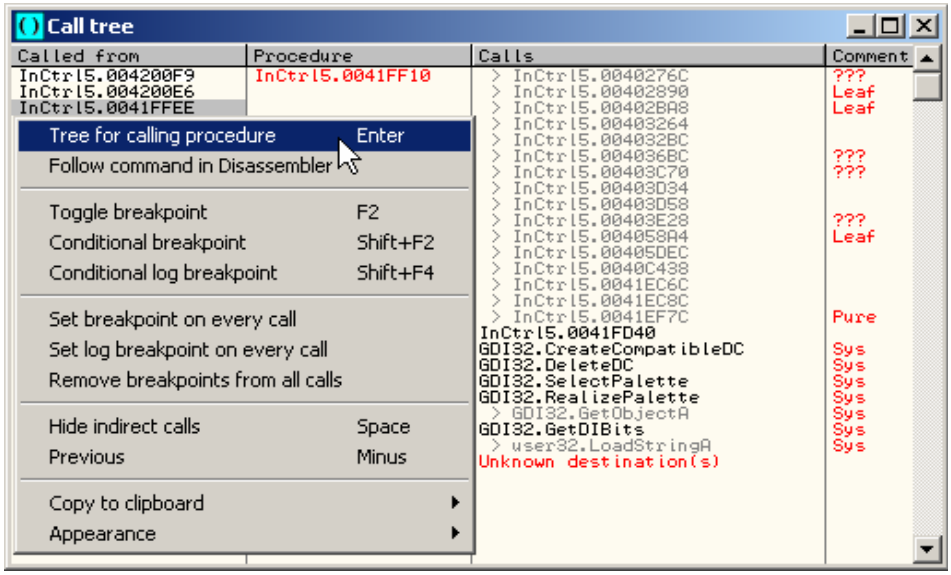


شکل (٥-٤٠)

همان‌طور که مشاهده می‌کنید این پنجره به چهار ستون تقسیم شده است که هر کدام اطلاعاتی خاصی را به نمایش می‌گذارند.

Called from

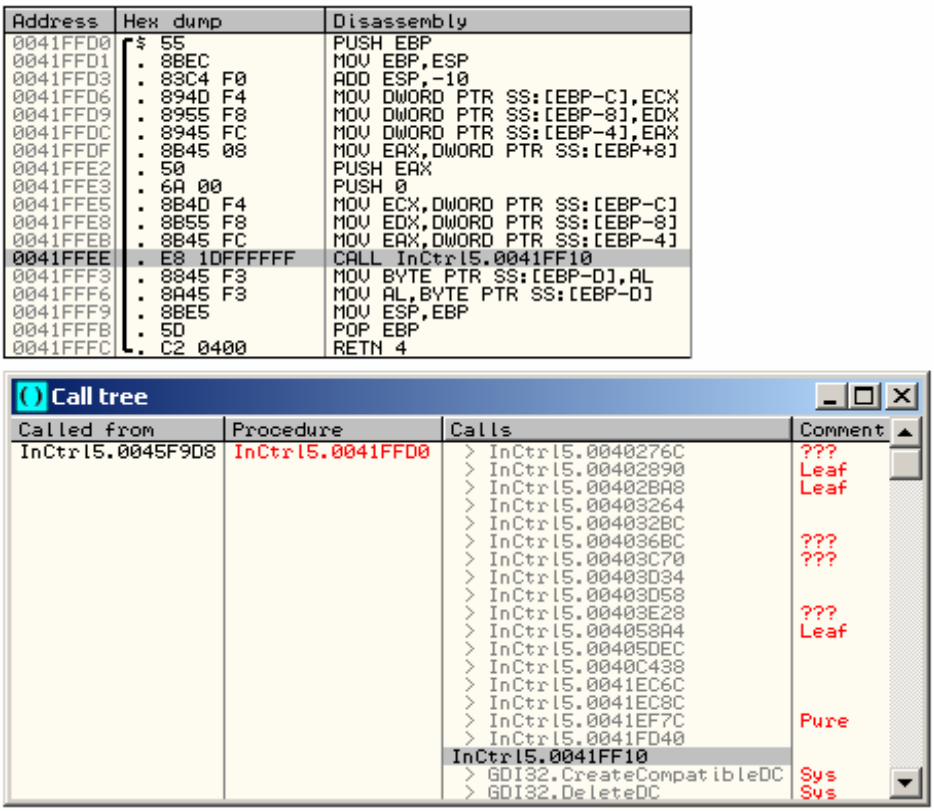
در این ستون آدرس‌های فراخوانی‌های انجام شده از تابع موردنظر نمایش داده می‌شود. همان‌طور که در شکل (٥-٤١) مشاهده می‌کنید، برای اعضاء این ستون گزینه‌های مختلفی درنظر گرفته شده است که برخی از آنها را مورد بررسی قرار می‌دهیم.



شکل (٥-٤١)

Call tree for calling procedure

با انتخاب فراخوانی موردنظر و استفاده از این گزینه، پنجرهٔ Call tree لیست سلسله مراتبی توابع فراخوانی‌کننده را نمایش خواهد داد. در مثال بالا با دنبال کردن فراخوانی انجام شده در آدرس 41FFEE، پنجرهٔ Call tree همانند شکل (٥-٤٢) لیست سلسله مراتبی تابع 41FFD0 را نمایش خواهد داد.



Address	Hex dump	Disassembly
0041FFD0	$ 55	PUSH EBP
0041FFD1	. 8BEC	MOV EBP,ESP
0041FFD3	. 83C4 F0	ADD ESP,-10
0041FFD6	. 894D F4	MOV DWORD PTR SS:[EBP-C],ECX
0041FFD9	. 8955 F8	MOV DWORD PTR SS:[EBP-8],EDX
0041FFDC	. 8945 FC	MOV DWORD PTR SS:[EBP-4],EAX
0041FFDF	. 8B45 08	MOV EAX,DWORD PTR SS:[EBP+8]
0041FFE2	. 50	PUSH EAX
0041FFE3	. 6A 00	PUSH 0
0041FFE5	. 8B4D F4	MOV ECX,DWORD PTR SS:[EBP-C]
0041FFE8	. 8B55 F8	MOV EDX,DWORD PTR SS:[EBP-8]
0041FFEB	. 8B45 FC	MOV EAX,DWORD PTR SS:[EBP-4]
0041FFEE	. E8 1DFFFFFF	CALL InCtrl5.0041FF10
0041FFF3	. 8845 F3	MOV BYTE PTR SS:[EBP-D],AL
0041FFF6	. 8A45 F3	MOV AL,BYTE PTR SS:[EBP-D]
0041FFF9	. 8BE5	MOV ESP,EBP
0041FFFB	. 5D	POP EBP
0041FFFC	. C2 0400	RETN 4

Call tree

Called from	Procedure	Calls	Comment
InCtrl5.0045F9D8	InCtrl5.0041FFD0	> InCtrl5.0040276C	???
		> InCtrl5.00402890	Leaf
		> InCtrl5.00402BA8	Leaf
		> InCtrl5.00403264	
		> InCtrl5.004032BC	
		> InCtrl5.004036BC	???
		> InCtrl5.00403C70	???
		> InCtrl5.00403D34	
		> InCtrl5.00403D58	
		> InCtrl5.00403E28	???
		> InCtrl5.004058A4	Leaf
		> InCtrl5.00405DEC	
		> InCtrl5.0040C438	
		> InCtrl5.0041EC6C	
		> InCtrl5.0041EC8C	
		> InCtrl5.0041EF7C	Pure
		> InCtrl5.0041FD40	
		InCtrl5.0041FF10	
		> GDI32.CreateCompatibleDC	Sys
		> GDI32.DeleteDC	Sys

شکل (۵-۴۲)

همان‌طور که در قسمت Disassembler مشاهده می‌کنید در آدرس 41FFEE از این تابع، فراخوانی موردنظر صورت گرفته است. توجه داشته باشید در بررسی‌های تودرتوی توابع می‌توانید با استفاده از کلیدهای + و - به مرحلهٔ بعدی و یا قبلی منتقل شوید.

Follow Command in Disassembler

با انتخاب این گزینه، آدرس موردنظر در قسمت Disassembler نمایش داده خواهد شد.

نقاط توقف

همان‌طور که در مثال‌ها مشاهده کردید، در این قسمت گزینه‌های متعددی به‌منظور ایجاد نقاط توقف مختلف در آدرس‌های فراخوانی‌ها وجود دارد. برخی از آنها نیز توانایی تعریف نقاط توقف را برای کلیهٔ فراخوانی‌های موجود در این ستون دارند. در مراحل تحلیل و بررسی نحوهٔ عملکرد توابع، از گزینه‌های موجود در این قسمت استفاده‌های فراوانی می‌شود.

Procedure

این ستون حاوی نام (آدرس) تابع موردنظر است.

Calls

در این ستون لیست توابعی که به‌طور مستقیم و یا غیرمستقیم توسط این تابع فراخوانی شده‌اند نمایش داده می‌شود. همان‌طور که مشاهده کردید این توابع شامل توابع داخلی فایل اجرایی، توابع API و یا فراخوانی‌های نامعلوم مانند Call EBX هستند.

همانند ستون Called From، با Double Click کردن برروی تابع موردنظر، نمودار سلسله مراتبی آن در پنجرهٔ Call tree نمایش داده خواهد شد و به همین صورت می‌توانید فراخوانی‌های صورت گرفته را دنبال کنید.

توجه داشته باشید که نام فراخوانی‌های غیرمستقیم با کاراکتر " > " آغاز می‌شود که دارای رنگ خاکستری هستند. در صورت نیاز می‌توانید با استفاده از کلید Space از نمایش آنها جلوگیری کنید.

Comment

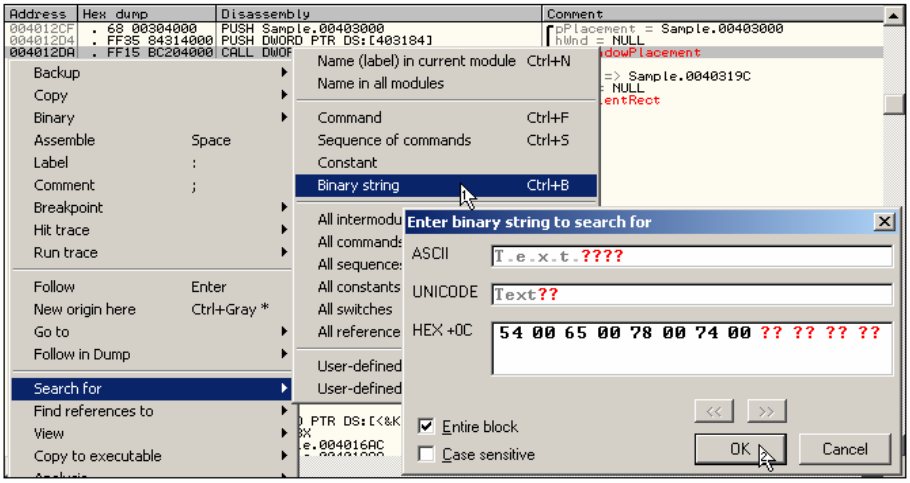
در ستون Comment اطلاعات مفیدی راجع به نوع تابع متناظر در لیست Calls از نظر وابستگی نمایش داده می‌شود. انواع Pure و Leaf در سلسله مراتب فراخوانی‌ها در آخرین سطح قرار داشته و هیچ تابع دیگری را فراخوانی نمی‌کنند و در حقیقت به عنوان برگ‌های درخت فراخوانی‌ها محسوب می‌شوند. فراخوانی‌های انجام شده از توابع موجود در dllهای سیستمی(dllهای موجود در دایرکتوری System ویندوز) از نوع SYS هستند. در صورتی که تابع، یک تابع معمولی بوده و در هیچ کدام از شرایط فوق صدق نکند، توضیحی برای آن در این ستون وجود نخواهد داشت. توجه داشته باشید که تنها توابعی که ابتدا و انتهای آنها شناسایی شده باشد مورد این بررسی‌ها قرار می‌گیرند. در غیراین‌صورت از رشتهٔ ؟؟؟ به‌عنوان توضیحات در این ستون استفاده خواهد شد.

گزینه‌های جستجو

در اکثر پنجره‌های موجود در OllyDbg که امکان نمایش آیتم‌های متعددی را دارند از جمله Disassembler، گزینه‌های مختلفی به منظور ایجاد قابلیت جستجو براساس موارد دلخواه در نظر گرفته شده است. بدیهی است که استفاده از این قابلیت‌های جستجو می‌تواند از صرف وقت و ایجاد سردرگمی‌در هنگام بررسی فایل اجرایی جلوگیری کند.در ادامه برخی از مهمترین این گزینه‌ها را مورد بررسی قرار خواهیم داد.

جستجوی رشته‌ها

OllyDbg امکان جستجوی یک رشته دلخواه را در فایل اجرایی دارد. این رشته می‌تواند با فرمت‌های ASCII، UNICODE و یا HEX باشد. به این منظور می توانید از کلیدهای Ctrl + B استفاده کنید. با این عمل پنجره Search همانند شکل (۵-۴۳) نمایش داده می‌شود.

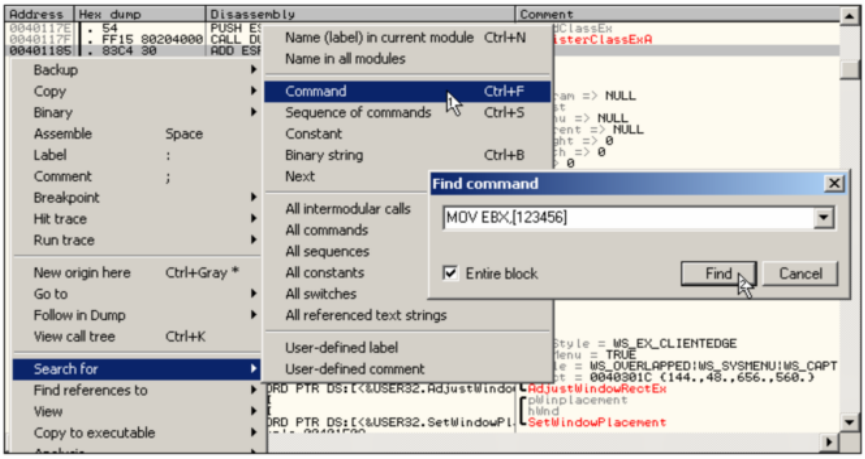


شکل (۵-۴۳)

همان‌طور که مشاهده می کنید در این پنجره امکان وارد کردن رشته براساس هر یک از سه فرمت استاندارد وجود دارد. علاوه بر این در قسمت Hex می‌توانید الگوهای موردنظر را برای کاراکترهای نامعلوم با استفاده از کاراکتر "؟" تعریف کنید. بدیهی است که این جستجو برروی section (block) فعلی صورت می‌گیرد. با استفاده از کلیدهای Ctrl + L می‌توانید عملیات جستجو را پس از پیدا شدن هر رشته به منظور یافتن رشته‌های دیگر ادامه دهید.

جستجو براساس یک دستورالعمل

در صورت نیاز می‌توانید جستجوهایی را براساس دستورالعمل موردنظر انجام دهید. به‌منظور جستجوی یک دستورالعمل از کلیدهای Ctrl + F استفاده کنید. با این عمل پنجرهٔ Find Command همانند شکل (۵-۴۴) نمایش داده شده و امکان تعیین دستورالعمل موردنظر را به کاربر می‌دهد.



شکل (٥-٤٤)

به‌منظور ایجاد قابلیت انعطاف بیشتر در عملیات جستجوی دستورالعمل‌ها، OllyDbg امکان تعریف غیرصریح دستورالعمل را در مراحل جستجو تدارک دیده است. به این منظور می‌توانید در دستورالعمل‌های موردنظر خود از کلمات کلیدی و یا دستورات غیرصریح مطابق جدول (٥-٣) استفاده کنید.

	Keyword	Matches
Keywords	R8	Any 8-bit register (AL,BL, CL, DL, AH, BH, CH, DH)
	R16	Any 16-bit register (AX, BX, CX, DX, SP, BP, SI, DI)
	R32	Any 32-bit register (EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI)
	FPU	Any FPU register (ST0..ST7)
	MMX	Any MMX register (MM0..MM7)
	CRX	Any control register (CR0..CR7)
	DRX	Any debug register (DR0..DR7)
	CONST	Any constant
	OFFSET	Same as CONST
	Command	**Matches**
Imprecise Commands	JCC	Any conditional jump (JE, JC, JNGE...)
	SETCC	Any conditional set byte (SETE, SETC, SETNGE...)
	CMOVCC	Any conditional move (CMOVE, CMOVC, CMOVNGE...)

جدول (٥-٣)

به عنوان مثال رشتهٔ جستجوی MOV R32, [CONST] هر دو دستورالعمل MOV ESI , [401045] و MOV EAX , [10000] را تحت پوشش قرار خواهد داد.

جستجو براساس دنباله‌ای از دستورالعمل‌ها

همان‌طور که در فصل ۳ بررسی شد، کامپایلرها از دنباله‌های ثابتی از دستورالعمل‌ها به‌منظور انجام عملیات موردنظر (مانند عملیات Stack، مدیریت توابع و...) استفاده می‌کنند. در نتیجه توانایی بررسی این دنباله‌ها در کدهای Disassemble شده یکی از موارد بسیار مهم و کلیدی در بررسی نحوهٔ عملکرد اجزاء فایل اجرایی محسوب می‌شود. OllyDbg به این منظور سیستم‌های خاصی را برای دنبال کردن آنها با استفاده از الگوهای تعریف شده توسط کاربر تدارک دیده است.

توجه داشته باشید که برخی از دستورالعمل‌ها مانند پرش‌ها، Call، Mov و... باتوجه به نوع عملوند مورد استفاده می‌توانند کدهای ماشین (Opcode) مختلفی را ایجاد کنند. OllyDbg در مراحل جستجو کلیهٔ حالات ممکن را برای هر آیتم، مورد بررسی قرار داده آنها را در تعیین صحت شرط موردنظر اعمال خواهد کرد.

در هنگام تعیین دنبالهٔ دستورالعمل‌های مورد جستجو می‌توانید از کلمات کلیدی و یا دستورات غیرصریح مطابق جدول (۵-۳) استفاده کنید. علاوه بر این به‌منظور ایجاد قابلیت تطابق دقیق‌تر دستورالعمل‌ها باشرایط موردنظر کاربر، دوثبات غیرصریح دیگر نیز درتعریف دنبالهٔ دستورالعمل‌ها می‌توانند مورد استفاده قرار بگیرند که عبارتند از RA و RB . تعریف این دو ثبات غیرصریح دقیقاً مشابه R32 در جدول (۵-۳) است با این تفاوت هر یک از آنها می‌تواند تنها به‌منظور پوشش یک ثبات در طول یک دستورالعمل به کار گرفته شود. به عنوان مثال دستورالعمل غیرصریح LEA RA , [4*RA + RA] می‌تواند هر یک از دو دستورالعمل LEA EAX , [4*EAX+EAX] و یا LEA ESI , [4*ESI+ESI] را تحت پوشش قرار دهد ولی دستورالعمل LEA ESI , [4*EAX+EAX] را پوشش نخواهد داد. توجه داشته باشید که می‌توانید از هر دوی آنها نیز به‌منظور مشخص کردن دو گروه ثبات در دستورالعمل غیرصریح استفاده کنید.

علاوه بر دو ثبات مذکور، دستور غیرصریح ANY n می‌تواند به‌منظور پوشش دادن تعداد n دستورالعمل در دنبالهٔ مورد جستجو به کار گرفته شود. در صورت مشخص نکردن تعداد n، این دستور هر تعداد از دستورالعمل‌ها را شامل خواهد شد. به عنوان نمونه به مثال‌های زیر توجه کنید.

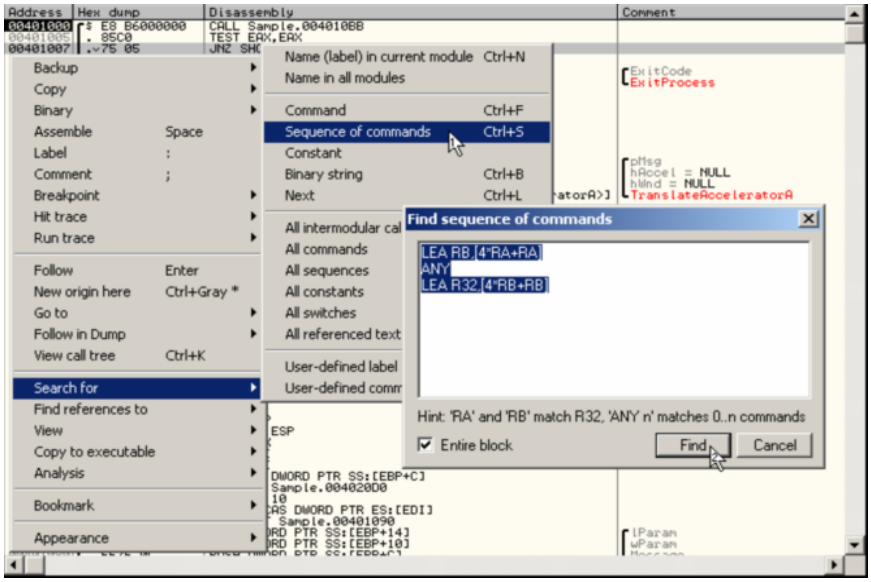
Search Sequence	Sequence 1	Sequence 1	Sequence 1

PUSH EBP
ANY 2
MOV EBP,ESP
	PUSH EBP
	SUB ESP,1C
	PUSH ESP
	PUSH 0
	MOV EBP,ESP
		PUSH EBP
		SUB ESP,1C
		PUSH ESP
		MOV EBP,ESP
			PUSH EBP
			SUB ESP,1C
			MOV EBP,ESP

	✓	✓	✗
LEA RB,[4*RA+RA] **ANY** **LEA R32,[4*RB+RB]**	LEA EAX,[4*EBX+EBX] LEA ESI,[4*ESI+ESI]	LEA EAX,[4*EBX+EBX] PUSH EAX LEA ESI,[4*ESI+ESI]	LEA EAX,[4*EBX+EBX] INC EAX PUSH EAX LEA ESI,[4*ESI+ESI]
	✓	✓	✓

جدول (۵-۴)

به‌منظور انجام اینگونه از جستجوها در قسمت Diassembler از کلیدهای Ctrl + S استفاده کنید. با این عمل پنجرهٔ Find Sequence همانند شکل (۵-٤٥) نمایش داده شده و امکان وارد کردن دنبالهٔ جستجو را ایجاد می‌کند.

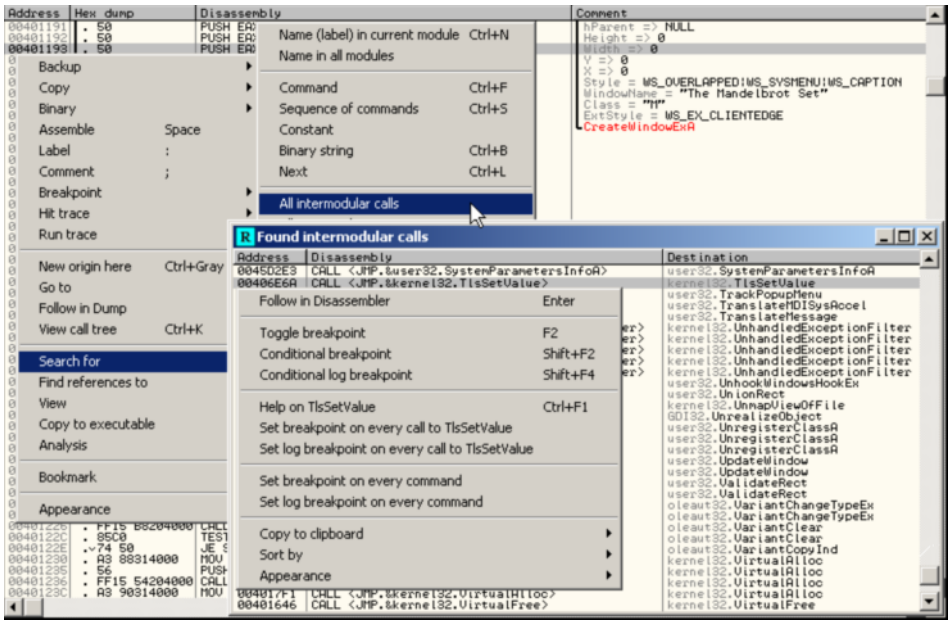


شکل (۵-٤٥)

جستجوی فراخوانی‌های خارجی انجام شده توسط فایل اجرایی

جستجو و بررسی فراخوانی‌های خارجی صورت گرفته توسط فایل اجرایی می‌تواند گامی‌مهم در بررسی اجزاء مختلف فایل اجرایی و نحوهٔ عملکرد هر کدام از آنها باشد. یک نمونه از این فراخوانی‌های خارجی، فراخوانی‌های API هستند که با آنها آشنایی کافی دارید. OllyDbg توانایی جستجو و شناسایی انواع مختلف فراخوانی‌های خارجی از جمله فراخوانی‌های مستقیم، غیرمستقیم، پرش‌های خارجی و ترکیب‌های آنها را برای کاربر درنظر گرفته است. علاوه بر این در مراحل دیباگ فراخوانی‌های انجام شده با استفاده از آدرس خروجی تابع GetProcAddress نیز قابل شناسایی هستند.

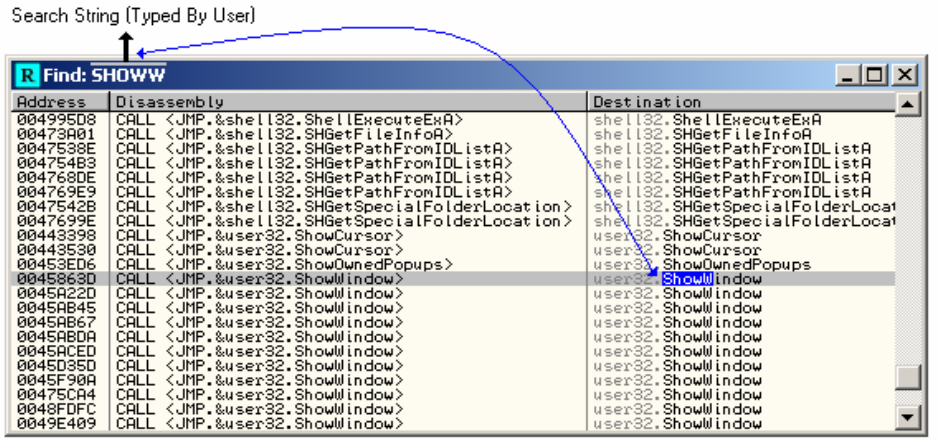
به‌منظور انجام اینگونه جستجوها از گزینهٔ All Intermodular Calls در قسمت Search for از منوی Diassembler استفاده کنید. با انجام این عمل پنجرهٔ Found Intermodular Calls همانند شکل (٥-٤٦) ظاهر می‌شود.



شکل (٥-٤٦)

همان‌طور که مشاهده می‌کنید در این پنجره لیست کاملی از کلیهٔ فراخوانی‌های خارجی انجام شده در فایل اجرایی نمایش داده می‌شود. از منوی موجود در این پنجره نیز می‌توانید به منظور دنبال کردن فراخوانی‌ها، ایجاد نقاط توقف، دریافت اطلاعات در مورد توابع و... استفاده کنید.

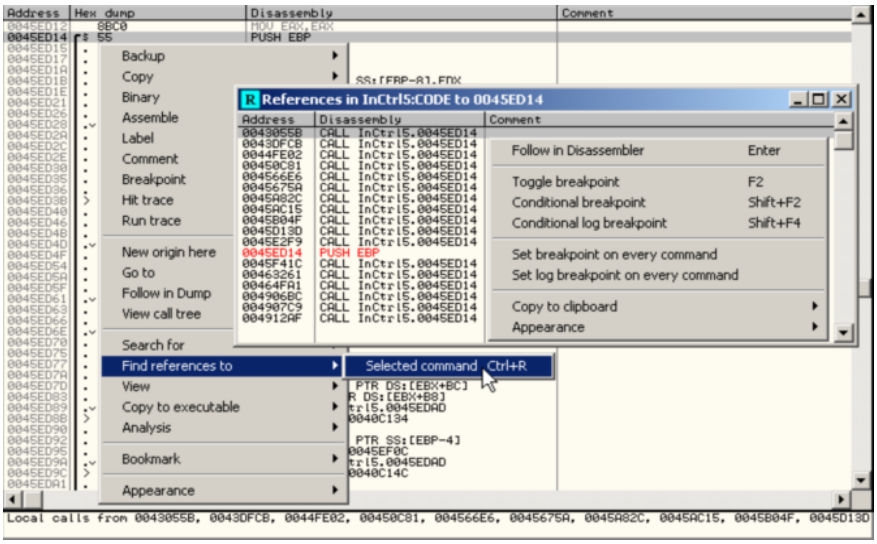
به‌دلیل تعداد زیاد فراخوانی‌های نمایش داده شده در لیست، OllyDbg امکان خاصی را به‌منظور جستجوی لیست براساس نام تابع فراخوانی شده تدارک دیده است. به این منظور در این پنجره نام تابع موردنظر را تایپ کنید. همانطور که درشکل (۵-۴۷) مشاهده می‌کنید، درهنگام تایپ، نزدیک‌ترین نام به رشتهٔ تایپ شده، در لیست انتخاب خواهد شد.



شکل (۵-۴۷)

جستجوی ارجاع‌های انجام شده

در صورت نیاز می‌توانید لیستی از کلیهٔ ارجاع‌های انجام شده به آدرسی مشخص را تهیه کنید. به این منظور ابتدا آدرس مورد نظر را در قسمت Diassembler انتخاب کرده و سپس گزینهٔ Selected Command را از قسمت Find References to از منوی Diassembler انتخاب کنید. با این عمل پنجرهٔ References همانند شکل (۵-۴۸) ظاهر شده و لیستی از کلیهٔ فراخوانی‌های انجام شده به آدرس موردنظر در Section فعلی را به نمایش می‌گذارد.



شکل (۵-۴۸)

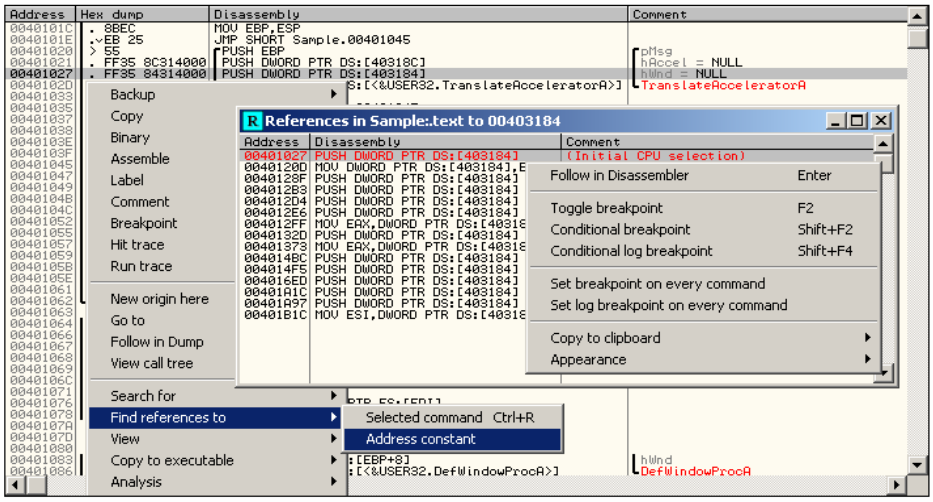
از منوی این پنجره نیز می‌توانید به‌منظور دنبال کردن ارجاع‌ها و ایجاد نقاط توقف استفاده کنید. همان‌طور که در شکل بالا مشاهده می‌کنید، در قسمت Information (قسمت پایین) نیز لیست کلیهٔ این ارجاع‌ها وجود دارد که می‌تواند به‌منظور دنبال کردن سریع‌تر آنها به کار گرفته شود.

در صورتی که دستورالعمل موردنظر از آدرس خاصی استفاده می‌کند به عنوان مثال MOV EAX , [EBP-8] و یا MOV EAX , [401045] می‌توانید لیستی از کلیهٔ ارجاع‌های انجام شده به آدرس معین و یا نامعین مذکور را تهیه کنید.

توجه داشته باشید که ارجاع‌ها به یک آدرس به‌طرق مختلفی قابل انجام هستند. به عنوان مثال دستورالعمل‌های زیر همگی به عنوان ارجاع‌هایی به آدرس 401234 محسوب می‌شوند.

```
MOV EAX,401234
MOV EAX,DWORD PTR [00401234]
MOV BYTE PTR [EBX*4+EDI+00401234],AL
JNE 00401234
CALL 00401234
DD 00401234
```

به این منظور ابتدا دستورالعمل مورد نظر را انتخاب کرده و سپس گزینهٔ Address Constant را از قسمت Find References to از منوی Diassembler انتخاب کنید. با این عمل پنجرهٔ References همانند شکل (۵-۴۹) ظاهر شده و لیستی از ارجاع‌های انجام شده به آدرس مورد استفاده در دستورالعمل مذکور را به نمایش می‌گذارد.



شکل (۵-۴۹)

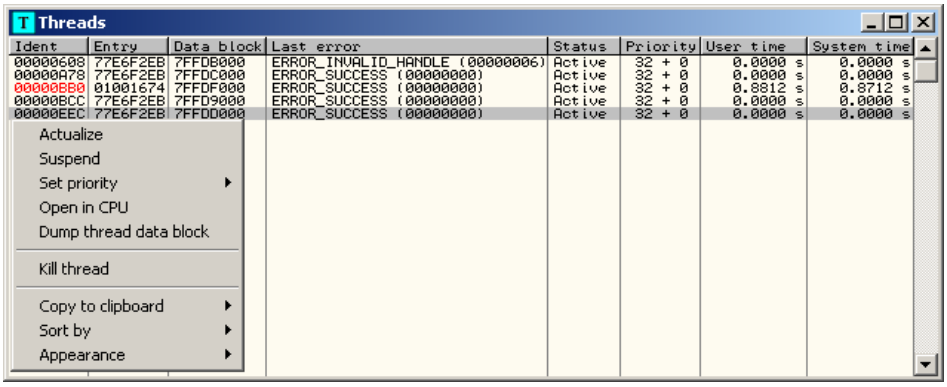
همان‌طور که درمثال بالا مشاهده می‌کنید. لیستی از ارجاع‌های انجام شده به آدرس 403184 نمایش داده شده است.

بررسی Threadها

همان‌طور که می‌دانید Threadها اجزاء موازی یک برنامه هستند که می‌توانند به‌طور همزمان اجرا شوند. معمولاً از آنها به منظور تقسیم وظایف در نرم‌افزارهای بزرگ استفاده می‌شود. بررسی جزئیات Threadهای ایجاد شده توسط یک نرم‌افزار و تغییر خصوصیات و یا توانایی متوقف کردن و یا از بین بردن آنها می‌تواند گامی‌مهم در شناسایی اجزاء تشکیل دهندهٔ یک نرم‌افزار و وظیفهٔ هر کدام از آنها باشد. به منظور بررسی کامل‌تر Threadها و نحوهٔ عملکرد و به کار گیری آنها می‌توانید به فصل ۸ مراجعه کنید.

به‌منظور ایجاد قابلیت دیباگ در مراحل اجرای خط به خط، Trace و Hit، OllyDbg با متوقف کردن موقت سایر Threadها (غیر از Thread فعلی) امکان دیباگ کردن برنامه‌های MultiThread را ایجاد می‌کند.

به‌منظور مشاهدهٔ Threadهای ایجاد شده توسط فایل اجرایی در مراحل دیباگ و ایجاد تغییرات در آنها می‌توانید گزینهٔ Threads را از منوی View انتخاب کنید. با این عمل پنجرهٔ Threadsها همانند شکل (۵-۵۰) نمایش داده می‌شود.



شکل (۵-۵۰)

همان‌طور که در مثال بالا مشاهده می‌کنید برای هر Thread اطلاعات خاصی نمایش داده شده است که برخی از آنها را در ادامه مورد بررسی قرار خواهیم داد:

Ident : شناسهٔ Thread که توسط تابع Create Thread تعیین شده است.

Entry : آدرس شروع تابع Thread در فضای حافظهٔ برنامه.

Last Error : آخرین خطای رخ داده در Thread که توسط تابع GetLastError تعیین می‌شود.

Status : وضعیت اجرایی فعلی Thread را نمایش می‌دهد که می‌تواند یکی از مقادیر زیر باشد.

Active : Thread فعال و در حال اجرا است.

Suspended : Thread در حالت تعلیق است.

Traced : Thread در حالت تعلیق است ولی قبلاً توسط OllyDbg ، Trace شده است.

Paused : Thread فعال است ولی به دلیل بررسی و یا دیباگ کردن Threadهای دیگر، OllyDbg به‌طور موقت آن را به حالت تعلیق درآورده است. توجه داشته باشید که درغیر این‌صورت امکان دیباگ کردن و کنترل کامل برنامه‌های MultiThread وجود نخواهد داشت.

Finished : روند اجرایی Thread خاتمه یافته است.

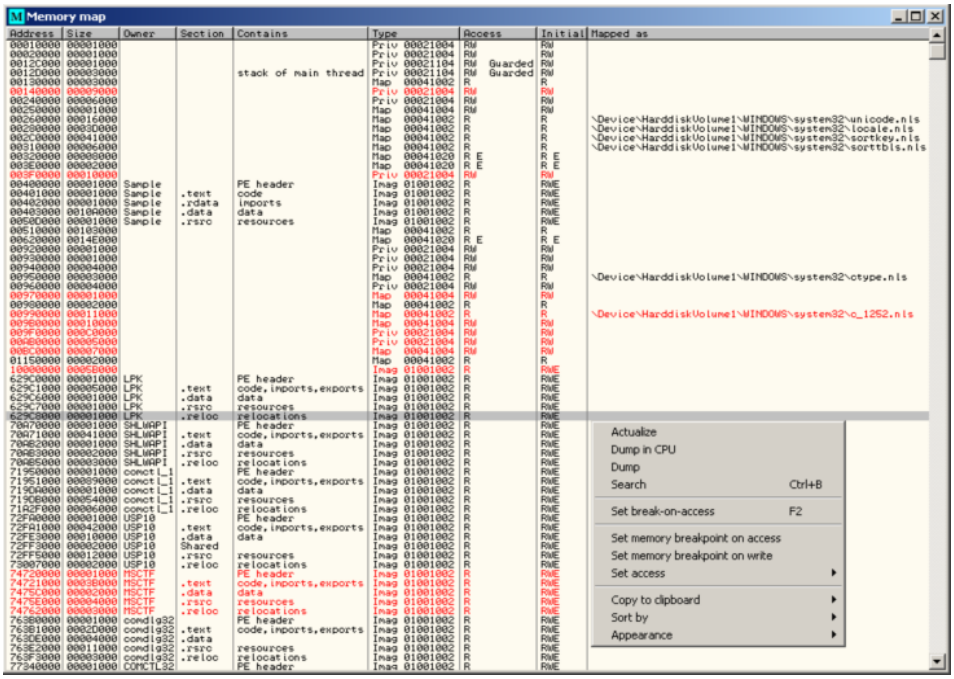
User Time : مدت زمان اجرایی Thread در مد کاربر.

System Time : مدت زمان اجرایی Thread در مد سیستم (هسته). توجه داشته باشید که دو مورد اخیر تنها در ویندوزهای خانوادهٔ NT (NT, 2000, XP, 2003) قابل دسترسی هستند.

از منوی این صفحه نیز می‌توانید به‌منظور تعلیق، از بین بردن و یا تغییر خصوصیات Threadها استفاده کنید.

بررسی نواحی حافظهٔ برنامه

همان‌طور که می‌دانید هر برنامه در ویندوز به ٤ گیگا بایت فضای حافظهٔ مجازی دسترسی داشته و می‌تواند این محدوده را آدرس‌دهی کرده و مورد استفاده قرار دهد. فایل‌های dll مورد نیاز یک فایل اجرایی در محدوده‌های خاصی از این حافظهٔ مجازی قرار می‌گیرند. این فضای حافظه برحسب خصوصیات مشترک به نواحی مختلفی تقسیم می‌شوند که هر یک دارای خصوصیات معینی از قبیل خواندن، نوشتن، اجرا و یا ترکیب‌های آنها می‌باشد. به‌منظور مشاهدهٔ نواحی حافظهٔ برنامه در حال دیباگ، گزینهٔ Memory را از منوی View انتخاب کنید. با این عمل پنجرهٔ Memory map همانند شکل (٥-٥١) ظاهر می‌شود.



شکل (٥-٥١)

همان‌طور که مشاهده می‌کنید نواحی مختلف حافظهٔ برنامه در این پنجره نمایش داده شده‌اند. برخی از این نواحی متعلق به فایل‌های dll مورد نیاز فایل اجرایی هستند که حاوی اطلاعاتی در مورد Section معادل هر یک در فایل اجرایی هستند. اطلاعات نمایش داده شده به ستون‌هایی تقسیم‌بندی شده‌اند که هر یک مسئول نمایش اطلاعات خاصی از ناحیهٔ مورد نظر است. در ادامه به بررسی این ستون‌ها و عملکرد هر یک خواهیم پرداخت.

Address : آدرس شروع ناحیهٔ مورد نظر را در فضای حافظهٔ برنامه مشخص می‌کند.

Size : اندازهٔ ناحیهٔ مورد نظر را مشخص می‌کند.

Owner : در صورتی که ناحیهٔ مورد نظر مربوط به قسمتی (Section) از یک فایل اجرایی و یا dll باشد در این ستون نام فایل مذکور نمایش داده می‌شود.

Section : در صورتی که ناحیهٔ مورد نظر مربوط به قسمتی (Section) از یک فایل اجرایی و یا dll باشد در این ستون نام Section معادل آن در فایل مذکور نمایش داده می‌شود.

Contains : این ستون در صورت امکان توضیحاتی را در مورد محتویات ناحیهٔ مذکور نمایش می‌دهد.

Type : در این ستون نوع ناحیهٔ مورد نظر از لحاظ روش ایجاد و یا استفاده مشخص می‌شود که به سه نوع زیر تقسیم می‌شوند.

Imag : این نوع نواحی معمولاً تصویری از یک فایل اجرایی یا dll را نگهداری می‌کنند.

Priv : برنامه‌ها معمولاً از این نواحی به منظور ذخیره و نگهداری مقادیر و اطلاعات متغیرهای خصوصی خود استفاده می‌کنند.

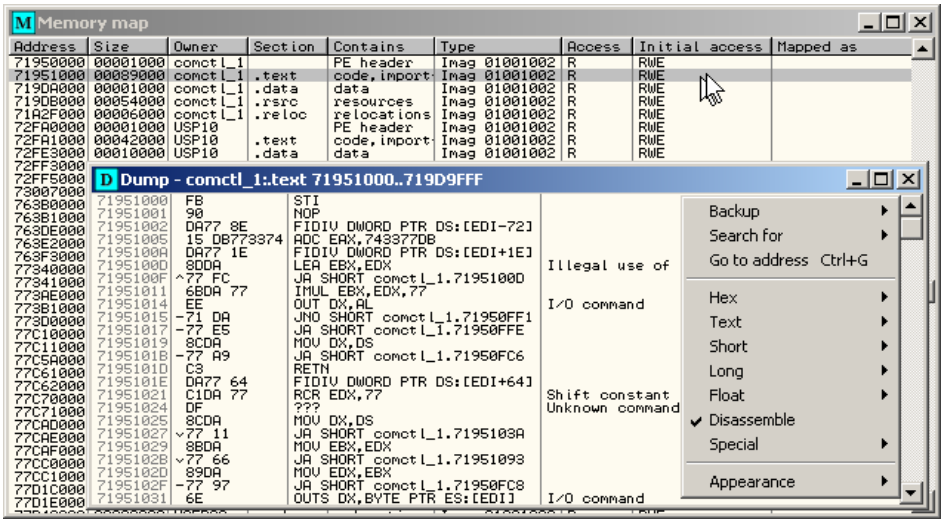
Map : این نواحی معمولاً مربوط به فایل‌ها و یا اطلاعات نگاشت شده به حافظهٔ برنامه هستند که از آنها به منظور تبادل اطلاعات بین برنامه‌های مختلف نیز استفاده می‌شود. در صورت نیاز به منظور کسب اطلاعات بیشتر در مورد فایل‌های نگاشت شده به حافظه و نحوهٔ استفاده از آنها می‌توانید به فصل ۸ مراجعه کنید.

Access : نوع دسترسی ناحیهٔ مورد نظر را از نظر خواندن ، نوشتن و یا اجرا معین می‌کند که توسط حروف R ، W و E معین می‌شوند.

Initial : در این قسمت نوع دسترسی اولیهٔ ناحیهٔ مورد نظر نمایش داده می‌شود.

با استفاده از این پنجره در مراحل دیباگ متوجه خواهید شد که نواحی تازه تخصیص یافته به رنگ قرمز درخواهند آمد که معمولاً با ایجاد متغیرهای جدید ویا عملیات مختلف تخصیص حافظه از قبیل Local Alloc و یا Global Alloc صورت می‌گیرند. در صورت نیاز به منظور کسب اطلاعات دقیق‌تر راجع به روش‌های تخصیص حافظه و نحوهٔ استفاده از آنها می‌توانید به فصل ۸ مراجعه کنید.

با Double Click کردن بر روی ناحیهٔ مورد نظر و یا استفاده از گزینهٔ Dump ، پنجرهٔ Dump همانند شکل (۵-۵۲) ظاهر شده و اطلاعات ناحیهٔ مذکور را با فرمت مناسب به نمایش می‌گذارد. در صورت نیاز می‌توانید با استفاده از منوی پنجرهٔ Dump این فرمت را به نوع دلخواه تغییر دهید.

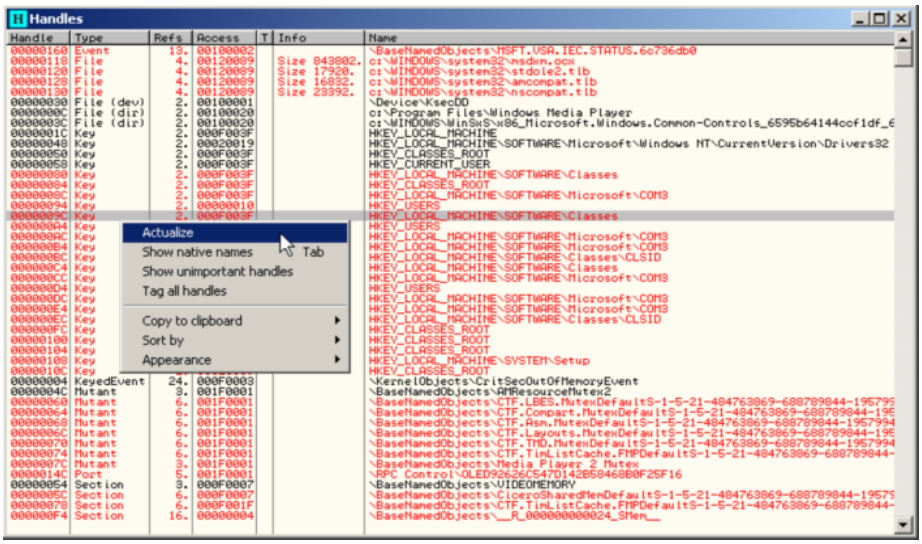


شکل (۵-۵۲)

بررسی شماره‌های دسترسی

همان‌طور که قبلاً اشاره شد هر برنامهٔ در حال اجرا (Process) در ویندوز شماره‌های دسترسی مخصوص به خود را دارد. این شماره‌های دسترسی می‌توانند مربوط به فایل‌ها، پورت‌های باز، رویدادها، ابزارهای همگام‌سازی و یا سایر موارد باشند. بدیهی است که بررسی شماره‌های دسترسی مورد استفادهٔ برنامه و شناخت جزئیات هر یک می‌تواند نکات بسیار مفیدی را در مورد نحوهٔ عملکرد و جزئیات پیاده‌سازی آن مشخص کند.

به منظور بررسی شماره‌های دسترسی مورد استفادهٔ برنامه در مراحل دیباگ و اجرا می‌توانید از گزینهٔ Handles از منوی View استفاده کنید. با این عمل پنجرهٔ Handles همانند شکل (۵-۵۳) ظاهر شده و لیستی از شماره‌های دسترسی ایجاد شده توسط برنامه را به نمایش می‌گذارد.



شکل (۵-۵۳)

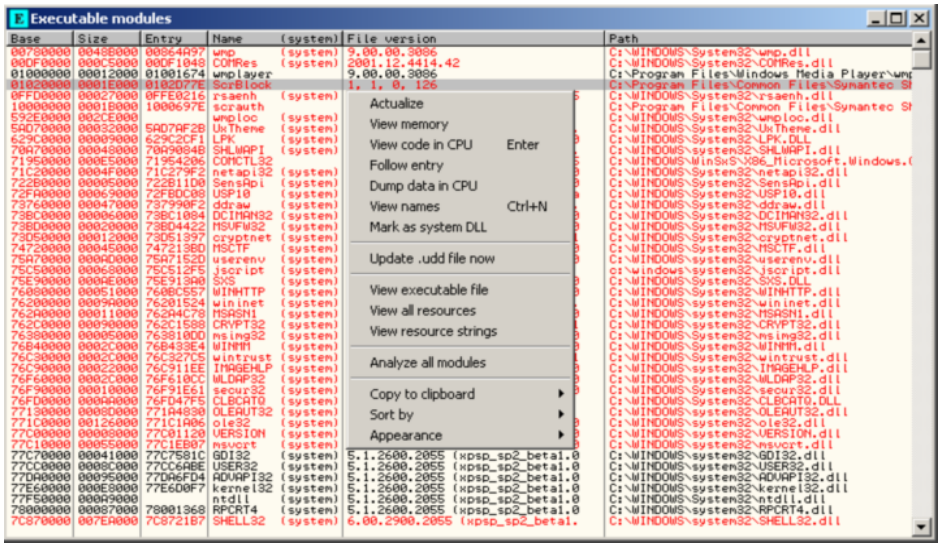
همان‌طور که مشاهده می‌کنید برای هر شمارهٔ دسترسی اطلاعات مربوطه از قبیل نوع، تعداد ارجاع‌ها، سطح دسترسی و نام به نمایش گذاشته می‌شود.

توجه داشته باشید که اطلاعات موجود در این پنجره معمولاً به‌طور خودکار بازسازی نمی‌شوند. به‌منظور بازسازی این اطلاعات همانند شکل (۵-۵۳) از گزینهٔ Actualize واقع در منوی این پنجره استفاده کنید. به‌منظور ایجاد خوانایی بهتر، آیتم‌های تغییر یافته نسبت به مرحلهٔ قبل از Actualize به رنگ قرمز در می‌آیند.

بررسی فایل‌های dll مورد استفادهٔ برنامه

همان‌طور که می‌دانید در ویندوز از فایل‌های dll به‌منظور به اشتراک گذاشتن کدها و منابع بین فایل‌های اجرایی مختلف استفاده می‌شود. یک نمونه از این موارد کتابخانه‌های استاندارد ویندوز و توابع API هستند که فایل‌های اجرایی در این سیستم عامل از آنها به‌طور مشترک استفاده می‌کنند. علاوه بر این موارد برخی از فایل‌های اجرایی از فایل‌های dll متعدد دیگری نیز به‌منظور انجام اعمال موردنظر خود استفاده می‌کنند. در مراحل بررسی و دیباگ کردن یک فایل اجرایی، موارد متعددی وجود دارد که در آنها نیاز به انجام بررسی‌هایی برروی این فایل‌های dll بوجود می‌آید. در اکثر موارد عملیات دیباگ بر روی توابع dll خاص مورد استفادهٔ فایل اجرایی نیز صورت می‌گیرد که نحوهٔ انجام این عملیات دقیقاً مشابه فایل اجرایی است.

به‌منظور بررسی فایل‌های dll مورد استفادهٔ یک برنامه در مراحل دیباگ می‌توانید گزینهٔ Executable Modules را از منوی View انتخاب کنید. با این عمل پنجرهٔ Executable Modules همانند شکل (۵-٥٤) نمایش داده شده و لیستی از فایل‌های dll مورد استفادهٔ برنامه را به همراه جزئیات هر یک به نمایش می‌گذارد. توجه داشته باشید که برخی از این فایل‌های dll توسط بارگذار فایل‌های اجرایی و بر طبق جدول ورودی موجود در آن و برخی دیگر در مراحل اجرا توسط کدهای فایل اجرایی و فراخوانی‌های API بارگذاری شده‌اند.



شکل (۵-٥٤)

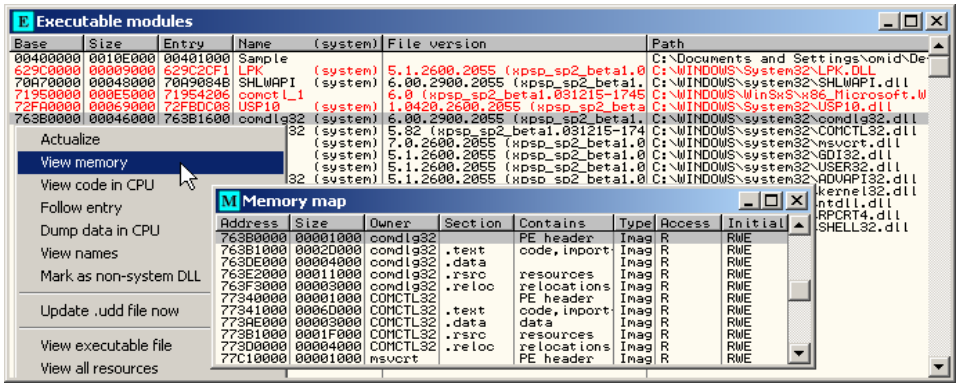
همان‌طور که مشاهده می‌کنید برای هر فایل dll اطلاعات نمایش داده شده شامل آدرس شروع در فضای حافظهٔ برنامه، حجم، آدرس ورودی، نام، شمارهٔ نسخه و مسیرفایل می‌باشد. گزینه‌های موجود در منوی این پنجره نیز توانایی انجام عملیات گسترده‌ای را بازای هر فایل dll دارند که در ادامه به بررسی برخی از آنها خواهیم پرداخت.

Actualize

با انتخاب این گزینه عملیات بررسی فایل‌های dll دوباره صورت گرفته و لیست موجود بازسازی می‌شود. در اکثر موارد این عمل به‌طور خودکار توسط OllyDbg در هنگام بارگذاری فایل‌های dll جدید صورت می‌گیرد و شما نیازی به انجام این عمل به صورت دستی ندارید. توجه داشته باشید که به‌منظور خوانایی بهتر، آیتم‌های جدید بارگذاری شده نسبت به مرحلهٔ قبل از Actualize به رنگ قرمز در می‌آیند.

View Memory

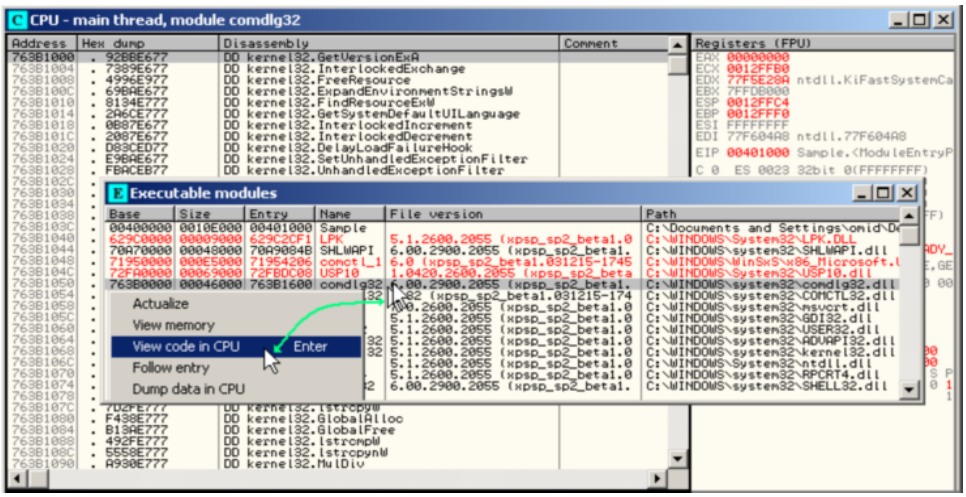
با انتخاب فایل dll موردنظر و استفاده از این گزینه پنجرهٔ Memory map همانند شکل (۵-۵۵) ظاهر شده و موقعیت فایل dll مذکور را در فضای حافظهٔ برنامه به نمایش می‌گذارد.



شکل (۵-۵۵)

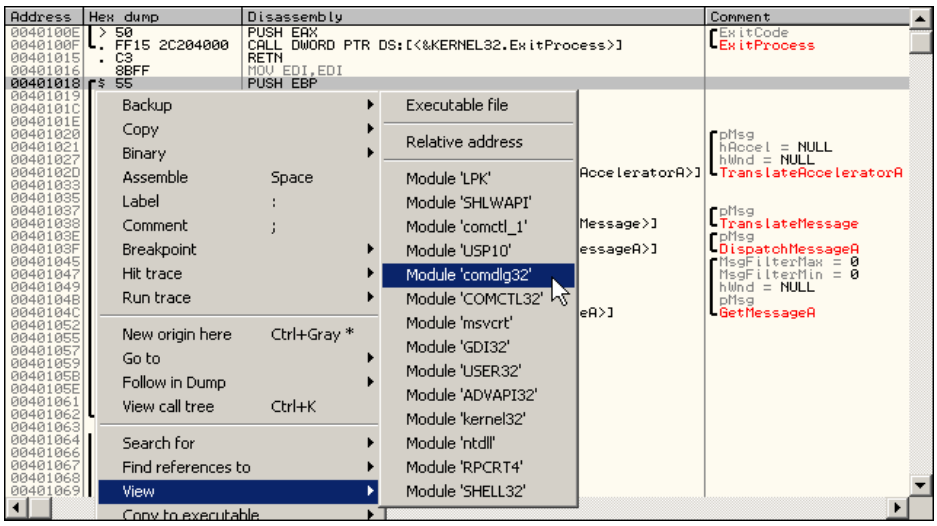
View Code in CPU

با انتخاب فایل dll موردنظر و استفاده از این گزینه و یا Double Click کردن برروی فایل مذکور، پنجرهٔ CPU (Disassembler) همانند شکل (۵-۵۶) اطلاعات اولین ناحیه (Section) فایل مذکور را نمایش خواهد داد.



شکل (۵-۵۶)

به‌منظور دسترسی سریع‌تر به فایل‌های dll در قسمت Disassembler نیز می‌توانید همانند شکل (۵-۵۷) از قسمت View از منوی این پنجره استفاده کنید. بدیهی است که نتیجهٔ دو مثال اخیر یکسان خواهد بود.



شکل (۵-۵۷)

Follow Entry

با انتخاب فایل dll مورد نظر و استفاده از این گزینه، پنجرهٔ CPU (Diesassembler) به آدرس ورودی (Entry Point) فایل dll مذکور منتقل خواهد شد.

View Names

با انتخاب این گزینه لیستی از نام‌های مورد استفاده در فایل dll موردنظر به نمایش درخواهد آمد. این نام‌ها معمولاً مربوط به توابع ورودی، خروجی، کتابخانه‌ها و یا نام‌های تعیین شده توسط کاربر هستند که در بخش "بررسی توابع ورودی و خروجی" به‌طور کامل مورد بررسی قرار خواهند گرفت.

Mark as non-system Dll / Mark as System Dll

در بررسی‌های مختلف به‌طور پیش‌فرض توابع و dllهای سیستمی‌فایل‌هایی هستند که در دایرکتوری System در ویندوزهای 9X و یا دایرکتوری System32 در ویندوزهای خانوادهٔ NT قرار دارند. ولی در صورت نیاز با استفاده از این گزینه‌ها می‌توانید یک فایل dll را به عنوان یک dll سیستمی و یا غیرسیستمی‌معرفی کنید. همان‌طور که در ادامه خواهید دید تغییر این خصوصیت در مراحل تحلیل و بررسی تأثیر مستقیم خواهد گذاشت.

Update .udd File Now

OllyDbg از فایل‌هایی با پسوند Udd به‌منظور ذخیرهٔ نتایج بررسی‌ها، آنالیزها، تنظیمات اعمال شده، نقاط توقف و... برای هر فایل dll و یا exe استفاده می‌کند. در نتیجه این اطلاعات در دفعات بعدی نیز قابل استفاده خواهند بود. به‌طور معمول فایل‌های Udd در هنگام خارج شدن فایل از حافظه (unload) بازنویسی می‌شوند ولی در صورت نیاز با استفاده از این گزینه می‌توانید اطلاعات مذکور را در فایل udd مربوطه بازنویسی کنید.

View Run Trace Profile

با انتخاب این گزینه نتایج عملیات ردیابی روند اجرایی (Run trace) برای فایل dll مورد نظر نمایش داده خواهد شد که در قسمت "روش‌ها و گزینه‌های دیباگ" به‌طور کامل مورد بررسی قرار خواهد گرفت.

Aualyze all modules

به‌طور معمول در هنگام بارگذاری یک فایل exe و یا dll در OllyDbg تنها Section کد اصلی (ناحیه‌ای که آدرس ورودی (Entry Point) در آن قرار دارد) مورد تحلیل و بررسی دقیق قرار

خواهد گرفت. در شکل (۵-۵۸) یک قسمت از کد اسمبلی را قبل از عملیات بررسی (Analyze) و بعد از آن مشاهده می‌کنید.

Before Analyse

```
00401018   55              PUSH EBP
00401019   83EC 1C         SUB ESP,1C
0040101C   8BEC            MOV EBP,ESP
0040101E  ~EB 25           JMP SHORT Sample.00401045
00401020   55              PUSH EBP
00401021   FF35 8C314000   PUSH DWORD PTR DS:[40318C]
00401027   FF35 84314000   PUSH DWORD PTR DS:[403184]
0040102D   FF15 98204000   CALL DWORD PTR DS:[<&USER32.TranslateAcceleratorA>]  USER32.TranslateAcceleratorA
00401033   85C0            TEST EAX,EAX
00401035  ~75 0E           JNZ SHORT Sample.00401045
00401037   55              PUSH EBP
00401038   FF15 9C204000   CALL DWORD PTR DS:[<&USER32.TranslateMessage>]      USER32.TranslateMessage
0040103E   55              PUSH EBP
0040103F   FF15 48204000   CALL DWORD PTR DS:[<&USER32.DispatchMessageA>]      USER32.DispatchMessageA
00401045   6A 00           PUSH 0
00401047   6A 00           PUSH 0
00401049   6A 00           PUSH 0
0040104B   55              PUSH EBP
0040104C   FF15 64204000   CALL DWORD PTR DS:[<&USER32.GetMessageA>]           USER32.GetMessageA
00401052   83F8 FF         CMP EAX,-1
00401055  ~74 07           JE SHORT Sample.0040105E
00401057   85C0            TEST EAX,EAX
00401059  ^75 C5           JNZ SHORT Sample.00401020
0040105B   8B45 08         MOV EAX,DWORD PTR SS:[EBP+8]
0040105E   83C4 1C         ADD ESP,1C
00401061   5D              POP EBP
00401062   C3              RETN
```

After Analyse

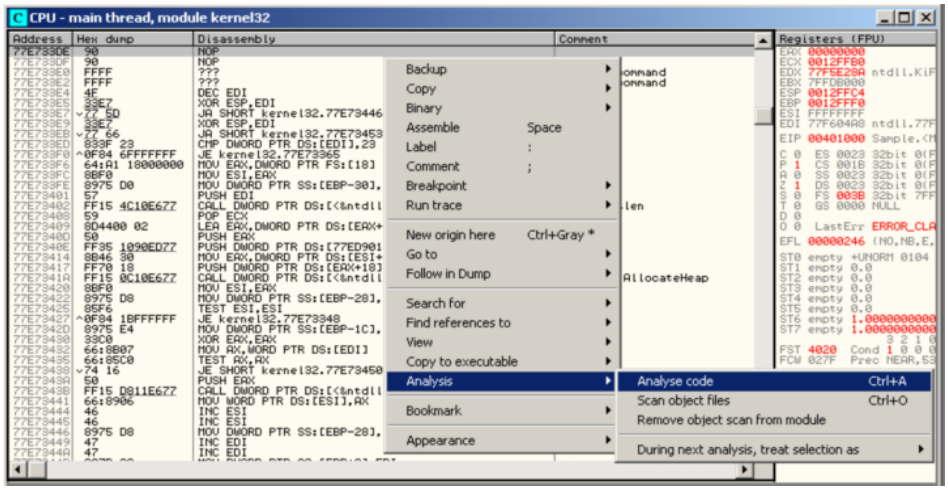
```
00401018  $ 55              PUSH EBP
00401019  . 83EC 1C         SUB ESP,1C
0040101C  . 8BEC            MOV EBP,ESP
0040101E  .~EB 25           JMP SHORT Sample.00401045
00401020  > 55              PUSH EBP                                  pMsg
00401021  . FF35 8C314000   PUSH DWORD PTR DS:[40318C]                 hAccel = NULL
00401027  . FF35 84314000   PUSH DWORD PTR DS:[403184]                 hWnd = NULL
0040102D  . FF15 98204000   CALL DWORD PTR DS:[<&USER32.TranslateAcceleratorA>]  TranslateAcceleratorA
00401033  . 85C0            TEST EAX,EAX
00401035  .~75 0E           JNZ SHORT Sample.00401045
00401037  . 55              PUSH EBP                                  pMsg
00401038  . FF15 9C204000   CALL DWORD PTR DS:[<&USER32.TranslateMessage>]  TranslateMessage
0040103E  . 55              PUSH EBP                                  pMsg
0040103F  . FF15 48204000   CALL DWORD PTR DS:[<&USER32.DispatchMessageA>]  DispatchMessageA
00401045  > 6A 00           PUSH 0                                    MsgFilterMax = 0
00401047  . 6A 00           PUSH 0                                    MsgFilterMin = 0
00401049  . 6A 00           PUSH 0                                    hWnd = NULL
0040104B  . 55              PUSH EBP                                  pMsg
0040104C  . FF15 64204000   CALL DWORD PTR DS:[<&USER32.GetMessageA>]  GetMessageA
00401052  . 83F8 FF         CMP EAX,-1
00401055  .~74 07           JE SHORT Sample.0040105E
00401057  . 85C0            TEST EAX,EAX
00401059  .^75 C5           JNZ SHORT Sample.00401020
0040105B  . 8B45 08         MOV EAX,DWORD PTR SS:[EBP+8]
0040105E  > 83C4 1C         ADD ESP,1C
00401061  . 5D              POP EBP
00401062  . C3              RETN
```

شکل (۵-۵۸)

همان‌طور که مشاهده می‌کنید خوانایی کدها بعد از انجام این بررسی‌ها بسیار افزایش یافته و رشته‌های راهنما و کاراکترهای کمکی فراوانی به آن اضافه شده است. علاوه بر ایجاد خوانایی بیشتر، پس از انجام این عملیات اطلاعات بسیاری در مورد محدوده‌های توابع، ارجاع‌ها، فراخوانی‌ها، رشته‌ها و... برای OllyDbg مشخص می‌شود که از آنها به منظور انجام بررسی‌های بعدی استفاده خواهد شد. در نتیجه بسیاری از عملیات بررسی ذکر شده در این قسمت، بدون انجام این عملیات اولیه بی‌نتیجه خواهند بود. همان‌طور که بیان شد اطلاعات بدست آمده از این بررسی‌های اولیه در فایل‌های udd مربوطه ذخیره شده و در دفعات بعد نیز مورد استفاده قرار می‌گیرند.

در هنگام حرکت در فضای حافظهٔ برنامه و فایل‌های dll بارگذاری شده در آن مشاهده خواهید کرد که بررسی‌های اولیهٔ لازم بر روی آنها صورت نگرفته و کدها نیز از خوانایی کافی برخوردار نیستند.

به منظور انجام این عملیات برروی ناحیه (Section) فعلی می‌توانید همانند شکل (۵-۵۹) از گزینهٔ Analyze code استفاده کنید.



شکل (۵-۵۹)

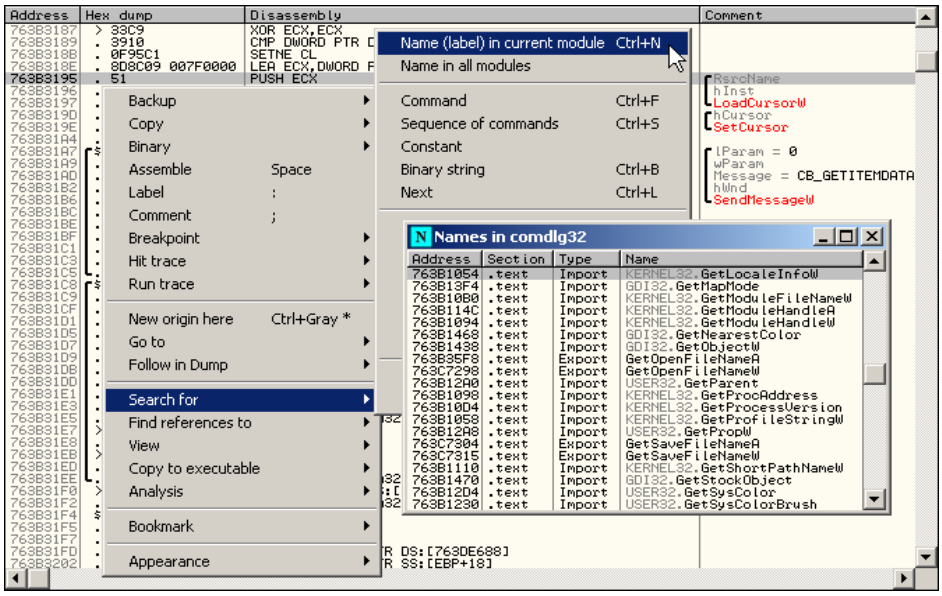
با توجه به تعداد زیاد فایل‌های dll موجود در نواحی مختلف حافظهٔ برنامه توصیه می‌شود که قبل از انجام هرگونه بررسی ابتدا با استفاده از گزینهٔ Analyze all modules عملیات آنالیز را برای تمام فایل‌های dll مورد استفادهٔ برنامهٔ مورد نظر انجام دهید. بدیهی است که انجام این عملیات به زمان نسبتاً زیادی نیاز خواهد داشت ولی باتوجه به ذخیرهٔ نتایج عملیات آنالیز (در فایل‌های udd) و استفادهٔ فایل‌های اجرایی از dll های مشترک، در دفعات بعدی و حتی برای فایل‌های اجرایی دیگر نیز معمولاً نیازی به انجام دوبارهٔ این عملیات نخواهید داشت.

بررسی توابع ورودی و خروجی

همان‌طور که می‌دانید توابع ورودی توابعی هستند که درون فایل اجرایی قرار ندارد و معمولاً به صورت غیرمستقیم از یک فایل dll بارگذاری شده به حافظهٔ برنامه، فراخوانی می‌شوند. توابع API نمونه‌ای از این توابع محسوب می‌شوند. توابع خروجی همانند سایر توابع داخلی یک فایل exe و یا dll هستند با این تفاوت که این توابع می‌توانند به عنوان توابع ورودی بوسیلهٔ سایر فایل‌های exe و یا dll مورد استفاده قرار بگیرند. به منظور کسب اطلاعات دقیق‌تر راجع به توابع و جدول‌های ورودی و خروجی فایل‌های اجرایی و نحوهٔ ایجاد و استفاده از آنها می‌توانید به فصل ۸ مراجعه کنید.

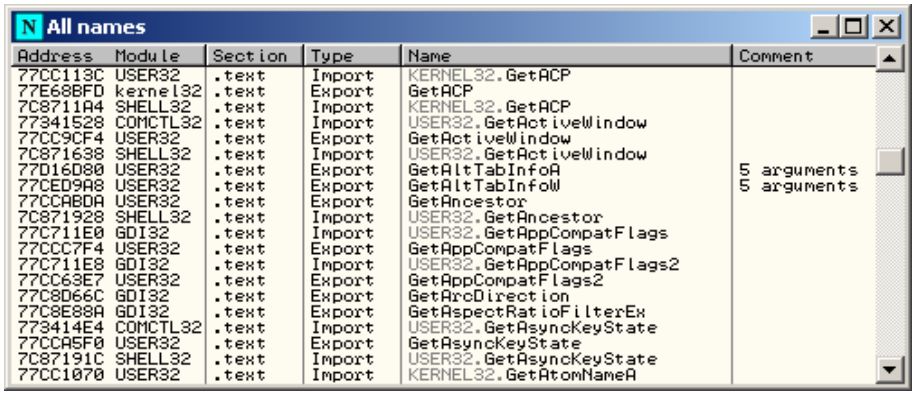
بررسی توابع ورودی و خروجی یک فایل اجرایی و یا dll و نیز کنترل کامل ارجاع‌ها و فراخوانی‌های انجام شده به آنها یکی از اعمال کلیدی در زمینهٔ شناسایی اجزاء و نحوهٔ عملکرد هر یک، در برنامه محسوب می‌شود.

به‌منظور بررسی توابع ورودی و خروجی فایل exe و یا dll متناظر با آدرس فعلی نمایش داده شده در قسمت Disassembler می‌توانید از کلیدهای Ctrl + N استفاده کنید. با این عمل پنجرهٔ Names همانند شکل (۵-۶۰) ظاهر شده و لیستی از نام‌های توابع ورودی (Import) و خروجی (Export) را به نمایش می‌گذارد.



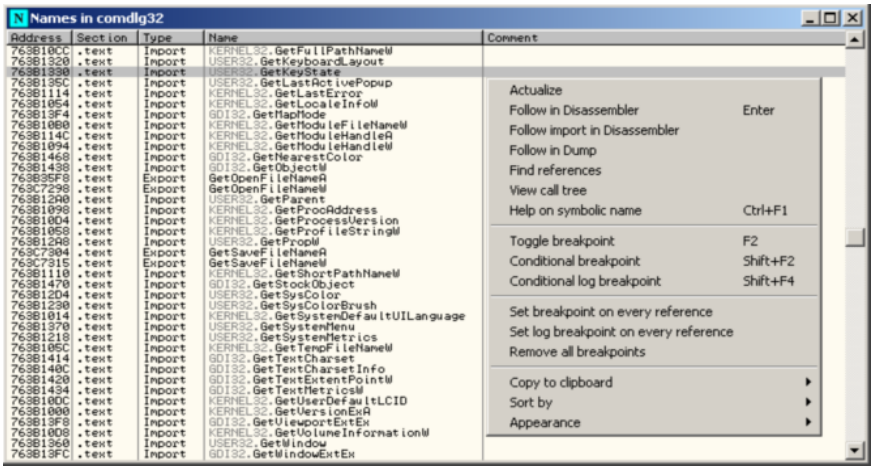
شکل (۵-۶۰)

در صورت نیاز می‌توانید از گزینهٔ Names in all modules به‌منظور نمایش لیست کلیهٔ توابع ورودی و خروجی استفاده شده در کلیهٔ فایل‌های exe و dll بارگذاری شده به حافظهٔ برنامه استفاده کنید. همان‌طور که در شکل (٥-٦١) مشاهده می‌کنید برای هر تابع در ستون Module نام فایل مربوطه نیز ذکر شده است.



شکل (٥-٦١)

با استفاده از گزینه‌های Sort by واقع در منوی این پنجره می‌توانید لیست را به صورت دلخواه مرتب کنید. در پنجرهٔ Names امکانات کاملی به منظور کنترل و دنبال کردن توابع ورودی و خروجی و ارجاع‌های صورت گرفته به آنها وجود دارد که توسط منوی این صفحه قابل دسترسی هستند. در شکل (٥-٦٢) گزینه‌های موجود در این منو را مشاهده می‌کنید.

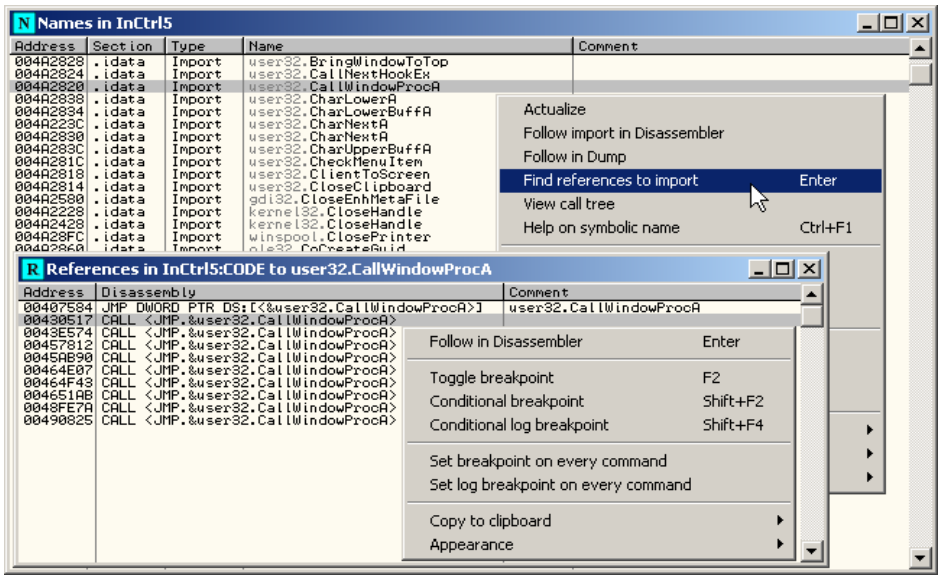


شکل (٥-٦٢)

همان‌طور که مشاهده می‌کنید گزینه‌های این لیست شامل عملیات ایجاد نقاط توقف، دنبال کردن توابع و ارجاع‌ها و نحوه نمایش اطلاعات هستند. در ادامه برخی از این گزینه‌ها را مورد بررسی قرار خواهیم داد.

Find References

با انتخاب تابع مورد نظر در لیست توابع ورودی و خروجی یک فایل و استفاده از این گزینه، پنجرهٔ References همانند شکل (۵-۶۳) ظاهر شده و لیستی از کلیهٔ فراخوانی‌های مستقیم و یا غیرمستقیم صورت گرفته از تابع مورد نظر در فایل مذکور را به نمایش می‌گذارد.

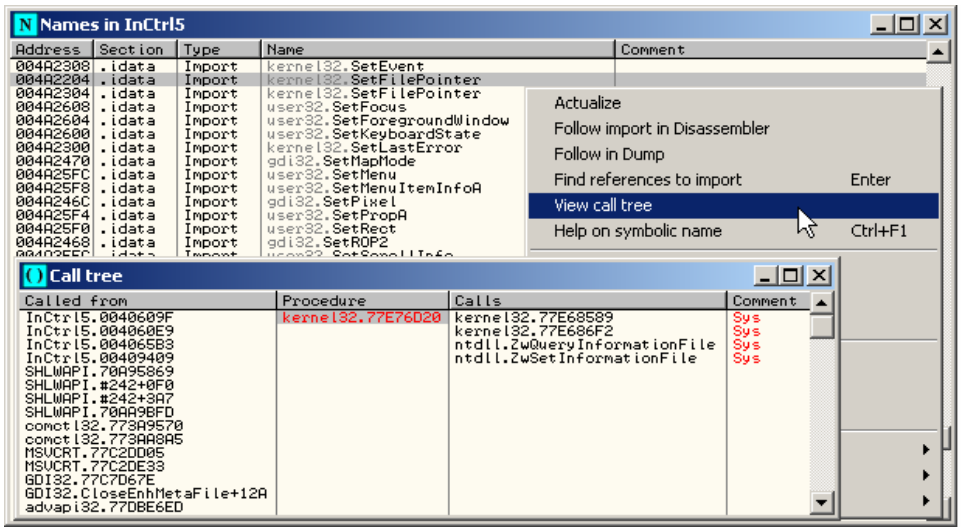


شکل (۵-۶۳)

همان‌طور که مشاهده می‌کنید در پنجرهٔ Reference نیز گزینه‌هایی به‌منظور دنبال کردن و ایجاد نقاط توقف عمومی و یا خصوصی برای فراخوانی‌های نمایش داده شده در لیست وجود دارد.

View call tree

با انتخاب تابع مورد نظر و استفاده از این گزینه، پنجرهٔ Call Tree همانند شکل (۵-۶۴) ظاهر شده و لیستی از فراخوانی‌های صورت گرفته از تابع ورودی یا خروجی موردنظر و فراخوانی‌های انجام گرفته توسط آن را نمایش خواهد داد. پنجرهٔ call tree و جزئیات آن را در قسمت‌های بعد به‌طور دقیق مورد بررسی قرار خواهیم داد.



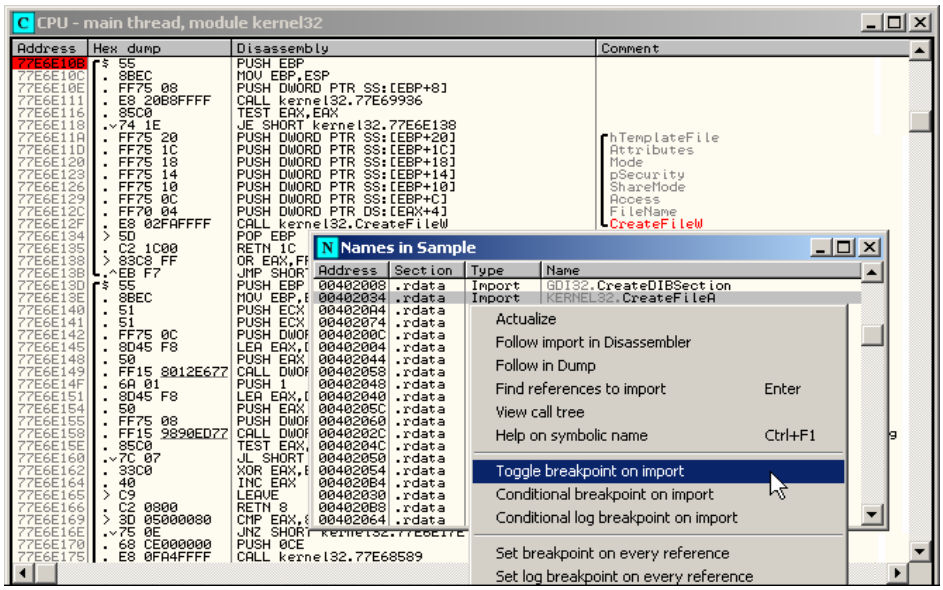
شکل (۵-٦٤)

Help on Symbolic Name

در صورت انتخاب تابع مورد نظر و موجود بودن فایل کمک Win32.hlp، با استفاده از این گزینه و یا کلیدهای Ctrl+F1، توضیحات کاملی در مورد آن تابع و نحوهٔ عملکرد آن نمایش داده خواهد شد.

Toggle breakpoint

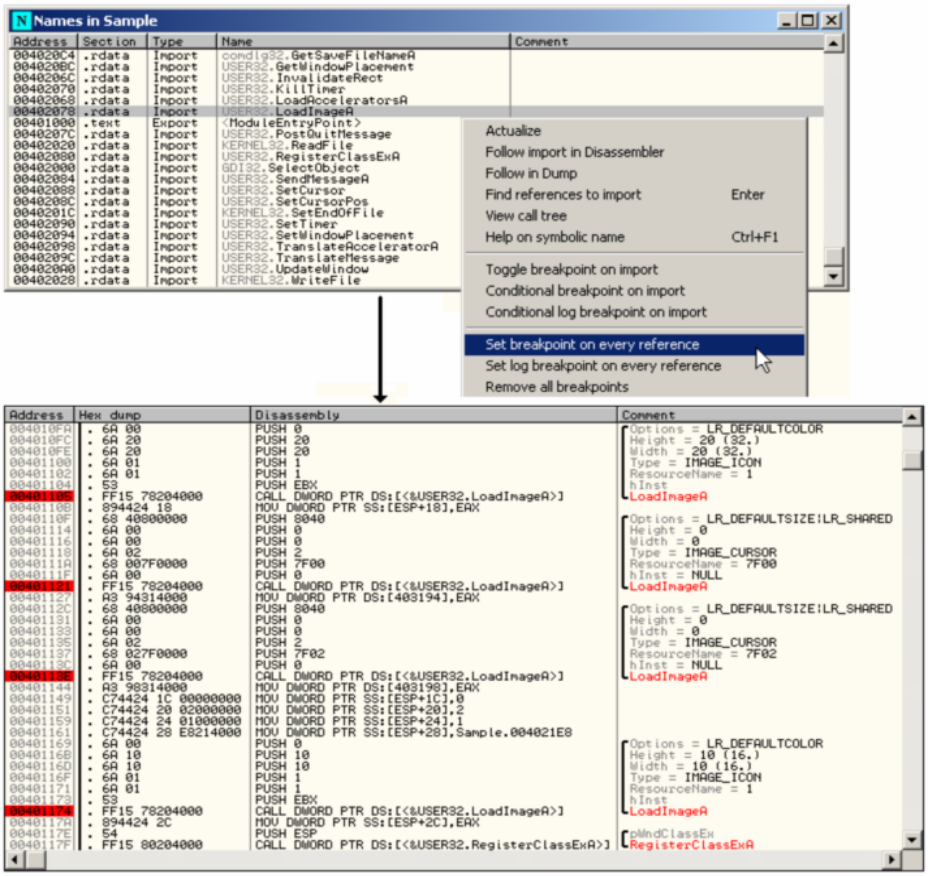
با استفاده از گزینه‌های این قسمت می‌توانید نقطهٔ توقفی را برای آدرس شروع تابع مورد نظر در فضای حافظهٔ برنامه تعیین کنید. در شکل (۵-٦٥) یک نقطهٔ توقف معمولی برای آدرس شروع تابع CreateFileA در فایل kernel32.dll تعیین شده است. بدیهی است که با تعیین این نقاط توقف، کنترل کاملی بر روی کلیهٔ فراخوانی‌های صورت گرفته به تابع مورد نظر ایجاد خواهید کرد.



شکل (۵-۶۵)

Set breakpoint on every reference

عملکرد نقاط توقف برای ارجاعها (on reference) مشابه نقاط توقف قبلی است با این تفاوت که آنها برروی کلیهٔ فرخوانی‌ها و ارجاع‌های شناسایی شده به تابع موردنظر در فایل exe و یا dll فعلی اعمال می‌شوند. برتری آنها نسبت به نوع قبلی در امکان ایجاد نقاط توقف برای فراخوانی‌های انجام شده در یک فایل مشخص است. در نوع قبل امکان تعیین محدوده‌ای برای کنترل فراخوانی‌ها وجود ندارد. همین امر باعث ایجاد توقف‌های غیردلخواه بسیاری می‌گردد. در شکل (۵-۶۶) نقاط توقف برای کلیهٔ فراخوانی‌های صورت گرفته به تابع LoadImageA در فایل Sample.exe تعیین شده‌اند.



شکل (۵-۶۶)

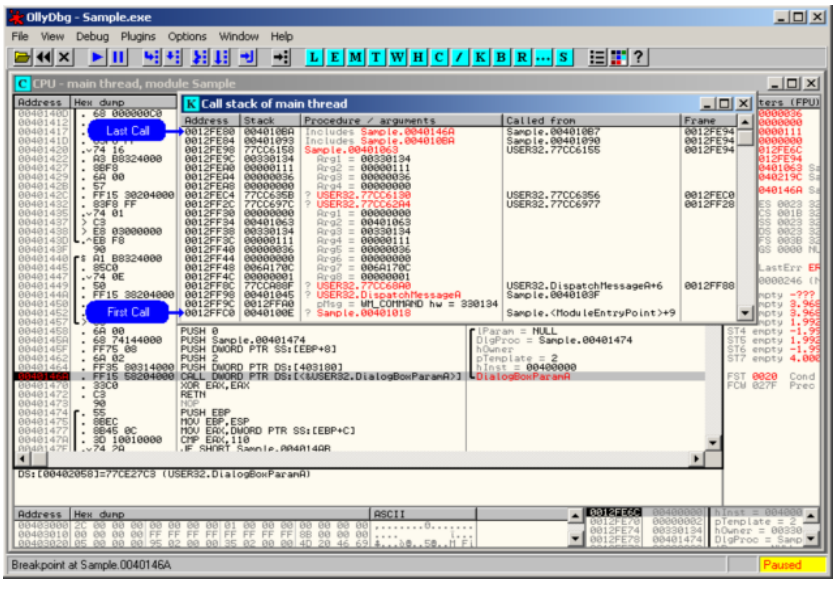
توجه داشته باشید که با استفاده از این‌گونه نقاط توقف دیگر کنترلی برروی فراخوانی‌های صورت گرفته با استفاده از آدرس خروجی تابع GetProcAddress نخواهید داشت. همان‌طور که می‌دانید از این تابع به‌منظور استخراج آدرس توابع خروجی (Export) موجود در یک فایل exe و یا dll استفاده می‌شود. این آدرس‌ها می‌توانند به‌منظور فراخوانی تابع موردنظر به کار گرفته شوند.

بررسی فراخوانی‌های تودرتوی انجام شده

همان‌طور که می‌دانید یکی از کاربردهای Stack ذخیرهٔ آدرس بازگشت در دستورالعمل‌های Call است. در نتیجه با اتمام عملیات یک تابع و فراخوانی دستورات RET (بازگشت) کنترل به دستورالعمل بعد از Call مذکور منتقل شده و روند اجرایی برنامه به‌طور صحیح دنبال خواهد شد. در بسیاری از حالات از جمله توقف‌های ایجاد شده بوسیلهٔ نقاط توقف، داشتن اطلاعات دقیق راجع به فراخوانی‌های تودرتوی انجام شده تا نقطهٔ فعلی یکی از مهمترین و کلیدی‌ترین نکات در زمینهٔ شناسایی نحوهٔ عملکرد توابع و قسمت‌های مختلف یک برنامه و وظایف هر یک محسوب می‌شود.

OllyDbg با استفاده از تکنیک‌های بسیار قوی و پیچیده‌ای تغییرات Stack را زیر نظر گرفته و فراخوانی‌ها، آدرس‌های بازگشت و پارامترهای ارسال شده به توابع را شناسایی و ذخیره می‌کند. توجه داشته باشید که این اطلاعات تنها در صورت متوقف شدن روند اجرایی بر اثر نقاط توقف و یا سایر موارد، قابل دسترسی و نمایش هستند و OllyDbg این عملیات را تنها برای Thread اصلی برنامه انجام خواهد داد.

به‌منظور نمایش این اطلاعات، پس از توقف برنامه در نقطهٔ موردنظر، از گزینهٔ Call Stack در منوی View و یا کلیدهای Alt+K استفاده کنید. با این عمل پنجرهٔ Call Stack همانند شکل (۵-۶۷) نمایش داده خواهد شد.



شکل (۵-۶۷)

همان‌طور که مشاهده می‌کنید در این پنجره اطلاعات دقیق و کاملی راجع به فراخوانی‌های تودرتوی انجام شده تا محل فعلی و پارامترهای ارسالی به هر یک به نمایش گذاشته شده است.اطلاعات نمایش داده شده در پنجرهٔ Call Stack به پنج ستون تقسیم‌بندی شده‌اند که هر یک مسئول نمایش قسمت خاصی از اطلاعات مربوطه است. در ادامه به بررسی جزئیات اطلاعات نمایش داده شده در این ستون‌ها خواهیم پرداخت.

Address

محل قرار گرفتن آدرس و یا پارامتر موردنظر در Stack را مشخص می‌کند که آدرسی در فضای حافظهٔ برنامه است.

Stack

مقدار متناظر با آدرس ذکر شده در قسمت قبل را در فضای Stack مشخص می‌کند. این عدد به طور معمول یک آدرس بازگشت یا مقدار یک پارامتر ارسال شده به یک تابع است.

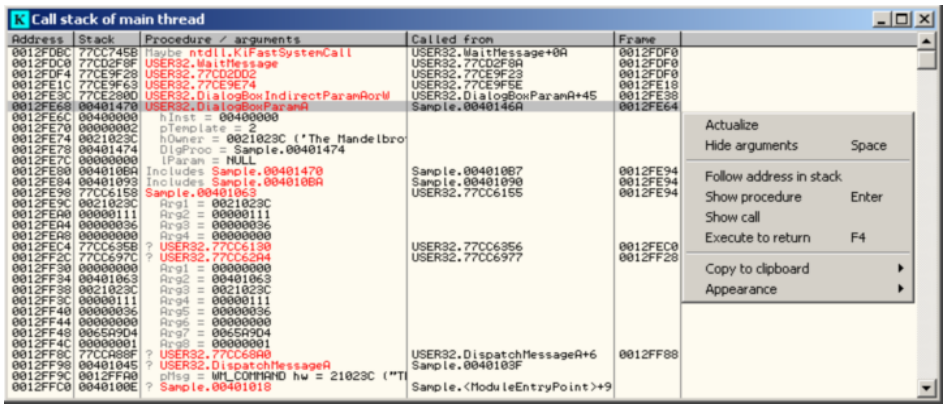
Procedure / Arguments و یا Procedre

در این ستون نام تابع فراخوانی شده و یا آرماگون‌های ارسالی به توابع نمایش داده می‌شوند.

Called From

این ستون حاوی آدرس دستورالعملی است که تابع ذکر شده در قسمت procedure را فراخوانی کرده است.

در پنجرهٔ Call stack امکاناتی به‌منظور دنبال کردن مقادیر و آدرس‌ها و کنترل روند اجرایی برنامهٔ در حال دیباگ وجود دارد که توسط منوی این پنجره قابل دسترسی هستند. در شکل (٥-٦٨) گزینه‌های موجود در این منو را مشاهده می‌کنید.



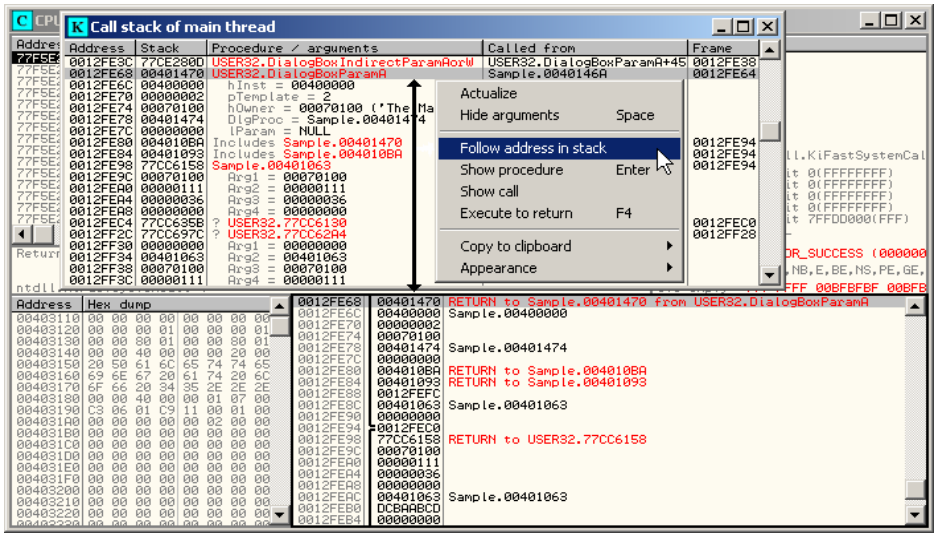
شکل (٥-٦٨)

Show Arguments / Hide Arguments

با استفاده از این گزینه‌ها می‌توانید نمایش و یا عدم نمایش پارامترهای ارسال شده به توابع را مشخص کنید.

Follow Address in stack

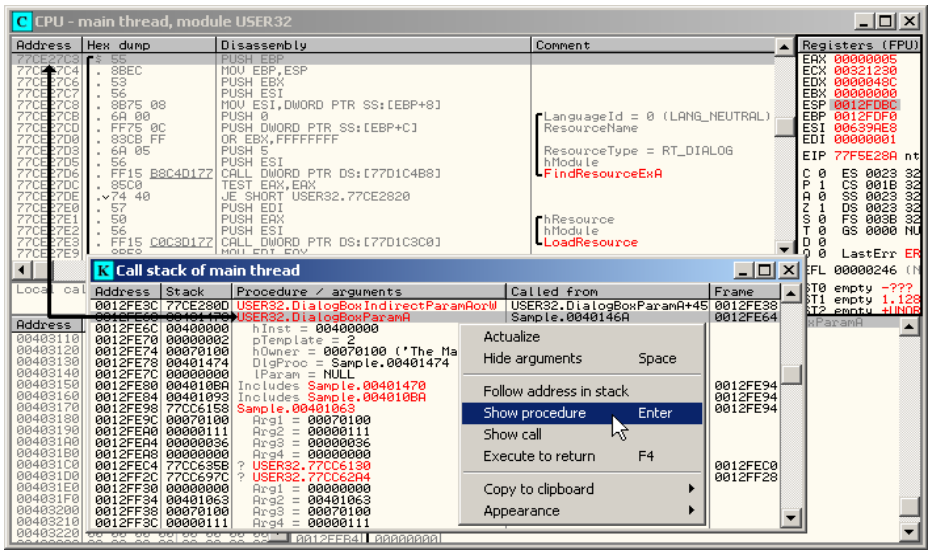
با انتخاب سطر موردنظر و استفاده از این گزینه، آدرس متناظر در قسمت stack از پنجرهٔ CPU همانند شکل (٥-٦٩) انتخاب خواهد شد.



شکل (٥-٦٩)

Show Procdure

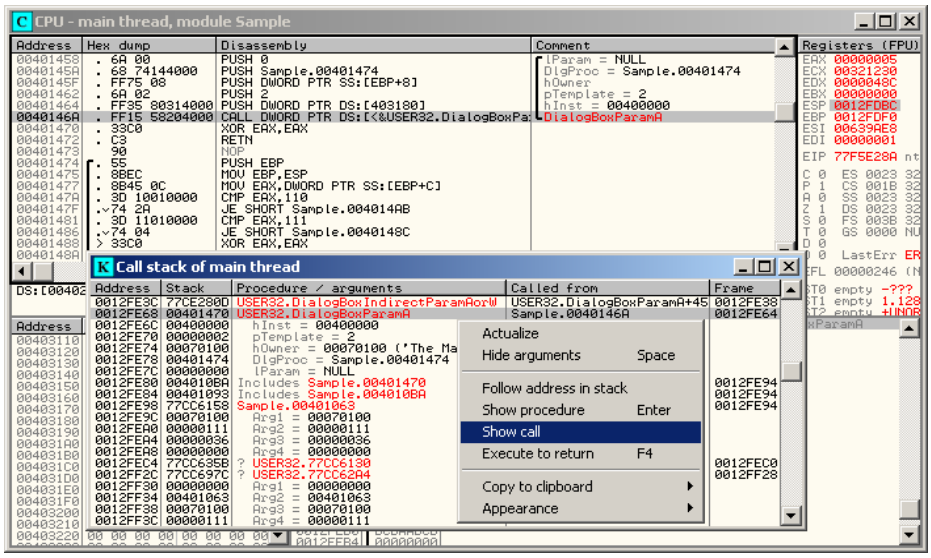
با انتخاب تابع موردنظر در لیست و استفاده از این گزینه، آدرس شروع تابع مذکور در فایل مربوطه انتخاب خواهد شد. همانطور که در شکل (۵-۷۰) مشاهده می‌کنید، با استفاده از این گزینه، آدرس شروع تابع DialogBoxParamA از فایل User32.dll در قسمت Disassembler انتخاب شده است.



شکل (۵-۷۰)

Show call

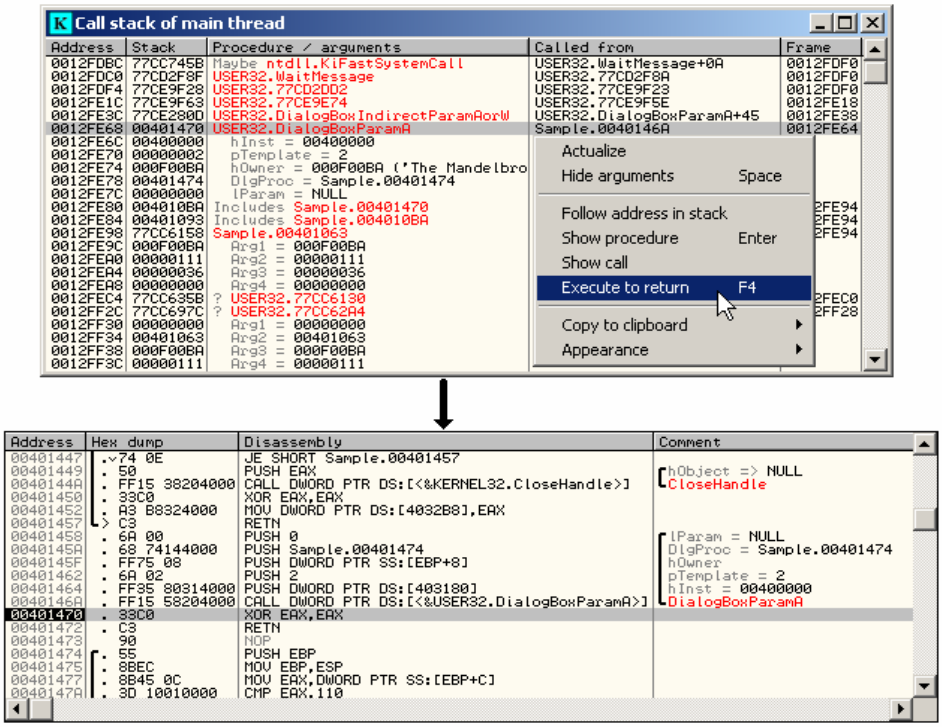
با انتخاب سطر مورد نظر در لیست و استفاده از این گزینه آدرس دستورالعمل call متناظر که در قسمت Called from نمایش داده شده است، در قسمت Disassembler انتخاب خواهد شد. همان‌طور که در شکل (۵-۷۱) مشاهده می‌کنید، با استفاده از این گزینه آدرس فراخوانی تابع DialogBoxParamA در قسمت Disassembler انتخاب شده است.



شکل (۵-۷۱)

Execute to return

با انتخاب این گزینه روند اجرای برنامه تا هنگام بازگشت از تابع فراخوانی شده که در ستون Procedure ذکر شده است ادامه خواهد یافت. در حقیقت این گزینه از یک نقطهٔ توقف در آدرس بعد از Call استفاده می‌کند. همانطور که در شکل (۵-۷۲) مشاهده می‌کنید. پس از استفاده از این گزینه روند اجرای برنامه ادامه یافته و در دستور بعد از Call DWORD PTR DS: [<& User32.DialogParamA>] در آدرس 401470 متوقف خواهد شد.



شکل (۵-۷۲)

توجه داشته باشید که اطلاعات نمایش داده شده در پنجرهٔ Call Stack در برخی موارد از صحت و دقت کافی برخوردار نیستند در نتیجه توصیه می‌شود که در صورت لزوم از اطلاعات نمایش داده شده در قسمت Stack از پنجرهٔ CPU به‌منظور ردیابی فراخوانی‌ها، پارامترها و سایر موارد استفاده کنید.

کنترل، بررسی و ردیابی روند اجرایی

یکی از امکانات استاندارد کلیهٔ Debuggerها، توانایی آنها در اجرای مرحله به مرحلهٔ یک برنامه است. در حقیقت با این عمل Debuggerها پس از اجرای هر دستورالعمل با متوقف شدن روند اجرایی، کنترل را به دست گرفته و بررسی‌ها و یا تغییرات موردنظر خود را انجام خواهند داد و در صورت لزوم دوباره کنترل را به برنامهٔ در حال دیباگ بازخواهند گرداند. این حلقه تا خاتمهٔ برنامه ادامه خواهد یافت. به‌منظور کسب اطلاعات دقیق‌تر راجع به Debuggerها و نحوهٔ عملکرد و پیاده‌سازی آنها به فصل ۸ مراجعه کنید.

بدیهی است که داشتن اطلاعات کافی در مورد ترتیب اجرای دستورالعمل‌ها در فایل اجرایی از طریق اجرای مرحله به مرحله و نیز کنترل تغییرات ایجاد شده توسط آنها در وضعیت ثبات‌ها، Stack، حافظه و ... گام اصلی و نهایی در شناخت نحوهٔ عملکرد دقیق برنامه و اجزاء آن محسوب می‌شود.

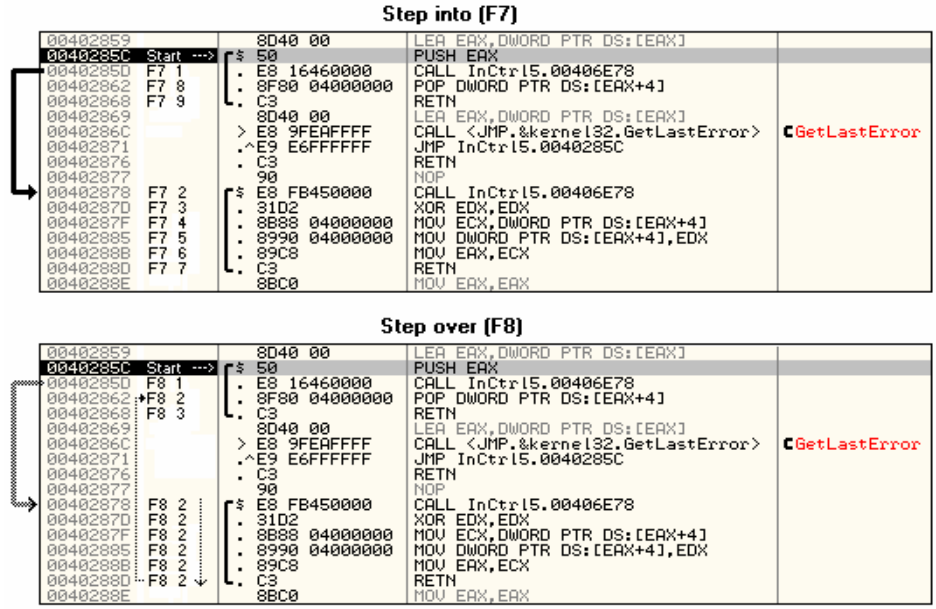
OllyDbg امکانات بسیار گسترده و منحصر به فردی را به‌منظور کنترل و اجرای مرحله به مرحلهٔ دستورالعمل‌ها در یک فایل اجرایی تدارک دیده است. علاوه بر این امکانات این نرم‌افزار از دو روش دیگر نیز به‌منظور ردیابی روند اجرایی استفاده می‌کند که به نوبه خود می‌توانند بسیار مفید واقع شوند. در ادامه، امکانات این نرم‌افزار را در زمینهٔ اجرای مرحله به مرحله و ردیابی روند اجرایی مورد بررسی قرار خواهیم داد.

اجرای مرحله به مرحله

در اجرای مرحله به مرحله، کاربر روند اجرایی دستورالعمل‌ها را کنترل کرده و آن را به صورت مرحله به مرحله به اجرا در می‌آورد. توجه داشته باشید گزینه‌های مربوط به این نوع اجرا تنها در حالت توقف روند اجرایی قابل دسترسی و استفاده هستند. همان‌طور که می‌دانید روش‌های گوناگونی به‌منظور متوقف کردن روند اجرایی یک فایل اجرایی وجود دارد از آن جمله می‌توان به انواع گوناگون نقاط توقف اشاره کرد. روش دیگر استفاده از کلید F12 است که باعث توقف روند اجرایی می‌شود. با استفاده از این کلید متوجه خواهید شد این روش از دقت کافی برخوردار نبوده و معمولاً روند اجرایی در داخلی‌ترین فراخوانی‌ها و در فایل‌های dll سیستمی ویندوز متوقف خواهد شد.

پس از توقف برنامه می‌توانید کنترل اجرای دستورالعمل‌ها را با استفاده از کلیدهای F7 (Step into) و F8 (Step over) به دست بگیرید. به این ترتیب که با هر بار فشردن این کلیدها یک دستورالعمل اجرا می‌شود. تفاوت Step over و Step into در این است که در صورتی که دستورالعمل فعلی یک فراخوانی (call) باشد با فشردن F7 (Step into) , ollyDbg وارد تابع فراخوانی شده می‌شود و

روند اجرایی در آدرس شروع این تابع متوقف خواهد شد و در حقیقت عملیات از آدرس شروع تابع موردنظر دنبال خواهد شد. با استفاده از گزینهٔ Step Over (کلید F8) دستورالعمل‌های call همانند سایر دستورالعمل‌ها اجرا و دنبال خواهند شد و درحقیقت بافشردن کلید F8 برروی یک دستورالعمل Call، کلیهٔ دستورالعمل‌های تابع موردنظر به‌صورت متوالی اجرا شده و روند اجرایی پس از بازگشت تابع در دستورالعمل بعد از Call متوقف می‌شود. در شکل (۵-۷۳) نحوهٔ عملکرد این دو روش و تفاوت‌های بین آنها را مشاهده می‌کنید.



شکل (۵-۷۳)

برای دنبال کردن اتوماتیک روند اجرایی با استفاده از متدهای step over و step inot که در حقیقت فشردن متوالی این کلیدها را توسط کاربر شبیه‌سازی می‌کنند، می‌توانید از کلیدهای Ctrl+F8 (Animate over) و Ctrl+F7 (Animate into) استفاده کنید. همان‌طور که ذکر شد این گزینه‌ها در حقیقت فشردن متوالی کلیدهای F7 و F8 را شبیه‌سازی می‌کنند. در صورت لزوم به‌منظور متوقف کردن این عملیات می‌توانید از کلید ESC استفاده کنید.

اجرای آزاد

در این نوع اجرا، روند اجرایی برنامه به‌طور معمول صورت می‌گیرد و کاربر از نحوهٔ اجرای مرحله به مرحله (تک تک دستورالعمل‌ها) مطلع نخواهد شد. بدیهی است که این نوع اجرا از سرعت بالایی

(نزدیک به سرعت اجرای بدون Debugger) برخوردار خواهد بود. به‌منظور دنبال‌کردن روند اجرایی به‌صورت فوق می‌توانید از کلید F9 (RUN) استفاده کنید. توقف روند اجرایی در روش فوق به دو صورت امکان‌پذیر است.

استفاده ازنقاط توقف: دراین روش روند اجرایی با استفاده ازنقاط توقف کنترل و متوقف خواهد شد.

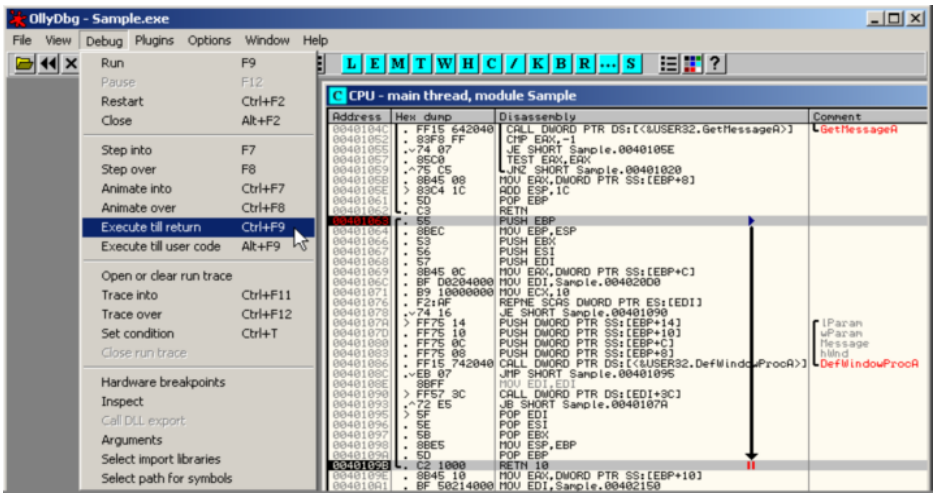
استفاده از گزینهٔ pause (F12): با استفاده از این کلید روند اجرایی در دستورالعمل فعلی در حال اجرا متوقف خواهد شد.

پس از توقف عملیات با استفاده از متدهای فوق و انجام بررسی‌ها و تغییرات موردنظر، روند اجرایی می‌تواند با استفادهٔ دوباره از کلید F9 و یا روش‌های دیگر ردیابی و اجرای مرحله به مرحله دنبال شود.

پس از توقف برنامه توسط نقاط توقف و یا گزینهٔ pause (F12) می‌توانید از دو گزینهٔ زیر به منظور دنبال کردن سریع‌تر کدها و رسیدن به نقطهٔ دلخواه استفاده کنید.

۱- Execute till return : در صورت توقف برنامه، با استفاده از این گزینه روند اجرایی برنامه تا رسیدن به یک دستورالعمل بازگشت (Return) به‌صورت آزاد دنبال خواهد شد. همان‌طور که می‌دانید دستورالعمل‌های بازگشت به‌طور معمول به‌منظور خاتمهٔ روند اجرایی یک تابع و بازگشت به آدرس فراخوانی‌کننده به کار گرفته می‌شوند.

در شکل (۵-۷٤) پس از توقف روند اجرایی در آدرس 401063 عملیات با استفاده از گزینهٔ Execute till return دنبال شده است.



شکل (۵-۷٤)

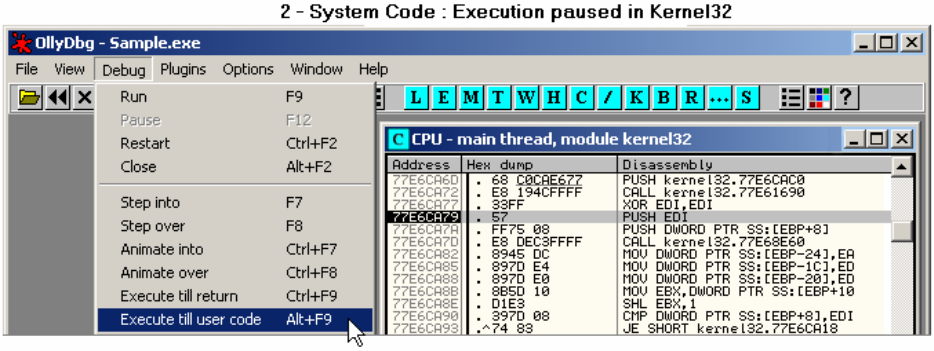
همان‌طور که مشاهده می‌کنید دستورالعمل‌های این تابع به‌صورت آزاد اجرا شده و روند اجرایی در آدرس 401098 که یک دستورالعمل بازگشت (RetN 10) است دوباره متوقف شده است.

۲- Execute till user code : در اکثر مواقع با استفاده از گزینهٔ pause (F12)، روند اجرایی در کدهای فایل‌های سیستمی و dll ویندوز متوقف خواهد شد. در این‌گونه مواقع به‌منظور اجرای آزاد کدهای سیستمی تا رسیدن (بازگشت) به کدهای کاربر می‌توانید از این گزینه استفاده کنید. توجه داشته باشید به‌صورت پیش‌فرض تمام dll ها و فایل‌هایی که درون دایرکتوری System ویندوز قرار داشته باشند به‌عنوان فایل‌ها و کدهای سیستمی درنظر گرفته خواهند شد. در صورت نیاز با استفاده از پنجرهٔ Executable modules می‌توانید فایل‌های مورد استفاده و بارگذاری شده را مورد بررسی قرار داده و فایل‌های موردنظر را به عنوان سیستمی و یا کاربر در نظر بگیرید. بدیهی است که این تغییرات در مراحل دنبال کردن کدها و روند اجرایی برنامه، در موارد زیادی از جمله مورد مذکور تأثیر خواهد گذاشت. در شکل (۵-۷۵) پس از فراخوانی تابع LoadImageA توسط کد کاربر روند اجرایی در مراحل اجرای تابع مذکور و در آدرسی از فایل Kernel32.dll متوقف شده است.

1- User Code : Call to LoadImageA

```
004010FA  . 6A 00          PUSH 0                                   Options = LR_DEFAULTCOLOR
004010FC  . 6A 20          PUSH 20                                  Height = 20 (32.)
004010FE  . 6A 20          PUSH 20                                  Width = 20 (32.)
00401100  . 6A 01          PUSH 1                                   Type = IMAGE_ICON
00401102  . 6A 01          PUSH 1                                   ResourceName = 1
00401104  . 53             PUSH EBX                                 hInst
00401105  . FF15 78204000  CALL DWORD PTR DS:[<&USER32.LoadImageA>] LoadImageA
0040110B  . 894424 18      MOV DWORD PTR SS:[ESP+18],EAX
0040110F  . 68 40800000    PUSH 8040                                Options = LR_DEFAULTSIZE|LR_SHARED
```

2 - System Code : Execution paused in Kernel32



3 - User Code : Return to User Code using 'Execute till user code'

```
004010FA  . 6A 00          PUSH 0                                   Options = LR_DEFAULTCOLOR
004010FC  . 6A 20          PUSH 20                                  Height = 20 (32.)
004010FE  . 6A 20          PUSH 20                                  Width = 20 (32.)
00401100  . 6A 01          PUSH 1                                   Type = IMAGE_ICON
00401102  . 6A 01          PUSH 1                                   ResourceName = 1
00401104  . 53             PUSH EBX                                 hInst
0040110E  . FF15 78204000  CALL DWORD PTR DS:[<&USER32.LoadImageA>] LoadImageA
0040110B  . 894424 18      MOV DWORD PTR SS:[ESP+18],EAX
0040110F  . 68 40800000    PUSH 8040                                Options = LR_DEFAULTSIZE|LR_SHARED
```

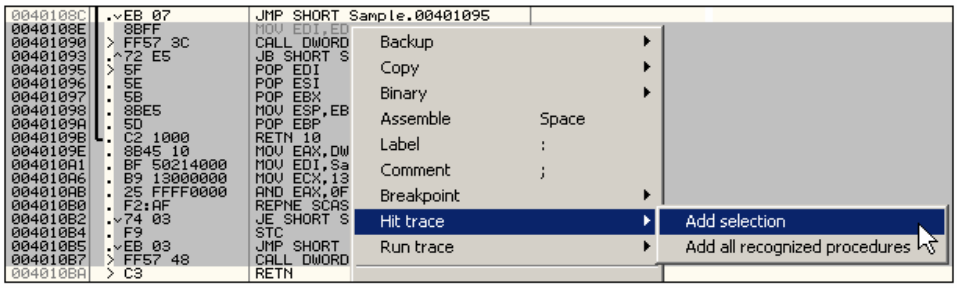
شکل (۵-۷۵)

همان‌طور که مشاهده می‌کنید با استفاده از گزینهٔ Execute till user code روند اجرایی به‌صورت آزاد تا بازگشت به کد کاربر دنبال شده و در آدرس دستورالعمل بعد از فراخوانی تابع LoadImageA متوقف شده است.

بررسی و کنترل دستورالعمل‌های اجرا شده

در بسیاری از موارد در اختیار داشتن اطلاعاتی در مورد قسمت‌های اجرا شده و اجرا نشدهٔ کد می‌تواند به‌منظور انجام کنترل‌ها و بررسی‌های دقیق‌تر بعدی بسیار مفید بوده و باعث ایجاد تسریع در عملیات شناسایی گردد. به این منظور OllyDbg متد اجرایی خاصی را با نام Hit trace ارائه کرده است که می‌تواند نواحی اجرا شده و اجرا نشدهٔ کد را در محدودهٔ زمانی خاصی مشخص کند. به این منظور ابتدا باید محدودهٔ موردنظر از کد انتخاب شود.سپس می‌توانید از سه گزینهٔ زیر استفاده کنید.

۱- انتخاب یک محدودهٔ مشخص: به این منظور ابتدا در قست Disassembler محدودهٔ مورد نظر را همانند شکل (۵-۷۶) انتخاب کرده و سپس از گزینهٔ Add selection در قسمت Hit trace از منوی Disassembler استفاده کنید.

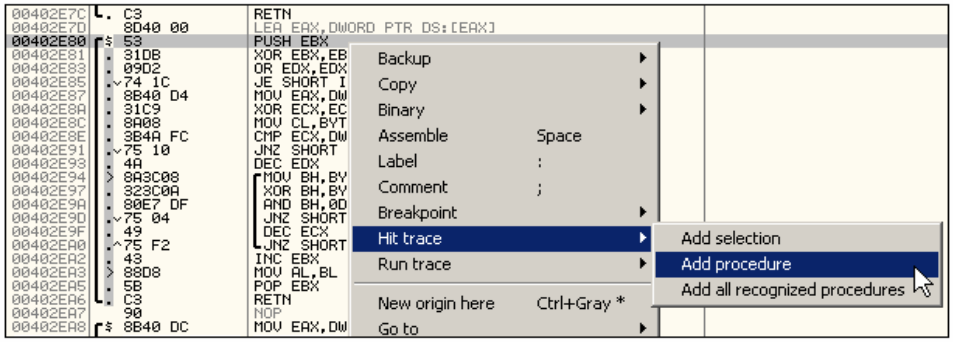


شکل (۵-۷۶)

با تعیین محدوده‌ها، OllyDbg برای هر دستورالعمل موجود در محدوده یک نقطهٔ توقف معمولی (INT3) ایجاد خواهد کرد و پس از فعال شدن نقطهٔ مذکور ابتدا آن را از بین برده و سپس دستورالعمل متناظر با آن را به عنوان یک دستورالعمل اجرا شده در قسمت Hex dump با رنگ قرمز علامت‌گذاری می‌کند.

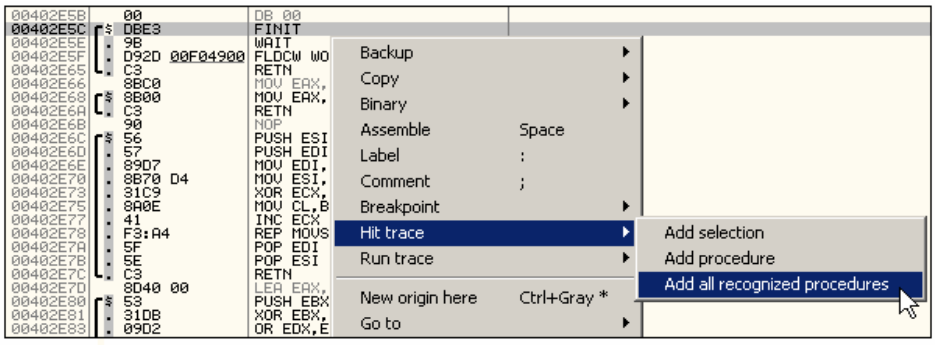
توجه داشته باشید که ایجاد نقاط توقف در محدوده‌های Data باعث تخریب آنها می‌شود در نتیجه در مراحل تعیین محدوده‌ها و نقاط توقف سعی کنید از این امر اجتناب کنید. در صورت نیاز به‌منظور کنترل دسترسی‌ها به محدوده‌های Data از نقاط توقف مخصوصی استفاده می‌شود که قبلاً آنها را مورد بررسی قرار داده‌ایم.

۲- انتخاب تابع فعلی : در صورت نیاز می‌توانید محدودهٔ تابع فعلی را به محدودهٔ تحت کنترل اضافه کنید. به این منظور پس از انتخاب آدرسی در محدودهٔ تابع مورد نظر در قسمت Disassembler، از گزینهٔ add procedure همانند شکل (۵-۷۷) استفاده کنید.



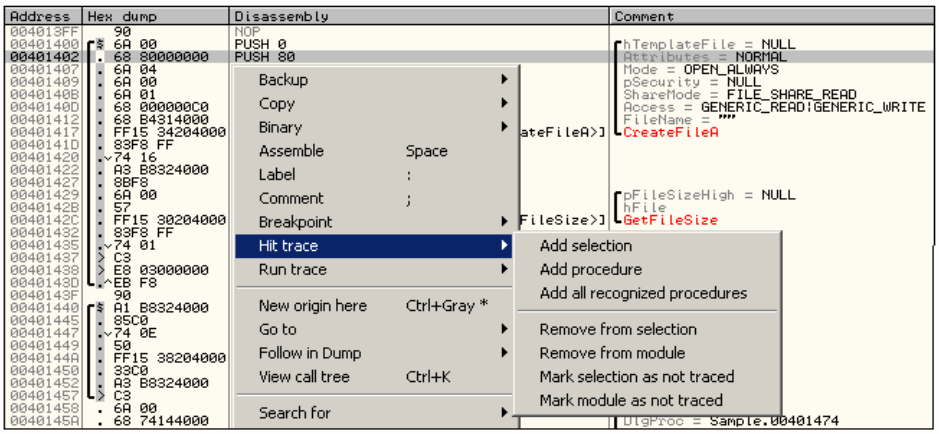
شکل (۵-۷۷)

انتخاب تمامی توابع شناسایی شده: پس از عملیات بررسی اولیه (Analyze) و شناسایی توابع موجود و محدوده‌های هر یک می‌توانید از گزینهٔ Add All recognized procedures به منظور انتخاب تمامی محدوده‌های توابع شناسایی شده برای عملیات کنترل استفاده کنید.



شکل (۵-۷۸)

پس از تعیین محدوده‌ها، چهار گزینهٔ دیگر نیز به منوی Hit trace افزوده می‌شود که می‌توانید از آنها به منظور حذف و یا تغییر خصوصیت محدوده‌های تعریف شده استفاده کنید.



شکل (۵-۷۹)

- **Remove from Selection** : با انتخاب قسمتی از یک محدودهٔ تحت کنترل و استفاده از این گزینه می‌توانید آن را از لیست محدوده‌های تحت کنترل خارج کنید.
- **Remove from module** : با استفاده از این گزینه می‌توانید کلیهٔ محدوده‌های کنترلی تعیین شده در فایل فعلی را از بین ببرید.
- **Mark selection as not traced** : با انتخاب محدودهٔ موردنظر و استفاده از این گزینه کلیهٔ علائم مربوط به کدهای اجرا شده از آن ناحیه برداشته شده و در حقیقت نقاط توقف جدیدی ایجاد می‌شود که می‌تواند به‌منظور کنترل دوبارهٔ آن محدوده، از زمان فعلی به بعد به کار گرفته شود.
- **Mark module as not traced** : عملکرد این گزینه دقیقاً همانند گزینهٔ قبل است با این تفاوت که کلیهٔ محدوده‌های تعیین شده در فایل فعلی تجدید خواهند شد.

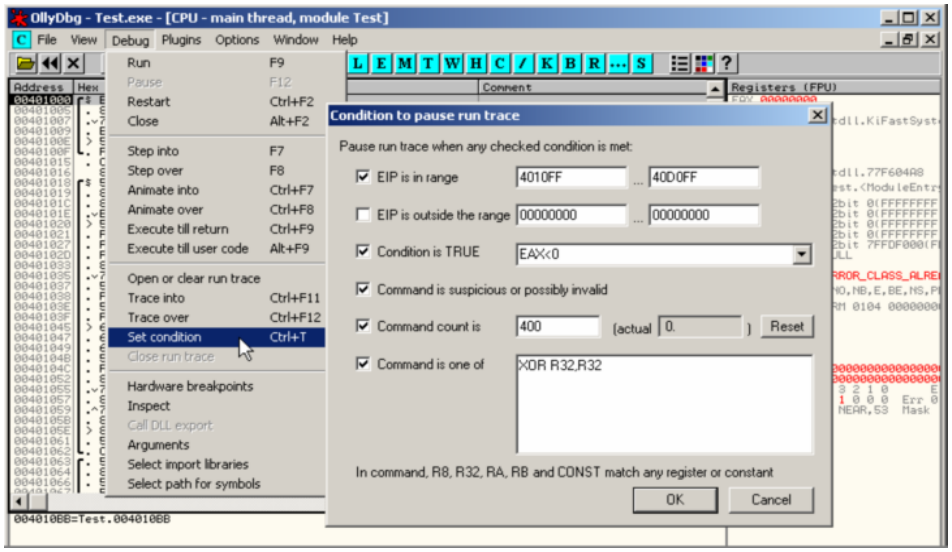
ردیابی مراحل اجرای برنامه (Run Trace)

همان‌طور که می‌دانید بررسی و ردیابی کدهای اجرا شده توسط یک برنامه و نیز داشتن اطلاعات کافی در مورد تغییراتی که پس از اجرای هر یک از این دستورالعمل‌ها در وضعیت ثبات‌ها وحافظهٔ برنامه ایجاد می‌گردد یکی از مهم‌ترین و دقیق‌ترین متدها در بررسی نحوهٔ عملکرد یک برنامه و شناسایی اجزاء و روابط بین آنها محسوب می‌شود.

OllyDbg گزینه‌ای را به‌منظور ردیابی دقیق مراحل اجرایی در قسمت‌های موردنظر از کد فایل اجرایی تدارک دیده است این روش Run trace نامیده می‌شود. نحوهٔ عملکرد این روش همانند روش Hit trace است با این تفاوت که OllyDbg پس از اجرای هر دستورالعمل جزئیات کاملی را راجع به

آدرس فعلی، وضعیت ثبات‌ها، پیام‌ها و... مورد بررسی قرار داده و ذخیره کرده و امکان انجام بررسی‌های دقیق‌تر و دنبال کردن آنها را برای کاربر فراهم می‌کند.

قبل از شروع عملیات ردیابی بهتر است ابتدا جزئیاتی را که پس از اجرای هر دستورالعمل توسط OllyDbg باید کنترل شوند، مشخص کنید. با استفاده از این گزینه‌ها می‌توانید شرایطی را برای توقف روند اجرا و ردیابی تعیین کنید که می‌تواند در مراحل ردیابی بسیار کارگشا باشد. استفاده از این شرایط می‌تواند کنترل کامل‌تری را نسبت به نقاط توقف معمولی و شرطی برای شما ایجاد کند ولی باعث کند شدن روند اجرایی شده وسربار زیادی را به سیستم تحمیل می‌کند. برای تعیین این شرایط از گزینهٔ Set condition از منوی Debug استفاده کنید. با این عمل پنجرهٔ Condition to pause همانند شکل (۵-۸۰) ظاهر شده و امکان تعریف شرط‌های مورد نظر را برای شما فراهم می‌کند.



شکل (۵-۸۰)

همان‌طور که مشاهده می‌کنید شرط‌ها به ٦ دسته تقسیم شده‌اند که در ادامه برخی از آنها را مورد بررسی قرار خواهیم داد.

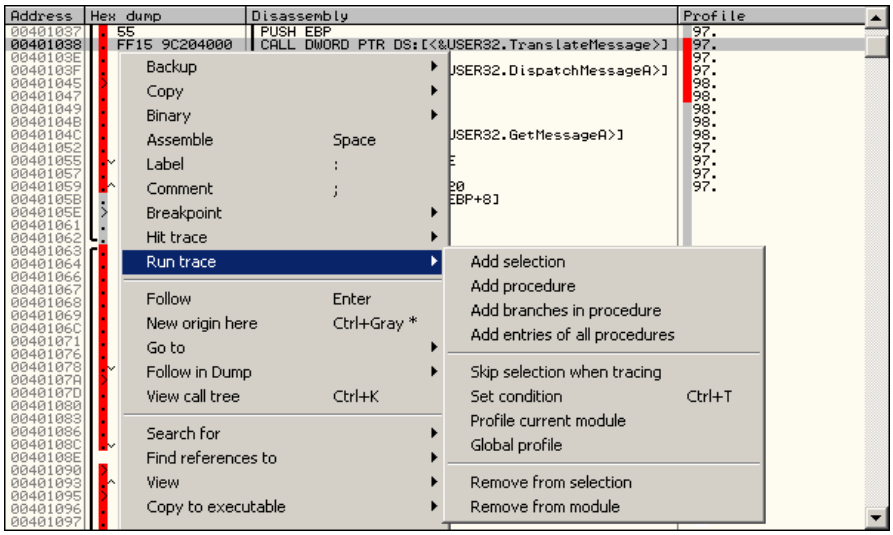
- **EIP is in range** : توقف در صورتی که آدرس دستورالعمل فعلی در بازهٔ موردنظر باشد.
- **EIP is outside the range** : توقف در صورتی که آدرس دستورالعمل فعلی خارج از بازهٔ موردنظر باشد.

- Condition is true : توقف در صورت بوجود آمدن وضعیتی که توسط یک رشتهٔ شرطی مشخص می‌شود. ساختار این رشته‌های شرطی در قسمت‌های قبل مورد بررسی قرار گرفته‌اند.
- **Command count is** : توقف در صورتی که تعداد معینی از دستورات از کدهای برنامه اجرا شوند.
- **Command is one of** : توقف در صورتی که دستورالعمل بعدی با یکی از الگوهای مشخص شده مطابقت داشته باشد. روش و ساختار تعریف الگوها در قسمت‌های قبل مورد بررسی قرار گرفته‌اند ولی به‌منظور یادآوری به جدول (۵-۵) توجه کنید.

	XOR EAX , EAX	XOR ESI , EDX
XOR R32 , R32	✓	✓
XOR RA , RA	✓	✘
XOR RA , RB	✘	✓

جدول (۵-۵)

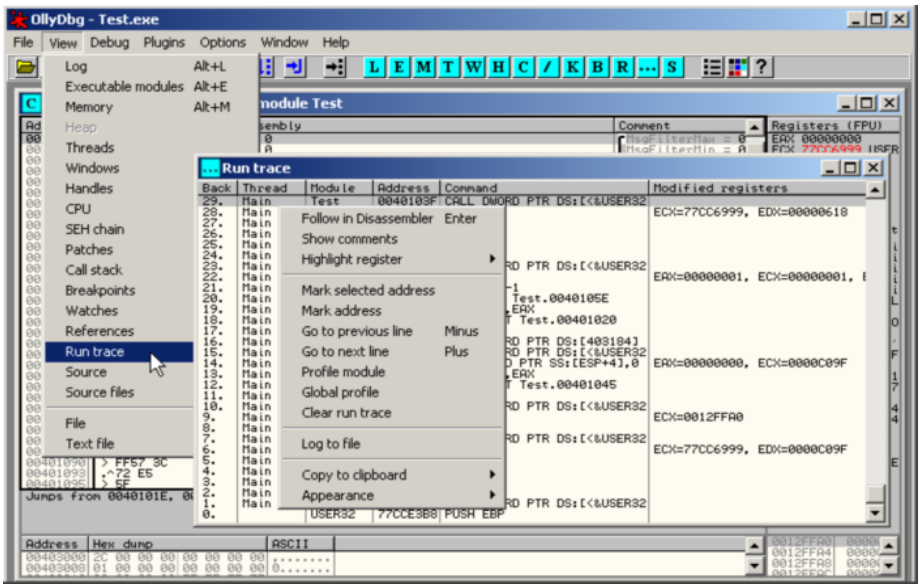
حال که شرطهای مورد نظر خود را برای توقف تعیین کردید ، می‌توانید عملیات ردیابی را آغاز کنید. به این منظور می‌توانید از کلیدهای Ctrl+F11 (Trace into) برای دنبال کردن و ردیابی کدهای اجرا شده توسط دستورالعمل‌های call و یا Ctrl+F12 (Trace over) به منظور اجرای آزاد کدهای اجرا شده توسط این دستورالعمل‌ها و گذر از آنها استفاده کنید. همان‌طور که متوجه خواهید شد، این روش ردیابی (Run trace) بسیار کند بوده و به حافظهٔ زیادی نیز احتیاج دارد. به‌منظور کاهش سربار در مراحل اجرا و تسریع روند اجرایی می‌توانید محدودهٔ خاصی را برای ردیابی تعیین کنید. به این منظور ابتدا با استفاده از گزینه‌های موجود در قسمت Run trace از منوی Disassembler محدوده‌های مورد نظر را تعیین کرده و سپس اجرای برنامه را با استفاده از گزینهٔ Run (F9) دنبال کنید. در شکل (۵-۸۱) گزینه‌های موجود در این قسمت را مشاهده می‌کنید. در ادامه نحوهٔ عملکرد برخی از آنها را مورد بررسی قرار خواهیم داد.



شکل (۵-۸۱)

- **Add selection** : محدودهٔ انتخاب شده به محدوده‌های تحت کنترل اضافه می‌شود.
- **Add procedure** : محدودهٔ تابع فعلی به محدوده‌های تحت کنترل اضافه می‌شود.
- **Add Branches in procedure** : دستورالعمل‌های Jump و Call شناسایی شده در تابع فعلی به محدوده‌های تحت کنترل اضافه می‌شود.
- **Add Entries of all procedures** : آدرس‌های شروع توابع شناسایی شده به محدوده‌های تحت کنترل اضافه می‌شوند.
- **Skip selection when tracing** : محدودهٔ انتخاب شده از لیست محدوده‌های تحت کنترل خارج می‌شود. معمولاً از این گزینه به‌منظور حذف عملیات ردیابی و کنترل از حلقه‌هایی که دارای تکرار زیاد هستند استفاده می‌شود.
- **Remove from module** : تمام محدوده‌های کنترلی موجود در فایل فعلی حذف می‌شوند.

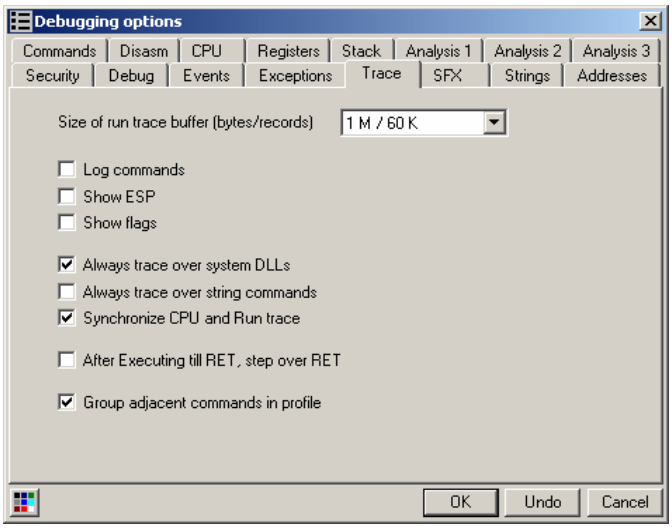
پس از توقف روند اجرا در صورت نیاز می‌توانید با استفاده از کلیدهای + و – دستورالعمل‌های ردیابی شده را مورد بررسی قرار داده و یا به‌منظور انجام بررسی‌های دقیق‌تر از گزینهٔ Run Trace در منوی View استفاده کنید. با انجام این عمل پنجرهٔ Run Trace همانند شکل (۵-۸۲) ظاهر شده و لیستی از دستورالعمل‌های ردیابی شده را به همراه اطلاعات دقیقی راجع به هر یک به نمایش می‌گذارد.



شکل (۵-۸۲)

به‌منظور ایجاد هماهنگی بین آدرس‌های انتخاب شده در دو پنجرهٔ Run Trace و Disassembler می‌توانید از گزینهٔ Synchronize CPU and Run Trace واقع در قسمت Trace از پنجرهٔ Debugging Options استفاده کنید. همان‌طور که در شکل (۵-۸۲) مشاهده می‌کنید با استفاده از منوی این پنجره امکان انجام بررسی‌های دقیق‌تر برروی اطلاعات ردیابی شده و نیز ذخیرهٔ آنها در صورت نیاز وجود دارد.

در اکثر موارد شما تمایلی به ردیابی روند اجرایی در فایل‌های dll سیستمی ویندوز و کدهای توابع API ندارید به این منظور می‌توانید گزینهٔ "Always Trace Over System Dlls" را فعال کنید. همان‌طور که در شکل (۵-۸۳) مشاهده می‌کنید این گزینه در قسمت Trace از پنجرهٔ Debugging Options قرار دارد.


Debugging options
Commands
Disasm
CPU
Registers
Stack
Analysis 1
Analysis 2
Analysis 3
Security
Debug
Events
Exceptions
Trace
SFX
Strings
Addresses
Size of run trace buffer (bytes/records)
1 M / 60 K
Log commands
Show ESP
Show flags
Always trace over system DLLs
Always trace over string commands
Synchronize CPU and Run trace
After Executing till RET, step over RET
Group adjacent commands in profile
OK
Undo
Cancel

شکل (۵-۸۳)

فصل ششم

ایجاد تغییرات در فایل های اجرایی

فصل ششم

ایجاد تغییرات در فایل های اجرایی

در فصل‌های قبل با نحوه انجام تحلیل‌ها و بررسی‌ها و شناسایی اجزاء و نحوه عملکرد آنها در فایل اجرایی آشنایی پیدا کردید. در برخی موارد پس از انجام این بررسی‌ها لازم است که تغییراتی در فایل و یا فایل‌های موردنظر ایجاد شده و نتایج آنها در روند اجرایی و نحوه عملکرد برنامه‌ها، مورد بررسی قرار گیرد. با توجه به شرایط و انواع گوناگون برنامه‌ها، نحوه ایجاد این تغییرات متفاوت بوده و نیاز به بررسی‌های کاملی دارد. با توجه به موارد مذکور انتخاب صحیح محل ایجاد تغییرات و روش به کار گرفته شده برای انجام آنها از اهمیت زیادی برخوردار است. می‌توان گفت که انتخاب مسیر صحیح در این مرحله تا حدود بسیار زیادی به اطلاعات جمع‌آوری شده در مراحل قبل بستگی دارد. در نتیجه انجام تغییرات بدون در دست داشتن اطلاعات کافی می‌تواند باعث ایجاد عملکردها و واکنش‌های ناخواسته از طرف برنامه موردنظر و ایجاد سردرگمی‌در شما شود.

در این فصل روش‌های گوناگون ایجاد تغییرات در فایل‌های اجرایی را مورد بررسی قرار خواهیم داد.

ایجاد تغییرات در منابع

این نوع از تغییرات معمولاً به‌منظور تغییر در جزئیات ظاهری یک نرم‌افزار، تغییردر زبان و رشته‌های نمایش داده شده و یا اضافه کردن منابع جدید برای استفاده‌های بعدی صورت می‌گیرد.

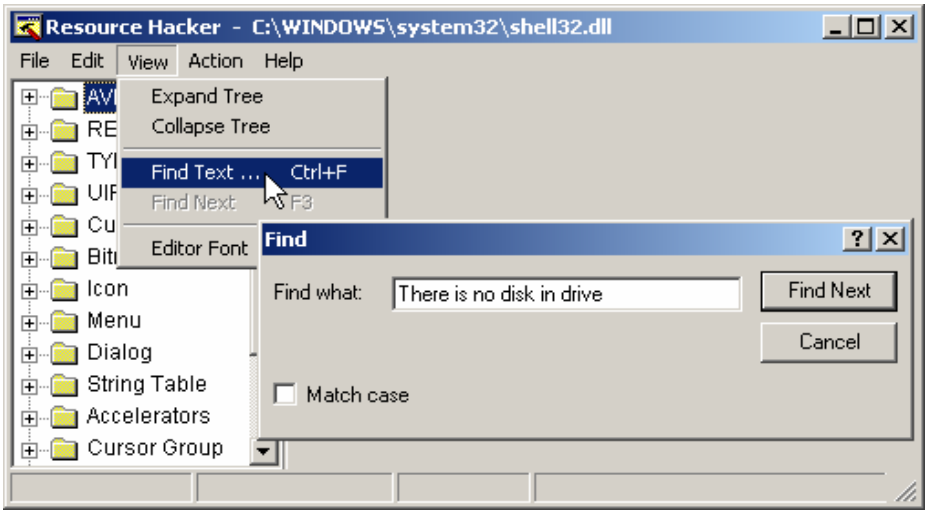
در این قسمت از قابلیت‌های نرم‌افزارهای Resource Tuner (PE Explorer), Resource Hacker به‌منظور ایجاد تغییرات در منابع فایل‌های اجرایی استفاده خواهیم کرد.

تغییر در منابع رشته ای

این‌گونه ازتغییرات معمولاً به‌منظور تغییر زبان و رشته‌های نمایش داده شده توسط فایل‌های اجرایی استفاده می‌شود. به این منظور ابتدا فایل موردنظر را با استفاده از برنامه Resource Hacker باز کرده و با استفاده از گزینه Find Text همانند شکل (٦-١) رشته موردنظر خود را در منابع رشته‌ای فایل اجرایی جستجو کنید.

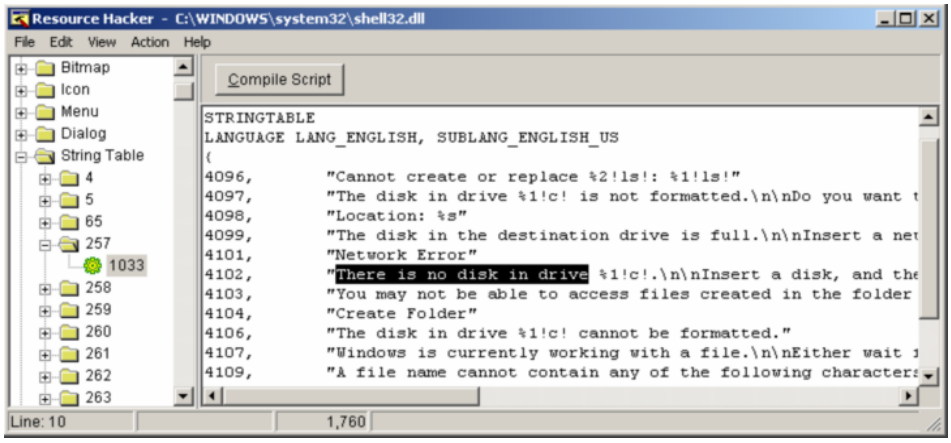
نسخه 3.4این نرم‌افزار در CD ضمیمه موجود می‌باشد

Tools\ ResourceHacker



شکل (٦-١)

پس از پیدا کردن رشته موردنظر و ایجاد تغییرات در آن، با استفاده از دکمه Compile script تغییرات اعمال شده را به‌طور موقت ذخیره کرده و در صورت نیاز برای اِعمال آنها در فایل اجرایی از گزینه Save استفاده کنید.



شکل (۶-۲)

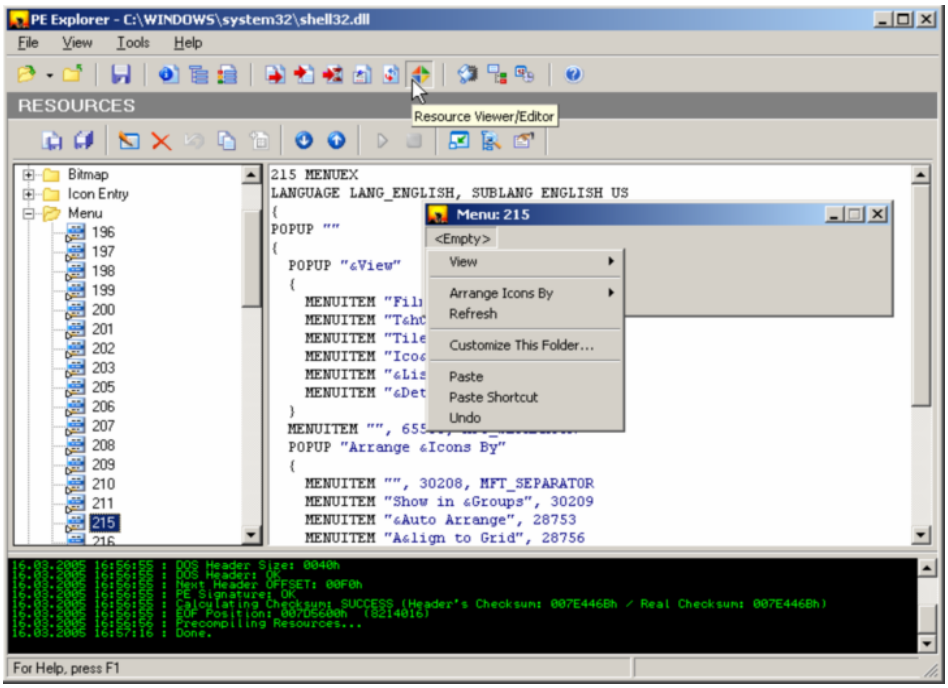
تغییر در منوها

موارد کاربرد این‌گونه تغییرات نیز مشابه قسمت قبل است. در این قسمت از امکانات نرم‌افزار PE Explorer به‌منظور ایجاد این گونه تغییرات استفاده می‌کنیم.

نسخه 1.97 این نرم‌افزار در CD ضمیمه موجود می‌باشد

Tools\PEExplorer

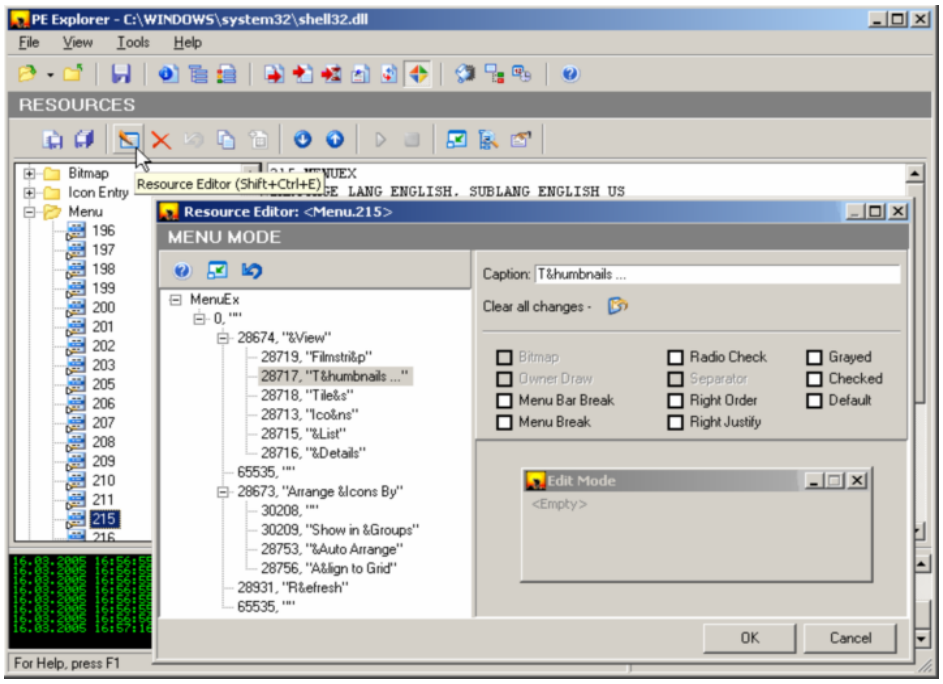
به این منظور ابتدا فایل اجرایی موردنظر را توسط نرم‌افزار PE Explorer بازکرده و سپس دکمه Resource viewer را از نوار ابزار کلیک کنید. با این عمل منابع موجود در فایل اجرایی موردنظر همانند شکل (۶-۳) در قسمت Resource نمایش داده خواهد شد.



شکل (۶-۳)

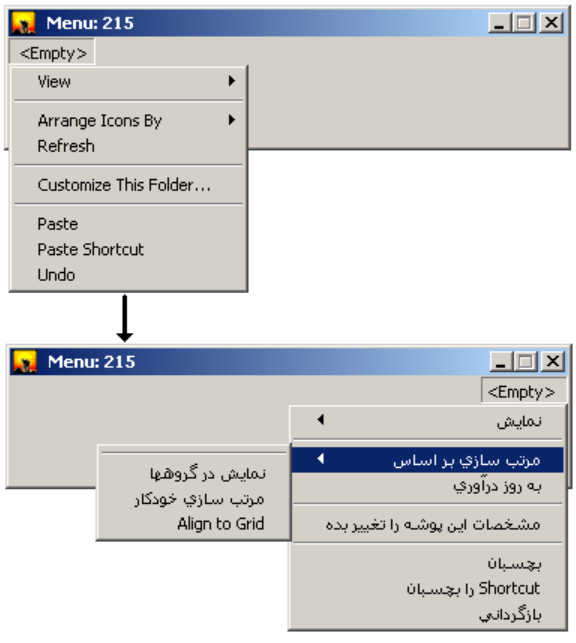
از لیست سلسله مراتبی نمایش داده شده در قسمت چپ شاخه Menu را انتخاب کرده و به‌منظور جستجو در منوهای موجود، کلیدهای Ctrl+F را فشار دهید. با این عمل پنجره Find نمایش داده شده و امکان جستجو برروی رشته‌های داخلی موجود در منوها را فراهم می‌کند.

پس از پیدا کردن منوی موردنظر در لیست منوهای موجود در منابع فایل اجرایی، به‌منظور ایجاد تغییرات برروی آن می‌توانید دکمه Resource Editor را از نوار ابزار کلیک کنید. با این عمل پنجره Resource Editor همانند شکل (۶-۴) نمایش داده شده و امکان ایجاد تغییر در خصوصیات و شکل ظاهر منوی موردنظر را ایجاد می‌کند.



شکل (٦-٤)

به عنوان مثال با ایجاد تغییر در رشته Caption و خصوصیات Right Order و Right Justify می‌توانید منوهای موجود در نرم‌افزار موردنظر را فارسی کنید. به عنوان نمونه به شکل (٦-٥) توجه کنید.



شکل (٦-٥)

به‌منظور اعمال تغییرات ایجاد شده برروی فایل اجرایی موردنظر پس از کلیک برروی دکمه Ok در پنجره Resource Editor از گزینه save File as واقع در منوی File استفاده کنید.

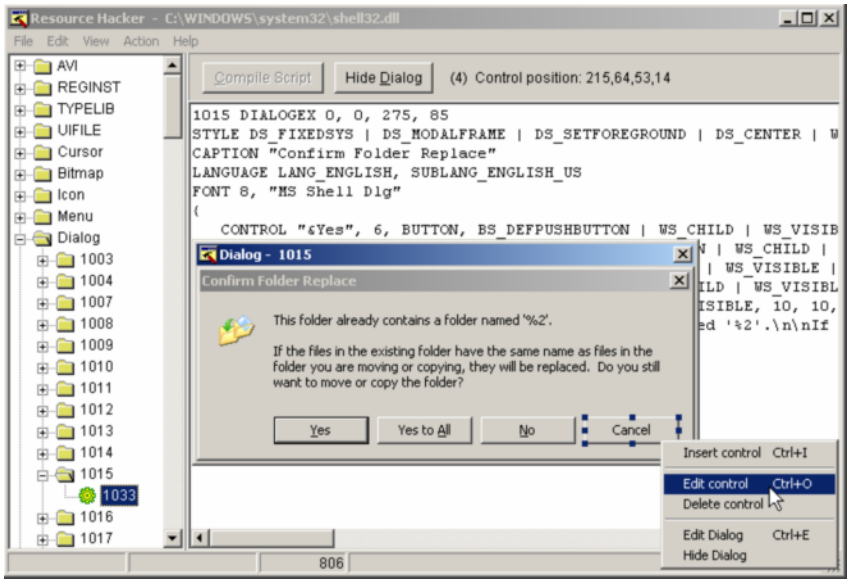
تغییر در پنجره‌ها و دیالوگ‌ها

همان‌طور که می‌دانید پنجره‌ها و دیالوگ‌ها نقش اصلی را در رابط کاربرد برنامه‌های تحت ویندوز بازی می‌کنند. در نتیجه با ایجاد تغییر برروی آنها می‌توانید شکل ظاهری نرم‌افزار موردنظر را در صورت نیاز تغییر دهید. در ادامه به نحوه ایجاد تغییرات برروی پنجره‌ها و دیالوگ‌های ذخیره شده در فایل‌های اجرایی ساخته شده توسط کامپایلرهای Visual C++ ، Delphi / C++ Builder و Visual Basic خواهیم پرداخت. توجه داشته باشید که به‌طور معمول پنجره‌ها و دیالوگ‌ها به عنوان جزئی از منابع فایل‌های اجرایی محسوب می‌شوند ولی در برخی از وارد ممکن است این اجزاء به صورت پویا و در هنگام اجرا توسط برنامه موردنظر ایجاد شوند. تغییر این گونه از پنجره‌ها و دیالوگ‌ها بسیار پیچیده بوده و نیاز به بررسی کامل و تغییر برروی کدهای فایل اجرایی دارد که در این قسمت مورد بررسی قرار نمی‌گیرند.

Visual C++ (دیالوگ های استاندارد)

به‌طور معمول برنامه‌های نوشته شده به وسیله Visual C++ از دیالوگ‌های استاندارد به‌منظور مدیریت پنجره‌ها استفاده می‌کنند همان‌طور که می‌دانید مدیریت رویدادها، هدایت آنها و سایر جزئیات مربوط به این دیالوگ‌های استاندارد به‌صورت پیش‌فرض برعهده سیستم عامل (ویندوز) است در نتیجه برنامه‌نویسان نیازی به کنترل جزئیات آنها ندارند. همین امر سبب سهولت استفاده از آنها می‌گردد.

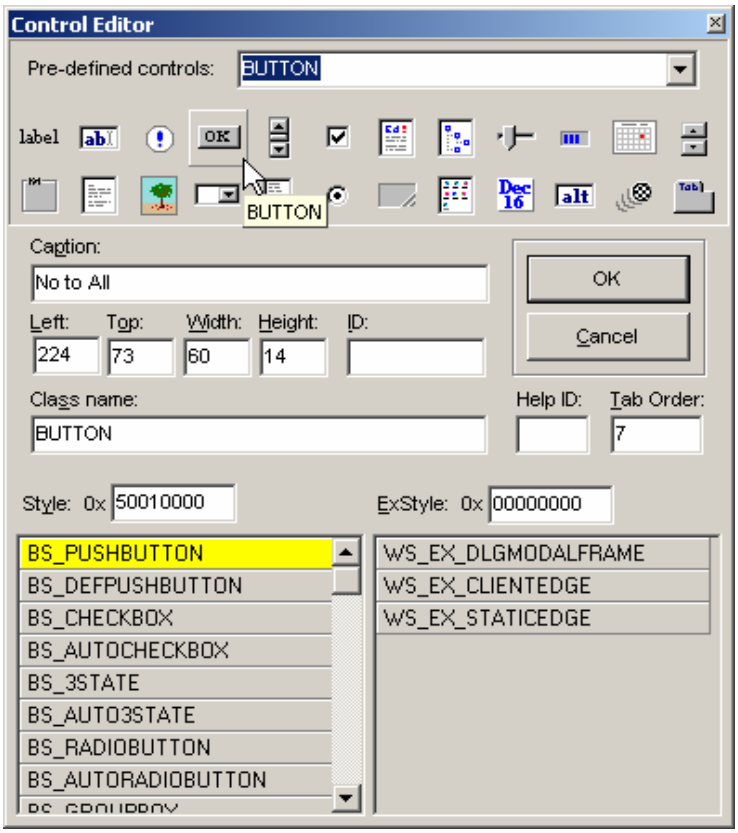
در این قسمت از نرم‌افزار Resource Hacker به‌منظور ایجاد تغییرات برروی دیالوگ‌های استاندارد ذخیره شده در فایل‌های اجرایی استفاده می‌کنیم. به این منظور ابتدا فایل اجرایی موردنظر را باز کرده و پس از انتخاب دیالوگ موردنظر از قسمت Dialog برروی پنجره نمایش داده شده (کنترل موردنظر) کلیک راست کنید. همان‌طور که در شکل (۶-۶) مشاهده می‌کنید با این عمل، منوی ایجاد تغییرات نمایش داده می‌شود.



شکل (۶-۶)

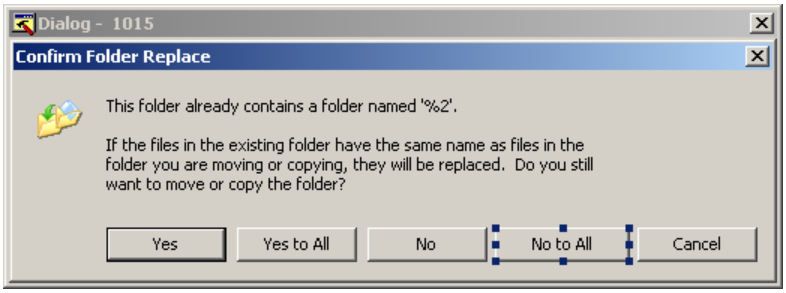
Insert Control

با استفاده از این گزینه پنجره Control Editor همانند **شکل (۶-۷)** نمایش داده شده و امکان تعریف و ایجاد یک کنترل جدید را برروی دیالوگ انتخاب شده می‌دهد. همان‌طور که مشاهده می‌کنید کنترل اضافه شده در مثال، یک دکمه با عنوان No to All است.



شکل (۶-۷)

در شکل (۶-۸) ظاهر نهایی دیالوگ مثال را پس از اضافه شدن کنترل جدید و انجام تغییراتی در مکان کنترل‌ها مشاهده می‌کنید.



شکل (۶-۸)

Edit Control

با استفاده از این گزینه می‌توانید تغییراتی را در خصوصیات کنترل موردنظر ایجاد کنید.

Edit Dialog

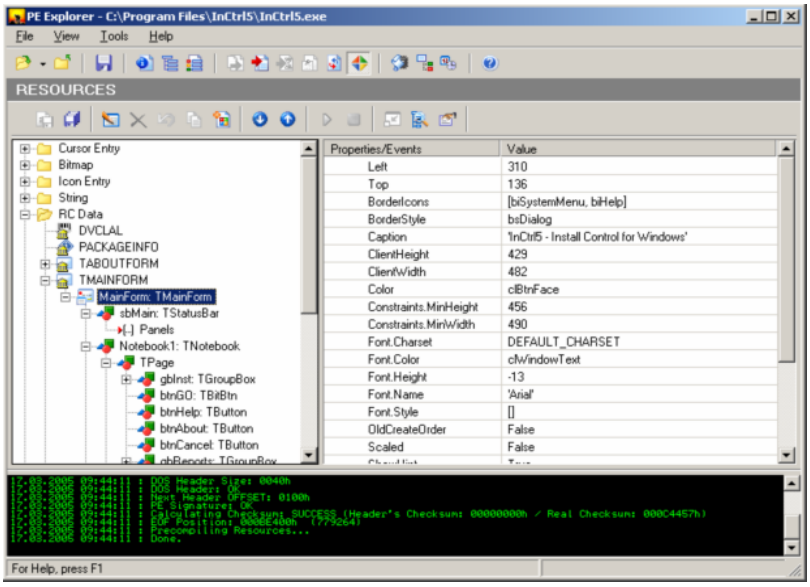
با استفاده از این گزینه می‌توانید تغییراتی را در خصوصیات دیالوگ موردنظر اعمال کنید.

پس از ایجاد تغییرات، به‌منظور اعمال آنها در فایل اجرایی ابتدا دکمه Compile Script را کلیک کرده و سپس فایل موردنظر را save کنید.

Delphi / C++ Builder

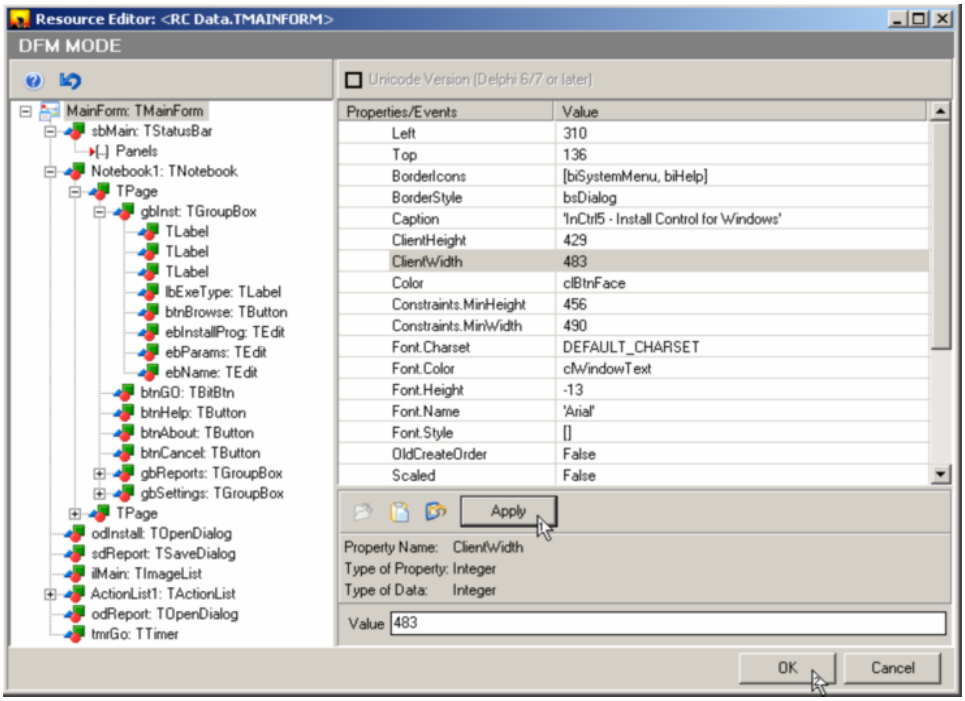
معمولاً کامپایلرهای ساخت شرکت Borland از ساختارهای خاص خود به‌منظور ذخیره و مدیریت رابط کاربر و پنجره‌ها استفاده می‌کنند. این داده‌ها در قسمت RC Data از منابع فایل اجرایی ذخیره می‌شوند.

در این قسمت از نرم‌افزار PE Explorer به‌منظور ایجاد تغییرات برروی این نوع از فایل‌های اجرایی استفاده می‌کنیم. به این منظور ابتدا فایل اجرایی موردنظر را باز کرده و در قسمت RC Data از Resource Viewer پنجره موردنظر را انتخاب کنید با این عمل لیست خصوصیات آن همانند شکل (۶-۹) در سمت راست ظاهر خواهد شد.



شکل (۶-۹)

در صورت نیاز به‌منظور ایجاد تغییرات برروی پنجره موردنظر برروی دکمه Resource Editor از نوار ابزار کلیک کنید. با این عمل پنجره Resource Editor همانند شکل (٦-۱۰) ظاهر می‌شود. با انتخاب کنترل موردنظر از لیست سلسله مراتبی کنترل‌های موجود در پنجره و تغییر در خصوصیات آن کنترل می‌توانید تغییرات دلخواه خود را در شکل ظاهری و ساختار پنجره موردنظر اعمال نمایید.



شکل (٦-۱۰)

توجه داشته باشید که پس از انجام تغییرات در قسمت Value از خصوصیات کنترل موردنظر، به‌منظور ثبت تغییرات از دکمه Apply استفاده کرده و در صورت لزوم به‌منظور اعمال تغییرات برروی فایل اجرایی، پس از تأیید تغییرات از گزینه Save file as از منوی File استفاده کنید.

Visual Basic

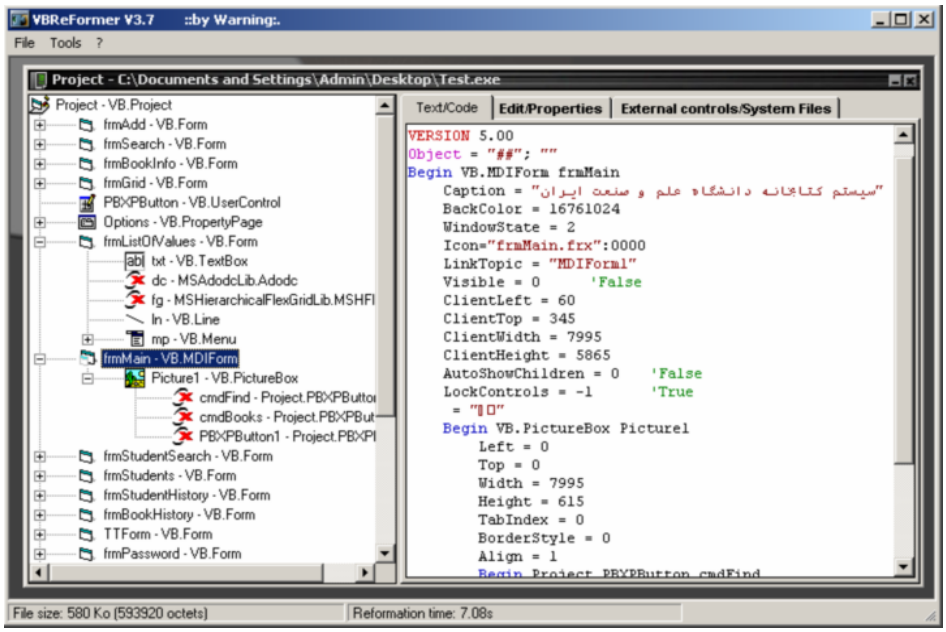
VB نیز همانند کامپایلرهای Borland از ساختارهای خاص خود به‌منظور ذخیره‌سازی و مدیریت رابط کاربر استفاده می‌کند. به‌طور معمول اطلاعات مربوط به پنجره‌ها و رابط کاربرد در قسمت منابع فایل‌های اجرایی تولید شده توسط VB قرار ندارند.

در این قسمت از قابلیت‌های نرم‌افزار VB Reformer به‌منظور ایجاد تغییرات برروی پنجره‌ها و رابط کاربر فایل‌های اجرایی VB استفاده می‌کنیم.

نسخه 4.1 این نرم‌افزار در CD ضمیمه موجود می‌باشد

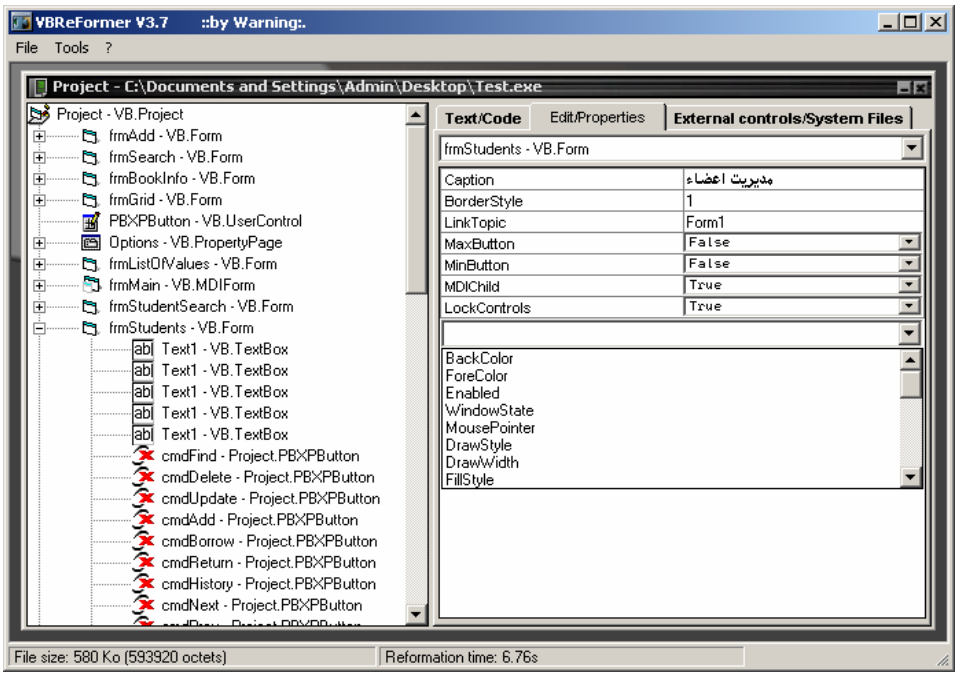
Tools\ VBReformer

به این منظور ابتدا فایل اجرایی موردنظر را باز کرده و پنجره موردنظر را از لیست پنجره‌ها (Form) انتخاب کنید. با این عمل در سمت راست اطلاعات خام مربوط به پنجره مذکور همانند شکل (۶-۱۱) نمایش داده خواهد شد.



شکل (۶-۱۱)

در صورت نیاز برای ایجاد تغییرات بر روی مشخصات و جزئیات کنترل‌ها و اجزاء گرافیکی موجود در فایل اجرایی، پس از انتخاب کنترل و یا جزء موردنظر، بخش Edit / Properties را همانند شکل (۶-۱۲) فعال کنید. با استفاده از این بخش می‌توانید تغییرات دلخواه خود را برروی خصوصیات جزء موردنظر عمال نمایید.



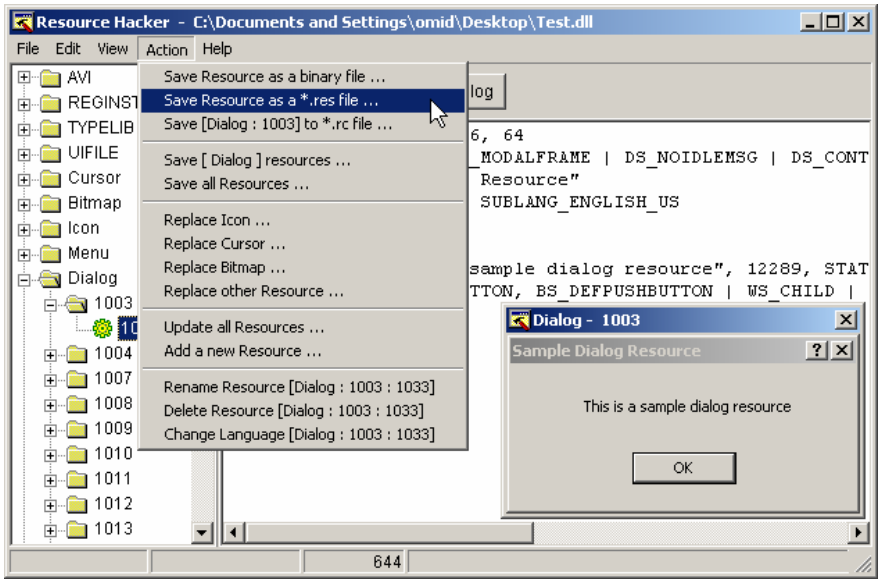
شکل (۶-۱۲)

پس از انجام تغییرات موردنظر برای اعمال آنها برروی فایل اجرایی می‌توانید از گزینه Save binary as واقع در منوی File استفاده کنید.

اضافه کردن منابع جدید به فایل‌های اجرایی

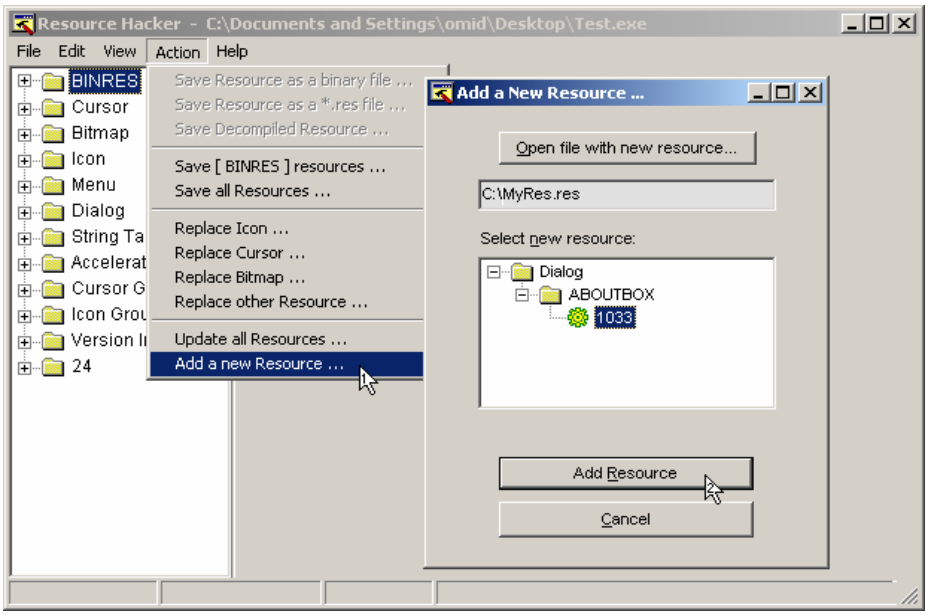
معمولاً در مراحل اضافه کردن کدها وقابلیت‌های جدید به برنامه‌ها و فایل‌های اجرایی، منابع جدیدی نیزاز جمله Dialogها، تصاویر، رشته‌ها و... مورد نیاز است که باید به قسمت منابع فایل و یا فایل‌های اجرایی موردنظر اضافه شوند. به‌طور معمول منابع جدید مورد نیاز از فایل‌های اجرایی دیگر به دست می‌آیند ولی در صورت نیاز می‌توانید منابع موردنظر خود را با استفاده از ابزارهای در نظر گرفته شده توسط کامپایلرهای گوناگون تحت ویندوز از جمله Visual C++ و یا Delphi کامپایل کرده و از آنها استفاده کنید. همان‌طور که می‌دانید در کامپایلرهای ویندوز منابع قبل از عملیات پیوند و ساخت فایل اجرایی نهایی به صورت یک فایل با پسوند res کامپایل می‌شوند. در حقیقت این فایل شامل کلیه منابع در نظر گرفته شده برای فایل اجرایی نهایی است. در ادامه از این فایل‌ها به‌منظور اضافه کردن منابع جدید به فایل‌های اجرایی استفاده خواهیم کرد.

در این قسمت از قابلیت‌های نرم‌افزار Resource Hacker به‌منظور اضافه کردن منابع جدید به فایل‌های اجرایی استفاده خواهیم کرد. برای سادگی در مثال این بخش منابع جدید مورد نیاز را از یک فایل اجرایی دیگر تهیه خواهیم کرد. در نتیجه ابتدا فایل اجرایی حاوی منابع مورد نیاز را توسط Resource Hacker باز کرده و منبع موردنظر را انتخاب کنید. سپس همانند شکل (۶-۱۳) از گزینه Save Resource as a *.res file استفاده کنید.



شکل (۶-۱۳)

پس از ذخیره فایل res، فایل اجرایی موردنظر را باز کرده و گزینه Add a new Resource را انتخاب کنید. با این عمل پنجره Add a new Resource همانند شکل (٦-١٤) نمایش داده می‌شود. با استفاده از گزینه Open فایل res تولید شده در مرحله قبل را انتخاب کنید. همان‌طور که مشاهده می‌کنید نام منبع ذخیره شده در فایل res در قسمت Select new resource نمایش داده می‌شود.



شکل (٦-١٤)

پس از انتخاب منبع موردنظر، Add resource را انتخاب کرده و به‌منظور ذخیره تغییرات در فایل اجرایی از گزینه‌های save واقع در منوی File استفاده کنید. حال منبع شما به فایل اجرایی اضافه شده و کدها و دستورالعمل‌های موجود در فایل اجرایی و یا فایل‌های اجرایی دیگر می‌توانند از آن استفاده کنند.

توجه داشته باشید در صورتی که منبع موردنظر شما یک تصویر Bitmap و یا یک Icon است، نیازی به ایجاد فایل res ندارید. به این منظور پس از باز کردن فایل اجرایی موردنظر گزینه Add a new Resource را انتخاب کرده و فایل Icon و یا Bitmap موردنظر را تعیین کنید. در قسمت Resource Name از پنجرهٔ Add a new Resource نامی را برای منبع جدید در نظر گرفته و منبع را اضافه کنید.

توجه داشته باشید که از نام‌های تعیین شده برای منابع جدید به‌منظور دسترسی به آنها در مراحل بعدی استفاده خواهد شد.

ایجاد تغییرات در مشخصات و ساختار فایل های اجرایی

فایل‌های اجرایی در ویندوز ازساختمان‌های دادهٔ پیچیده‌ای به‌منظور ذخیره و مدیریت اطلاعات مربوط به ساختارها و جزئیات داخلی خود استفاده می‌کنند. داشتن اطلاعات کافی در مورد ساختار این فایل‌ها و نحوه عملکرد آنها یکی از مهم‌ترین موارد درمرحله ایجاد تغییرات محسوب می‌شود. به‌منظور کسب اطلاعات دقیق‌تر راجع به ساختار و نحوه عملکرد فایل‌های اجرایی می‌توانید به فصل ۸ مراجعه کنید.

در موارد گوناگونی از جمله اضافه کردن sectionها و کدهای جدید به فایل‌های اجرایی، اضافه کردن توابع و dllهای مورد نیاز به جدول توابع ورودی و بسیاری از موارد دیگر، نیاز به تغییر سرآیندها و اطلاعات کنترلی ذخیره شده در فایل اجرایی پیدا می‌کنیم. در این فصل ابزارهایی را به‌منظور ایجاد سریع‌تر این‌گونه تغییرات معرفی کرده و نحوه عملکرد آنها را نیز مورد بررسی قرار خواهیم داد.

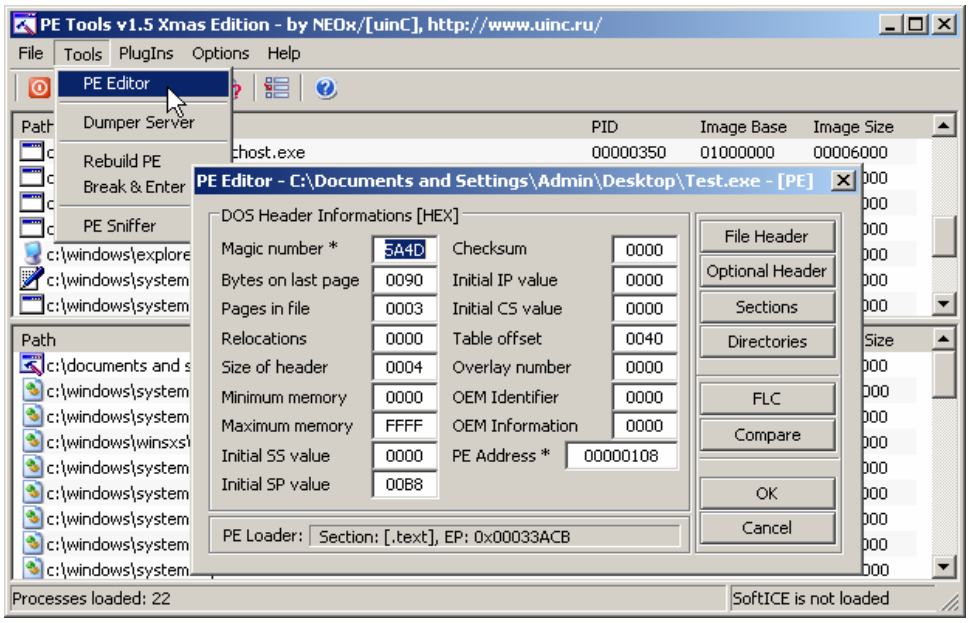
تغییر در اطلاعات سرآیندها

در این قسمت از قابلیت‌های نرم‌افزار PE Tools به‌منظور ایجاد تغییرات در سرآیندهای فایل‌های اجرایی استفاده خواهیم کرد.

نسخه 1.54 این نرم‌افزار در CD ضمیمه موجود می‌باشد

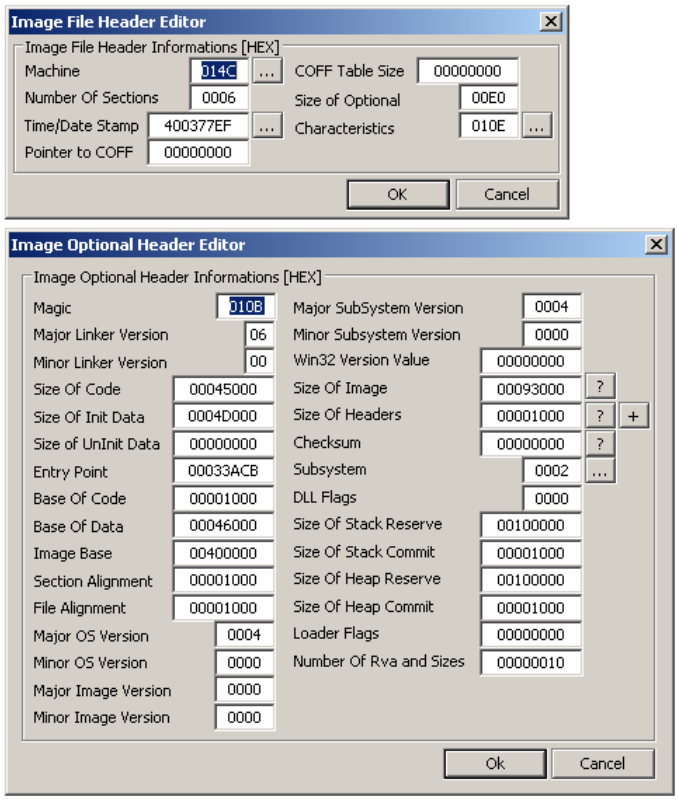
Tools\ PETools

به این منظور پس از انتخاب گزینه PE Editor از منوی Tools فایل اجرایی موردنظر را انتخاب کنید. با این عمل پنجره PE Editor همانند شکل (٦-١٥) باز شده و جزئیات قسمت Dos Header از فایل اجرایی موردنظر رابه نمایش می‌گذارد. در صورت نیاز می‌توانید تغییرات دلخواه خود را در مقایر اجزاء موردنظر از این سرآیند انتخاب کنید.



شکل (٦-١٥)

در صورت نیاز به‌منظور ایجاد تغییرات در سرآیندهای Image File Header و Image Optional Header می‌توانید از دکمه‌های File Header و Optional Header استفاده کنید. درشکل (٦-١٦) این نرم‌افزار را درحال نمایش اطلاعات و اجزاء این سرآیندها مشاهده می‌کنید.



شکل (۶-۱۶)

اضافه کردن dll ها و توابع در لیست توابع ورودی

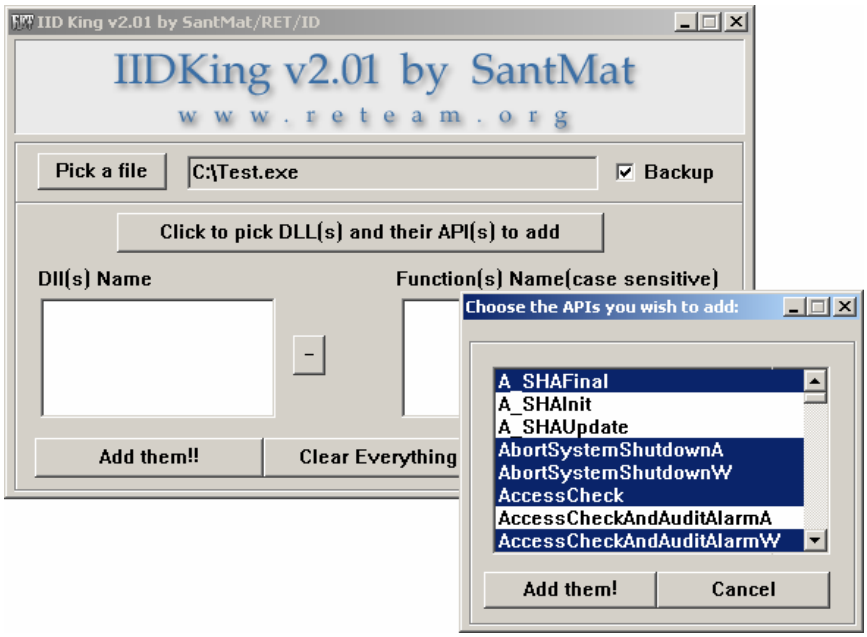
امکان اضافه کردن dllها و توابع مورد نیاز به لیست توابع ورودی فایل اجرایی، سبب بارگذاری فایل dll موردنظر در هنگام بارگذاری فایل اجرایی شده و امکان استفاده از توابع داخلی dll مذکور را توسط کدهای فایل اجرایی به شما می‌دهد. معمولاً از این روش به‌منظور اضافه کرن قابلیت‌های جدید به نرم‌افزارهای موجود استفاده می‌شود زیرا در اکثر موارد فضای اضافی کافی برای درج کدهای جدید در فایل‌های اجرایی وجود ندارد.

در این قسمت از قابلیت‌های نرم‌افزار و IIDKing به‌منظور اضافه کردن توابع و dllها به لیست توابع ورودی فایل‌های اجرایی استفاده خواهیم کرد.

نسخه 2.01 این نرم‌افزار در CD ضمیمه موجود می‌باشد

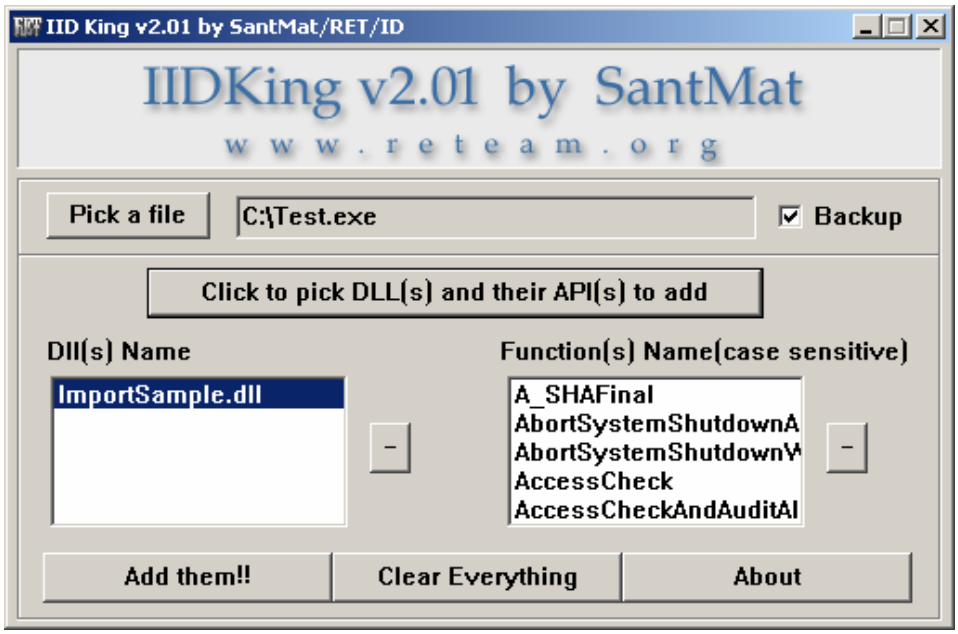
Tools\ IIDKing

به این منظور پس از کلیک برروی دکمه Pick File و انتخاب فایل اجرایی یا dll موردنظر برای مشخص کردن dll ورودی برروی دکمه Pick dlls کلیک کرده و فایل dll موردنظر را انتخاب کنید. با این عمل پنجره Choose API همانند شکل (۶-۱۷) ظاهر شده و لیستی از توابع صادر شده (Export Functions) توسط dll مذکور را به نمایش می‌گذارد. توابع مورد نیاز خود را از لیست نمایش داده شده انتخاب کرده و برروی دکمه add کلیک کنید.



شکل (۶-۱۷)

همان‌طور که در شکل (۶-۱۸) مشاهده می‌کنید نام dll و توابع انتخاب شده، در لیست صفحه اصلی نمایش داده می‌شوند. در صورت نیاز می‌توانید به‌منظور اضافه کردن dllها و توابع دیگر مراحل فوق را تکرار کنید.



شکل (۶-۱۸)

پس از تعیین کلیه dllها و توابع ورودی مورد نیاز، به‌منظور اعمال تغییرات برروی فایل اجرایی و یا dll موردنظر، برروی دکمه Add them کلیک کنید.

اضافه‌کردن توابع داخلی فایل‌های اجرایی به لیست توابع صادر شده (Export Table)

یکی از ساده‌ترین و مطمئن‌ترین روش‌ها به‌منظور دسترسی و استفاده از کدها و توابع داخلی و خصوصی فایل‌های اجرایی، اضافه کردن آدرس ورودی این توابع به لیست توابع صادر شده از طرف این فایل‌ها است. بدیهی است که قبل از استفاده از این توابع داخلی لازم است بررسی‌های کامل و دقیقی به‌منظور شناسایی پارامترهای ورودی آنها و نحوه عملکرد هر یک از این پارامترها انجام شود.

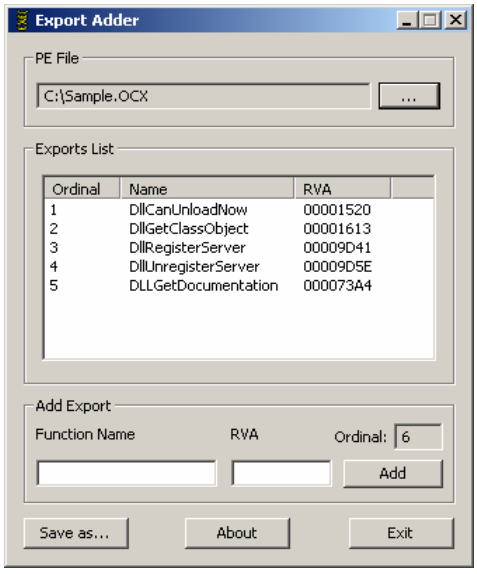
پس از اضافه کردن آدرس توابع داخلی موردنظر به قسمت Export Table از فایل اجرایی ویا dll موردنظر می‌توانید همانند سایر توابع استاندارد صادر شده از طرف dllهای سیستمی از آنها استفاده کنید.

در این قسمت از امکانات نرم‌افزار Export Adder به‌منظور اضافه کردن توابع دلخواه به لیست توابع صادر شده (Export table) از طرف فایل‌های اجرایی استفاده می‌کنیم.

نسخه 1.1 این نرم‌افزار در CD ضمیمه موجود می‌باشد

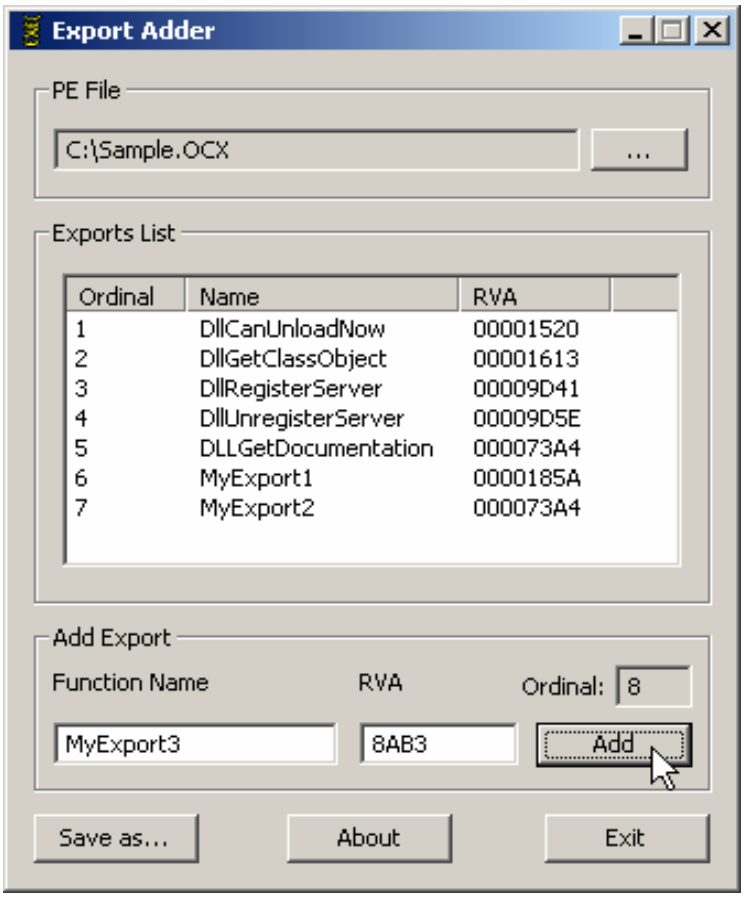
Tools\ ExportAdder

به این منظور ابتدا فایل اجرایی موردنظر را در قسمت PE File مشخص کنید همان‌طور که در شکل (۶-۱۹) نیز مشاهده می‌کنید پس از باز شدن فایل اجرایی موردنظر لیست توابع صادر شده از طرف این فایل در صورت وجود در قسمت Export List نمایش داده خواهد شد.



شکل (۶-۱۹)

برای اضافه کردن توابع جدید، ابتدا در قسمت Add Export نامی برای تابع موردنظر انتخاب کرده و سپس آدرس ورودی (مجازی) آن را در فایل اجرایی در قسمت RVA وارد کرده و دکمه Add را کلیک کنید. همان‌طور که مشاهده می‌کنید توابع تعریف شده از طرف شما به لیست Exports اضافه می‌شوند.



شکل (۶-۲۰)

پس از اضافه کردن توابع موردنظر می‌توانید از دکمه Save as به‌منظور ذخیره آنها در فایل اجرایی استفاده کنید.

اضافه کردن و یا تغییر Section ها در فایل های اجرایی

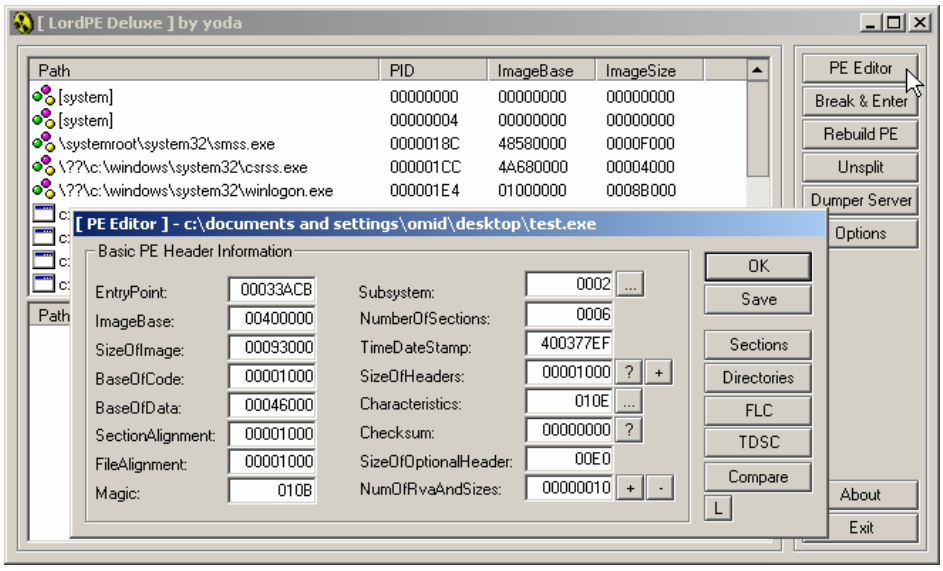
در مراحل ایجاد تغییرات در فایل‌های اجرایی، موارد متعددی وجود دارند که در آنها نیاز به اضافه کردن قسمت‌های جدیدی از کد، داده، منابع، سرآیندها و... به فایل اجرایی موردنظر پیدا می‌کنیم. همان‌طور که ذکر شد sectionهای جدید می‌توانند به‌منظور ذخیره‌سازی هرگونه داده جدید به کار گرفته شده و در مراحل اجرای برنامه توسط دستورالعمل‌های گوناگون آدرس‌دهی شوند.

در این قسمت از قابلیت‌های نرم‌افزار Lord PE به‌منظور اضافه کردن و ایجاد تغییر برروی خصوصیات sectionها استفاده خواهیم کرد.

نسخه Delux این نرم‌افزار در CD ضمیمه موجود می‌باشد

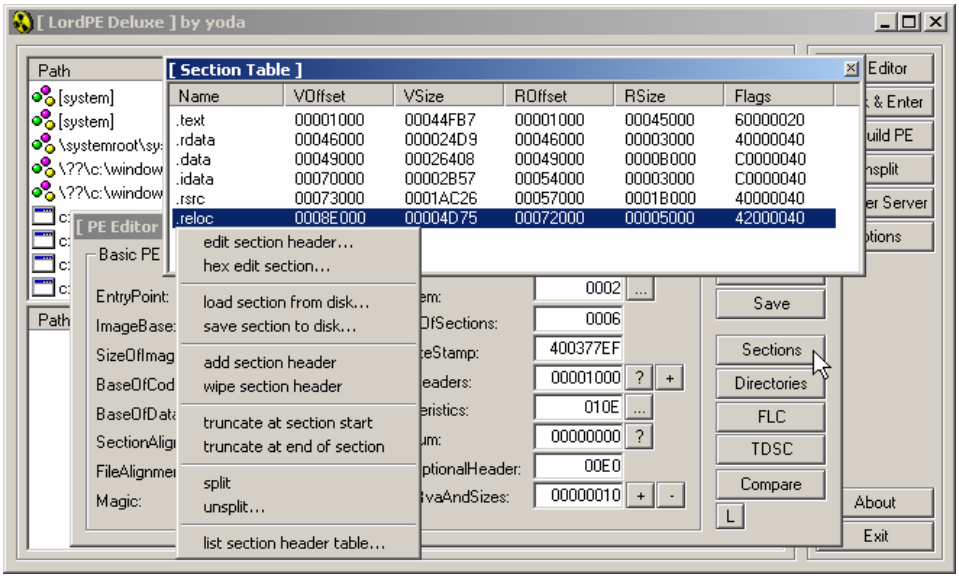
Tools\ LordPE

به این منظور پس از کلیک برروی دکمه PE Editor فایل اجرایی موردنظر را انتخاب کنید. با این عمل پنجره PE Editor همانند شکل (۶-۲۱) ظاهر شده و برخی از اجزاء مهم سرآیندهای فایل را به نمایش می‌گذارد. در صورت نیاز می‌توانید تغییرات دلخواه خود را نیز در مقادیر اجزاء نمایش داده شده ایجاد کنید.



شکل (۶-۲۱)

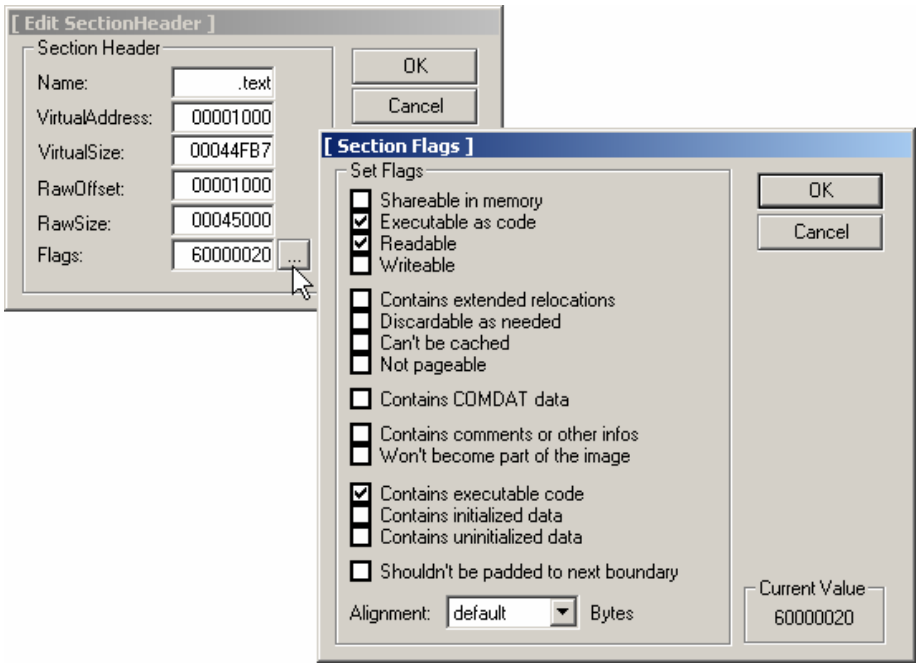
به‌منظور ایجاد تغیرات و یا اضافه کردن section جدید، گزینه sections را انتخاب کنید. با این عمل پنجره Section Table ظاهر شده و لیست Section های فعلی موجود در فایل اجرایی موردنظر را به نمایش می‌گذارد. همان‌طور که در شکل (۶-۲۲) مشاهده می‌کنید در منوی این پنجره گزینه‌های متعددی به‌منظور بررسی و یا تغییر Section های فایل اجرایی در نظر گرفته شده است.



شکل (۶-۲۲)

Edit Section Header

با استفاده از این گزینه می‌توانید خصوصیات section موردنظر را تغییر دهید. همان‌طور که در شکل (۶-۲۳) مشاهده می‌کنید این خصوصیات عبارتند از نام، آدرس شروع مجازی در فایل اجرایی، اندازه مجازی، اندازه واقعی، آدرس شروع حقیقی در فایل اجرایی و یک Flag که خصوصیات section را تعیین می‌کند.

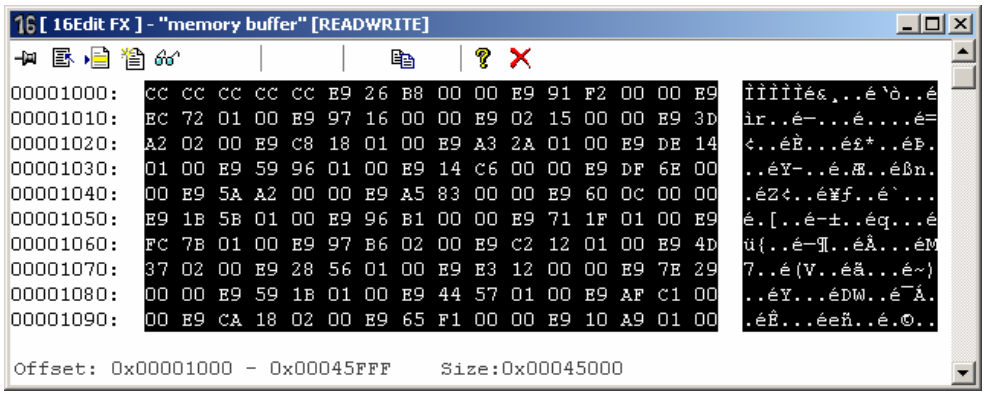


شکل (٦-٢٣)

همان‌طور که مشاهده می‌کنید این فلگ خصوصیات section را از جمله قابلیت اجرا، خواندن، نوشتن و بافر شدن تعیین می‌کند. با استفاده از گزینه‌های موجود در پنجره Section Flags، می‌توانید این خصوصیات را تغییر دهید.

Hex Edit Section

با استفاده از این گزینه می‌توانید توسط یک ادیتور Hex محدوده section موردنظر را همانند شکل (٦-٢٤) مشاهده کرده و در صورت نیاز عملیات جستجو و یا تغییر را برروی آن انجام دهید. گزینه‌های جستجوی موجود در این صفحه از رشته‌های Unicode و ASCII پشتیبانی می‌کنند.



شکل (٦-٣٤)

Load Section From Disk

با استفاده از این گزینه می‌توانید یک section ذخیره شده در فایل را به section های فعلی اضافه کنید و یا به عبارت دیگر یک فایل را به عنوان یک section جدید به section های فایل اجرایی موردنظر اضافه کنید.

می‌توانید به‌منظور شکستن یک فایل اجرایی به چند فایل جداگانه برحسب section ها وسرآیندها، از گزینه Split استفاده کنید.

Save section to disk

اطلاعات section فعلی را به عنوان یک فایل جداگانه ذخیر می‌کند.

Add Section Header

اطلاعات کنترلی لازم برای ایجاد یک section جدید را به سرآیندهای فایل اجرایی اضافه می‌کند.

Wipe Section Header

اطلاعات کنترلی مربوط به section فعلی را از سرآیندهای فایل اجرایی پاک می‌کند.

ایجاد تغییرات در کدهای فایل‌های اجرایی

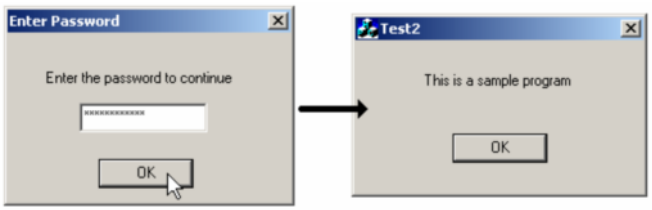
همان‌طور که می‌دانید به‌طورکلی یک فایل اجرایی از کدها و داده‌ها تشکیل شده است که کامپایلرهای گوناگون معمولاً آنها را در sectionهای جداگانه‌ای قرار می‌دهند. با توجه به این نکته که روند اجرایی و مدیریت اطلاعات فایل‌های اجرایی توسط کدهای داخلی آن انجام می‌گیرد ایجاد تغییرات در این کدها می‌تواند روند اجرایی و واکنش‌های برنامه موردنظر را به کلی تغییر دهد.

بدیهی است به‌منظور انجام تغییرات در قسمت‌های کد، اطلاعات کاملی راجع به ساختار و نحوه علمکرد نرم‌افزار موردنظر مورد نیاز است. نحوه به دست آوردن این اطلاعات را در قسمت‌های قبل مورد بررسی قرار داده‌ایم.

این مرحله معمولاً به‌عنوان مرحله نهایی در اکثر عملیات مهندسی معکوس محسوب می‌گردد که نیازمند داشتن اطلاعات کافی و دقت بسیار زیاد است. در این قسمت روش‌های ایجاد تغییرات در کدهای فایل‌های اجرایی را با استفاده از یک مثال ساده مورد بررسی قرار خواهیم داد. بدیهی است که ایجاد تغییرت در فایل‌های اجرایی بزرگتر و پیچیده‌تر نیازمند انجام تحلیل‌ها و بررسی‌های بیشتر و پیچیده‌تری نیز خواهد بود که جزء با تمرین و کسب تجربه‌های کافی به دست نخواهد آمد.

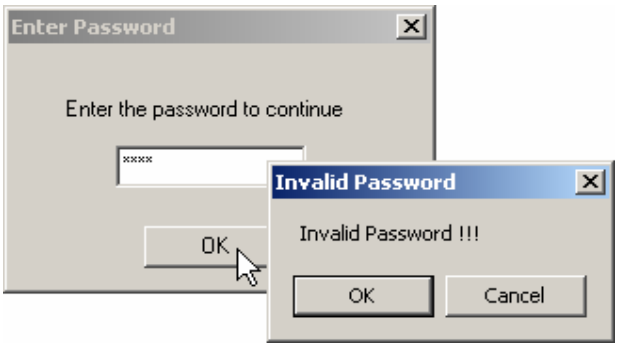
ایجاد تغییرات به‌صورت ایستا

در مثال این قسمت برنامه ساده‌ای داریم که قبل از اجرا کلمه عبور از کاربر دریافت کرده و در صورت صحیح بودن صفحه اصلی خود را نمایش خواهد داد.



شکل (٦-۲۵)

با اجرای نرم‌افزار و وارد کردن کلمه عبور نادرست، با پیغام خطایی به‌صورت زیر مواجه خواهید شد.

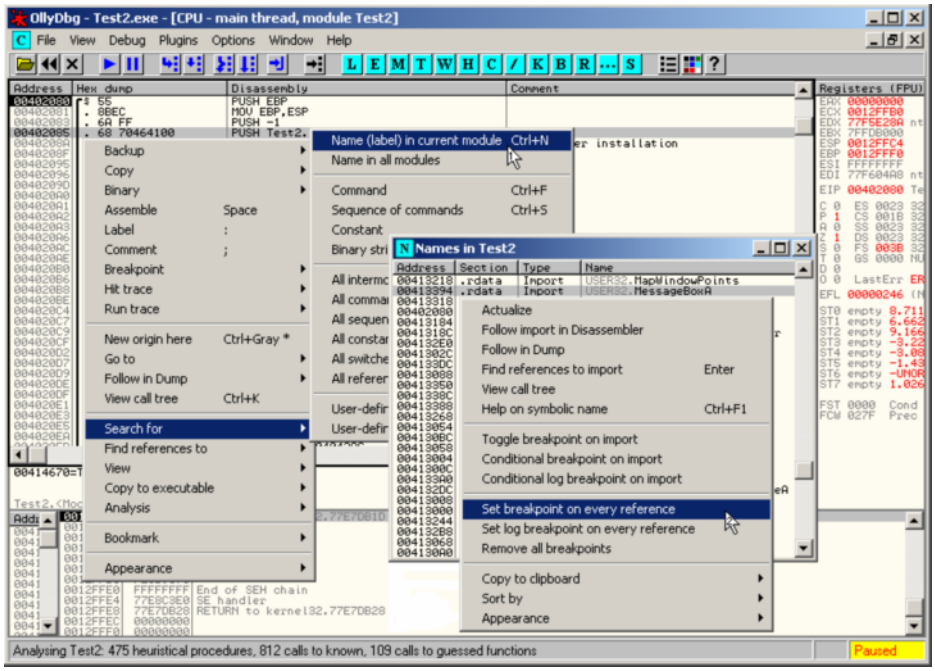


شکل (٦-٢٦)

حال می‌خواهیم روند اجرایی و کدهای این برنامه را به نحوی تغییر دهیم که بدون چک کردن صحت کلمه عبور و با هر کمله عبور دلخواهی اجازه ورودبه صفحه اصلی را به ما بدهد.

در مراحل ایجاد تغییرات هدف نهایی به دست آوردن آدرس مناطق موردنظر از کدهای فایل اجرایی و اعمال تغییرات برروی آنها است. همان‌طور که در قسمت Debug بررسی شد، به‌منظور پیدا کردن محدوده‌ها و آدرس‌های کنترلی موردنظر، توسط Debugger ها روند اجرایی را مورد تحلیل و بررسی قرار می‌دهیم. یکی از مهم‌ترین موارد در شروع عملیات دیباگ انتخاب نقطه و یا نقاط شروع مناسب و یا به عبارتی انتخاب نقاط توقف مناسب در ابتدای این عملیات است. به دلیل استفاده برنامه‌های تحت ویندوز از توابع استاندارد API، به‌طور معمول بسته به شرایط از این توابع به‌منظور انتخاب این نقاط شروع استفاده می‌شود. بدیهی است که انتخاب صحیح این نقاط تأثیر مستقیمی بر موفقیت عملیات مراحل بعدی خواهد داشت و این امر نیز جز با تمرین و کسب تجربه دست یافتنی نیست. با توجه به نحوه عملکرد سیستم امنیتی این برنامه، توابع API , MessageBox و یا GetWindowText می‌توانند به عنوان نقاط شروع، انتخاب‌های مناسبی باشند.

حال با استفاده از نرم‌افزار OllyDbg عملیات دیباگ را برای فایل اجرایی این مثال آغاز می‌کنیم. همان‌طور که در فصل ٤ بررسی شد OllyDbg امکاناتی را به‌منظور ایجاد نقاط توقف برای فراخوانی‌های توابع API تدارک دیده است که همانند شکل (٦-٢٧) از آنها به‌منظور ایجاد این نقاط برای فراخوانی‌های تابع MessageBoxA استفاده خواهیم کرد.



شکل (۶-۲۷)

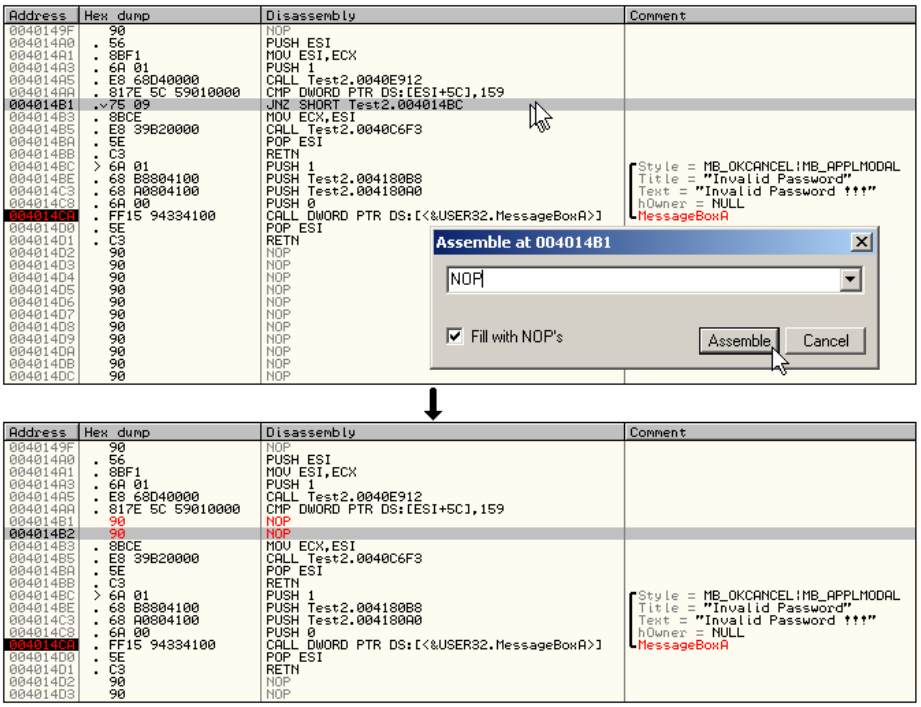
پس از ایجاد این نقاط توقف برنامه مثال را در مد دیباگ اجرا کرده و کلمه عبور نادرستی را وارد می‌کنیم. همان‌طور که در شکل (۶-۲۸) مشاهده می‌کنید پس از فشردن دکمه OK، روند اجرایی توسط یک نقطه توقف در آدرس 4014CA متوقف می‌شود که یک دستورالعمل فراخوانی تابع MessageBoxA است.

```
0040149E    90              NOP
0040149F    90              NOP
004014A0  . 56              PUSH ESI
004014A1  . 8BF1            MOV ESI,ECX
004014A3  . 6A 01           PUSH 1
004014A5  . E8 68D40000     CALL Test2.0040E912
004014AA  . 817E 5C 59010000 CMP DWORD PTR DS:[ESI+5C],159
004014B1  .v75 09           JNZ SHORT Test2.004014BC
004014B3  . 8BCE            MOV ECX,ESI
004014B5  . E8 39B20000     CALL Test2.0040C6F3
004014BA  . 5E              POP ESI
004014BB  . C3              RETN
004014BC  > 6A 01           PUSH 1                          Style = MB_OKCANCEL|MB_APPLMODAL
004014BE  . 68 B8804100     PUSH Test2.004180B8             Title = "Invalid Password"
004014C3  . 68 A0804100     PUSH Test2.004180A0             Text = "Invalid Password !!!"
004014C8  . 6A 00           PUSH 0                          hOwner = NULL
004014CA  . FF15 94334100   CALL DWORD PTR DS:[<&USER32.MessageBoxA>] MessageBoxA
004014D0  . 5E              POP ESI
004014D1  . C3              RETN
004014D2    90              NOP
```

شکل (۶-۲۸)

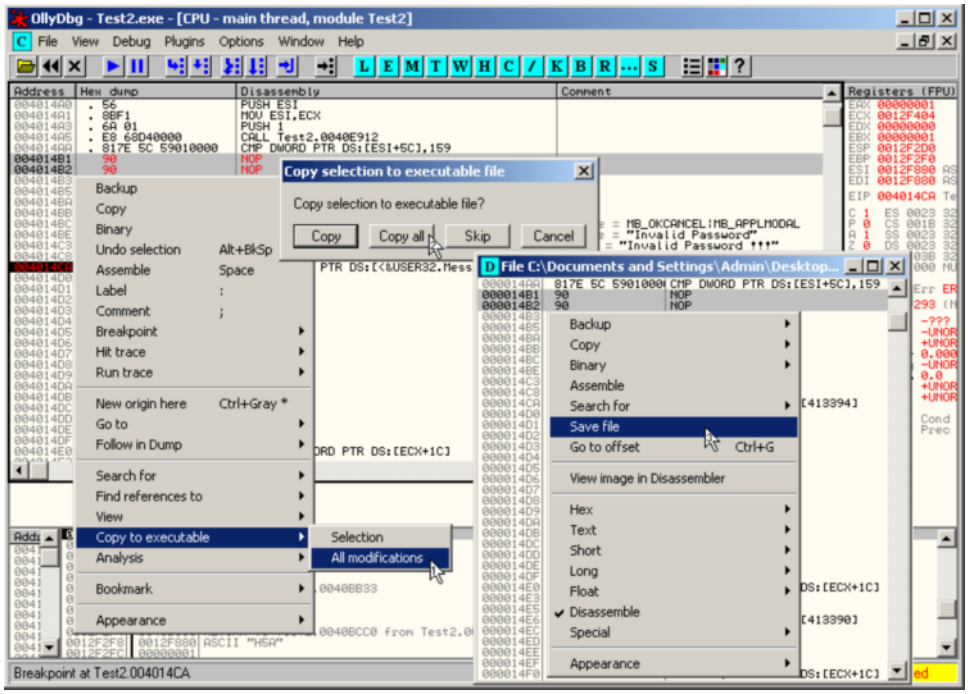
با نگاهی به دستورالعمل‌های ما قبل از آدرس 4014CA متوجه یک دستورالعمل پرش شرطی JNZ 4014BC در آدرس 4014B1 خواهیم شد. این دستورالعمل در حقیقت وظیفه بررسی صحت کلمه عبور وارد شده را برعهده داشته و در صورت نادرست بودن کلمه عبور به آدرس 4014BC

پرش خواهد کرد. با جایگزین کردن این دستورالعمل پرش شرطی، با دستورالعمل پرسش قطعی (JMP) معادل، بدون در نظر گرفتن کلمه عبور همیشه پیغام خطا دریافت خواهیم کرد. بدیهی است که با جایگزین کردن این دستورالعمل با دستورالعمل‌های بی‌اثر (NOP) هیچ گاه پرش صورت نگرفته و پیغام خطا دریافت نخواهیم کرد و برنامه به درستی اجرا خواهد شد. به این منظور در آدرس 4014B1 در قسمت Disassembly ، Double Click می‌کنیم با این عمل پنجره Assemble همانند شکل (۶-۲۹) نمایش داده شده و امکان ایجاد تغییر در دستورالعمل فعلی را ایجاد می‌کند. توجه داشته باشید که طول Opcode دستورالعمل جایگزین شده نباید از طول دستورالعمل فعلی بیشتر باشد زیرا در این صورت دستورالعمل‌های بعدی تخریب خواهند شد.



شکل (۶-۲۹)

همان‌طور که مشاهده می‌کنید دستورالعمل پرش شرطی با دو دستورالعمل NOP جایگزین شده است. حال به‌منظور ذخیره سازی تغییرات اعمال شده در فایل اجرایی همانند شکل (۶-۳۰) از قسمت Copy to Excutable از منوی اصلی استفاده کنید. با این عمل فایل اجرایی جدیدی از روی فایل اولیه ساخته شده و تغییرات برروی آن اعمال می‌گردد که می‌تواند جایگزین فایل اصلی شود.



شکل (۶-۳۰)

با اجرای فایل ساخته شده متوجه خواهیم شد که برنامه مثال بدون بررسی و واکنش درمقابل کلمات عبور آنها را قبول کرده و به روند اجرایی خود ادامه می‌دهد.

ایجاد تغییرات در مراحل اجرا

در مواردی ممکن است ایجاد تغییرات به صورت ایستا (ذخیره تغییرات در فایل اجرایی) ممکن نبوده و یا بسیار سخت باشد. به‌عنوان مثال در اغلب فایل‌های اجرایی محافظت شده ذخیره تغییرات در فایل اجرایی به سادگی امکان‌پذیر نیست.

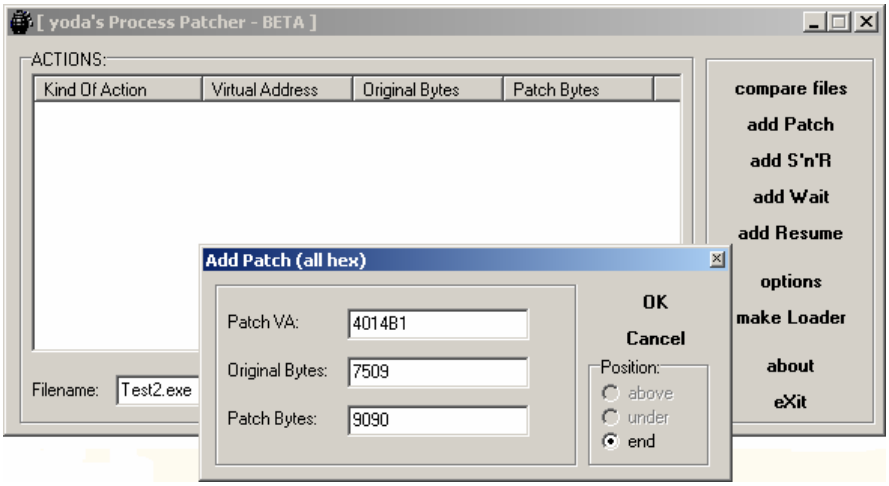
به‌منظور ایجاد تغییرات در این فایل‌ها، پس از بارگذاری تغییرات موردنظر را در حافظه برنامه اعمال می‌کنیم. به این منظور از ابزارهایی استفاده می‌کنیم که روند بارگذاری و ایجاد تغییرات را به سادگی مدیریت کرده و تغییرات موردنظر را اعمال می‌کنند. نتیجه نهایی این ابزارها در یک فایل اجرایی ذخیره می‌شود که حاوی اطلاعات کافی در مورد تغییرات مورد نیاز و نیز نحوه اعمال آنها در فایل اجرایی هدف می‌باشد. در نهایت این فایل‌های اجرایی به‌طور خودکار فایل اجرایی هدف را بارگذاری کرده و تغییرات را به‌صورت از پیش تعیین شده اعمال می‌نمایند.

در ادامه به‌منظور روشن‌تر شدن نحوه عملکرد این ابزارها از نرم‌افزار Y0da Process Patcher به‌منظور ایجاد یک Loader برای بارگذاری و ایجاد تغییرات در مثال قبل استفاده می‌کنیم.

نسخه 2000 این نرم‌افزار در CD ضمیمه موجود می‌باشد

Tools\Y0daProcessPatcher

به این منظور پس از مشخص کردن فایل اجرایی موردنظر در این نرم‌افزار، گزینه Add Patch را انتخاب کنید. با این عمل پنجره Add Patch همانند شکل (۶-۳۱) نمایش داده می‌شود.

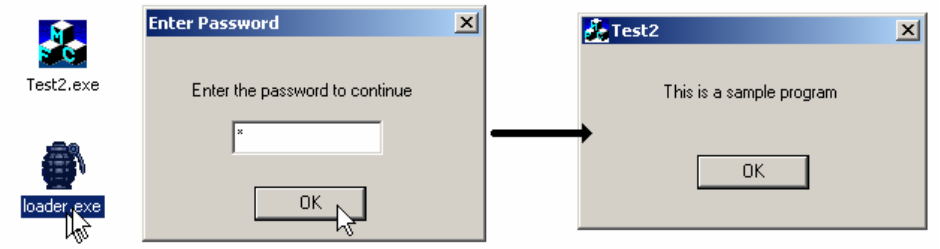


شکل (۶-۳۱)

اطلاعات مورد نیاز برای این مرحله به سه قسمت تقسیم شده‌اند:

- **RVA**: در این قسمت آدرس مجازی شرع محدوده موردنظر را مشخص می‌کنیم که در مثال آدرس 4014B1 است.
- **Original Bytes** : در این قسمت Opcodeهای خام محدوده موردنظر را به‌صورت Hex وارد می‌کنیم. این Opcodeها را می‌توانید در ستون Hex Dump از پنجره Disassembly از OllyDbg مشاهده کنید. در مثال، این کدها مقدار 7509 را دارند که معادل دستورالعمل پرش شرطی است.
- **Patch Bytes** : در این قسمت Opcode جایگزین را به‌صورت Hex وارد می‌کنیم که در مثال، 9090 می‌باشد که معادل دو دستورالعمل NOP است.

پس از اضافه کردن، گزینه Make Loader را انتخاب کنید. با این عمل یک فایل اجرایی با نام loader.exe در کنار فایل اجرایی اصلی ایجاد می‌شود. با اجرای Loader.exe به‌طور خودکار فایل اجرایی اصلی بارگذاری شده و سپس تغییرات موردنظر در حافظه آن اعمال می‌شود. که سبب تغییر کدهای فایل اجرایی در حافظه شده و روند اجرایی به نحو موردنظر تغییر پیدا می‌کند.



شکل (۶-۳۲)

اضافه کردن کدهای جدید به فایل‌های اجرایی

در بسیاری از موارد علاوه بر ایجاد تغییرات به‌منظور دستیابی به نتیجه مطلوب نیاز داریم که کدهای جدیدی به فایل اجرایی اضافه کرده و از آنها به‌منظور تغییر روند اجرایی و یا اضافه کردن قابلیت‌های جدید به نرم‌افزار موردنظر استفاده کنیم. بدیهی است که پس از افزودن این کدهای جدید می‌توان از آنها همانند کدهای اصلی فایل اجرایی استفاده کرده و در صورت نیاز توابع موجود در آنها را از داخل کدهای اصلی فراخوانی کنیم.

به این منظور کدهای مورد نیاز در فضاهای بلااستفاده فایل اجرایی قرار داده شده و یا sectionهای جدیدی ایجاد کرده و از آنها به‌منظور ذخیره‌سازی کدهای جدید استفاده می‌کنیم. این روش نسبتاً دشوار بوده و محدودیتهای زیادی دارد زیرا کدهای جدید مجبور هستند از منابع، فضای حافظه و جداول ورودی فایل اجرایی اصلی استفاده کنند که این امر سبب ایجاد حدودیت‌های بسیاری برای کدهای جدید خواهد شد.

به‌دلیل محدودیت‌های موجود، معمولاً از زبان‌های سطح بالا مانند Visual C++ ، Delphi به‌منظور ایجاد کدهای جدید استفاده نمی‌شود. در این قسمت، از کامپایلر Macro Assembler 6.11 و ابزار Code snippet Creator به‌منظور ایجاد کدهای جدید و اضافه کردن آنها به فایل‌های اجرایی موردنظر استفاده خواهیم کرد.

نسخه 6.11 این کامپایلر در CD ضمیمه موجود می‌باشد

Tools\ MacroAssembler

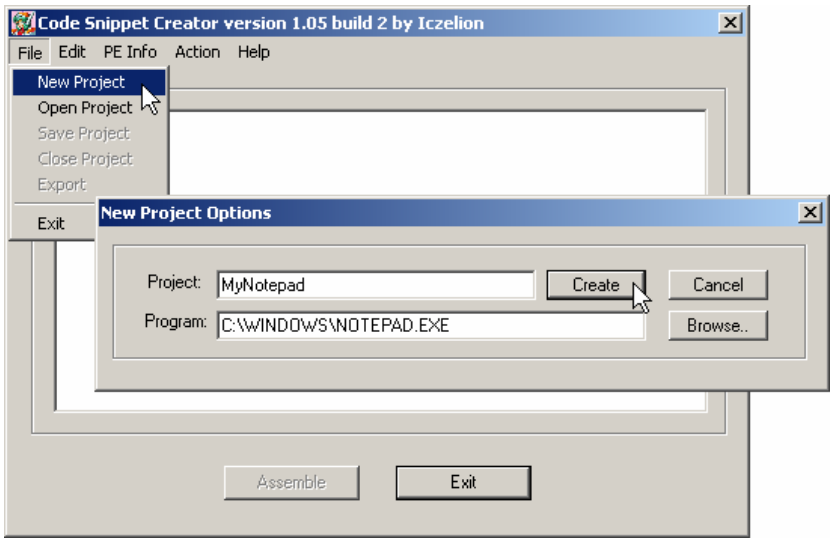
نسخه 1.05 این نرم‌افزار در CD ضمیمه موجود می‌باشد

Tools\ CodesnippetCreator

روش‌های برنامه‌نویسی در ویندوز با Macro Assembler در قسمت Assembly به‌طور کامل مورد بررسی قرار گرفته است. در صورت نیاز می‌توانید از آن به‌منظور کسب اطلاعات دقیق‌تر راجع به نحوه برنامه‌نویسی در این کامپایلر استفاده کنید.

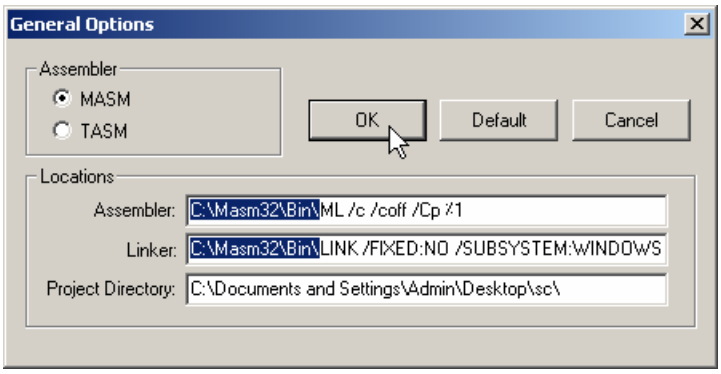
در مثال این بخش قصد داریم تکه کد ساده‌ای به فایل اجرایی Notepad.exe اضافه کنیم. این کد جدید سبب خواهد شد که در ابتدا، این برنامه پیغامی را نمایش داده و پس از تایید وارد برنامه اصلی

شود. به این منظور ابتدا برنامه Code Snippet Creator را اجرا کرده و پروژهٔ جدیدی را همانند شکل (٦-٣٣) ایجاد کنید.



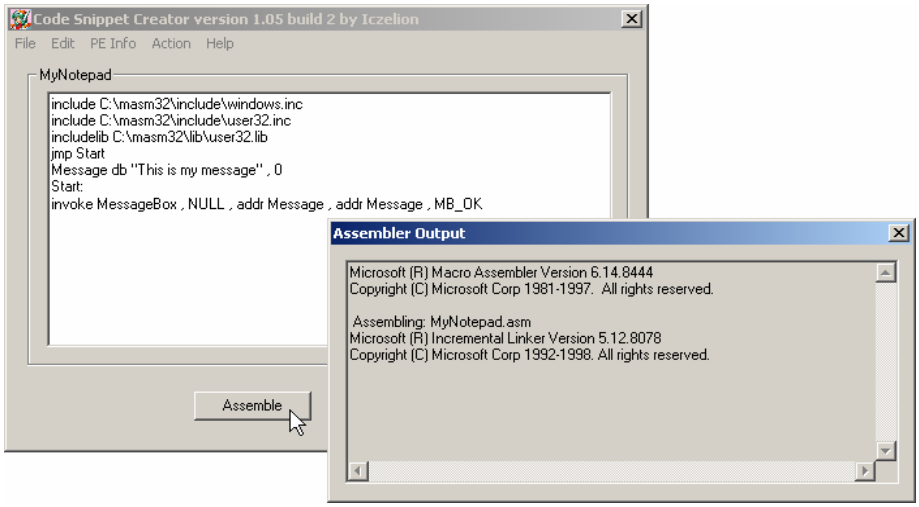
شکل (٦-٣٣)

حال که پروژهٔ جدید را ایجاد کردید می‌توانید کد موردنظر خود را که قصد دارید به فایل اجرایی Notepad.exe اضافه کنید مشخص کرده و در صفحه اصلی این نرم‌افزار وارد کنید. همان‌طور که ذکر شده به‌منظور کامپایل کردن کدهای موردنظر می‌توانید از Macro Assembler 6.11 و یا Turbo Assembler استفاده کنید. قبل از انجام عملیات Assemble باید مسیر و جزئیات کامپایلر خود را مشخص کنید. به این منظور از گزینه Options واقع در منوی Action استفاده کنید با این عمل پنجره General Options همانند شکل (٦-٣٤) نمایش داده شده و امکان ایجاد تغییر در مسیرها و پارامترهای Linker و Assembler را ایجاد می‌کند.



شکل (٦-٣٤)

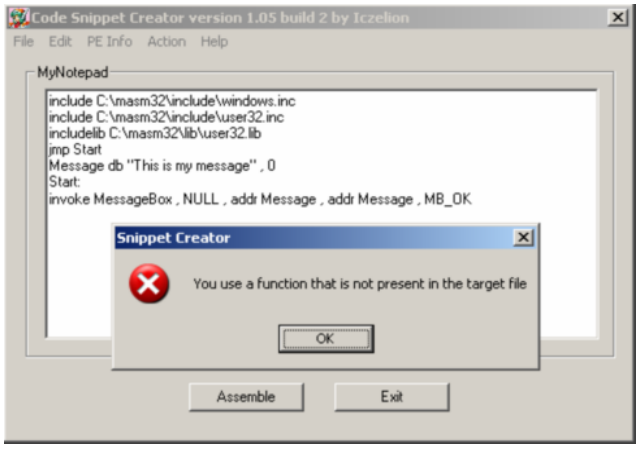
همان‌طور که مشاهده می‌کنید دراین مثال تغییراتی را درمسیرهای فایل‌های Linker.exe (Linker) و ML.exe (Assembler) ایجاد کرده‌ایم. پس از مشخص کردن پارامترها و جزئیات موردنظر به‌منظور ساخت فایل Obj مورد نیاز برای عملیات جایگذاری کد، از گزینه Assemble استفاده کنید. با این عمل پنجره Assembler Output همانند شکل (٦-٣٥) ظاهر شده و نتیجه عملیات Link و Assemble را به نمایش می‌گذارد.



شکل (٦-٣٥)

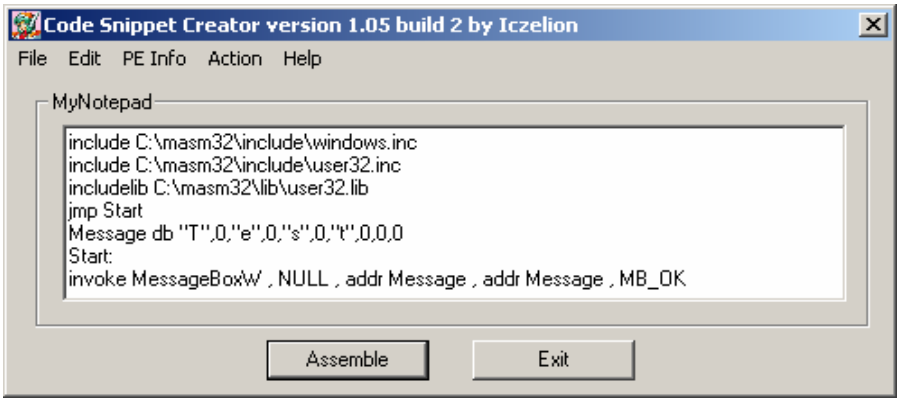
در صورت درست بودن برنامه نوشته شده، فایل Obj مربوطه ساخته شده و گزینه Export واقع در منوی File فعال می‌شود. پیش از اضافه کردن کدهای ایجاد شده به فایل اجرایی موردنظر، این کدها باید با فرمت خاصی ذخیره شوند. به این منظور از گزینه Export استفاده کرده و فایل Bin

نهایی را ایجاد کنید. توجه داشته باشید در صورتی که توابع API مورد استفاده در کدهای شما در لیست توابع ورودی فایل اجرایی موردنظر وجود نداشته باشد، فایل bin نهایی ایجاد نشده و خطایی به‌صورت زیر دریافت خواهید کرد.



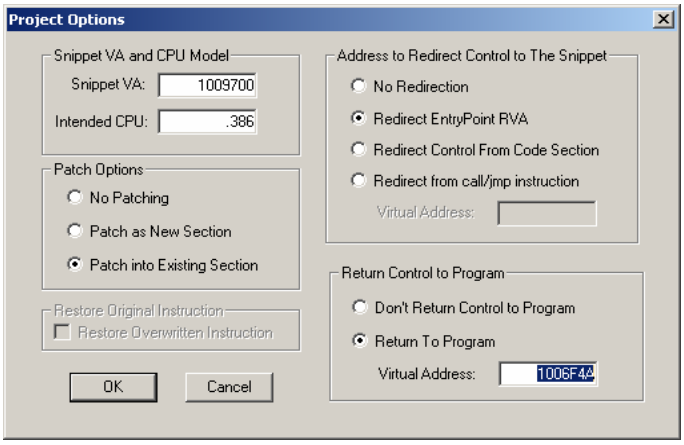
شکل (٦-٣٦)

با نگاهی به جدول توابع ورودی فایل Notepad.exe متوجه خواهید شد که تابع API مورد استفاده ما (MessageBoxA) در این لیست وجود ندارد ولی تابع MessageBoxW در این لیست وجود دارد و می توانیم از آن به جای تابع MessageBoxA استفاده کنیم همان‌طور که می‌دانید تابع MessageBoxW نسخه Unicode تابع MessageBoxA است. در نتیجه برنامه خود را به‌صورت زیر تغییر داده و پس از انجام عملیات Assemble فایل Bin نهایی را ایجاد می‌کنیم.



شکل (٦-٣٧)

قبل از اضافه کردن کدهای نهایی به فایل اجرایی باید با استفاده از گزینه Project Options نحوه اضافه کردن و جزئیات کد جدید را مشخص کنیم. با این عمل پنجره Project Options همانند شکل (۶-۳۸) نمایش داده می‌شود.



شکل (۶-۳۸)

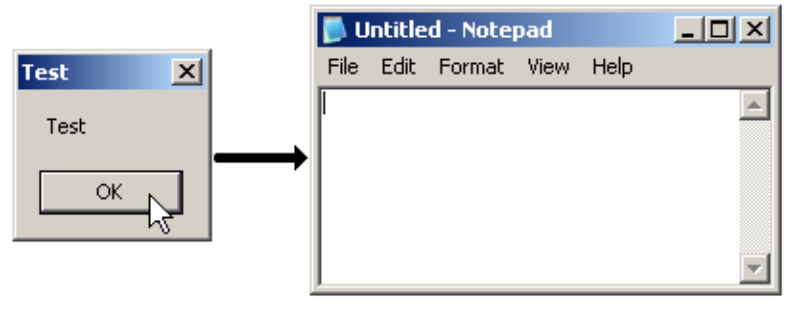
با نگاهی به Basic Header از فایل اجرایی Notepad.exe و کدهای Disassemble شده آن متوجه خواهیم شد که آدرس شروع (Entry Paint) برابر 1006F4A بوده و فضای خالی از کد و داده‌ای، در محدوده آدرس 1009700 قرار دارد که با مقادیر 0 پر شده است و می‌تواند به عنوان فضایی برای ذخیره سازی کدهای جدید به کار گرفته شود. به دلیل اینکه کدهای جدید تنها یک بار و آن هم در ابتدای برنامه اجرا خواهند شد، در صورتی که محدوده آدرس این کدها در مراحل اجرا توسط داده‌های برنامه در حافظه بازنویسی شوند، مشکلی برای روند اجرایی و کدهای جدید پیش نخواهد آمد.

با انتخاب گزینه Patch in to Existing section و مشخص کردن آدرس 1009700 در قسمت Snippet VA، کدهای جدید در فضای فعلی فایل اجرایی Notepad.exe و در آدرس مجازی 1009700 ذخیره خواهند شد.

با انتخاب گزینه Redirect EntryPoint RVA و مشخص کردن آدرس 1006F4A (EntryPoint فعلی)، آدرس ورودی فایل اجرایی Notepad.exe به آدرس شروع کدهای جدید تغییر داده شده و مشخص می‌شود که پس از اتمام کدهای جدید، روند اجرایی به آدرس شروع اصلی فایل اجرایی منتقل شده و مراحل اجرا همانند حالت عادی ادامه پیدا کند.

پس از مشخص کردن پارامترهای موردنظر به‌منظور اضافه کردن کدهای جدید به فایل اجرایی و اعمال تغییرات موردنظر، از گزینه Patch Target File واقع در منوی Action استفاده کنید.

در صورت صحت عملیات، با اجرای فایل Notepad.exe ابتدا پیغام Test نمایش داده شده و سپس پنجره اصلی این نرم‌افزار نمایش داده خواهد شد.



شکل (٦-۳۹)

فصل هفتم

درک کدهای اسمبلی

فصل هفتم

درک کدهای اسمبلی

همان‌طور که در قسمت‌های قبل متوجه شده‌اید، یکی از نکات کلیدی در مهندسی معکوس، توانایی شخص در درک و تحلیل درست کدهای Disassemble شده به‌وسیله نرم‌افزارهای Disassembler مانند IDA، W32Dasm و غیره است.

با توجه به اهمیت زیاد این موضوع، قسمت آخر این کتاب به نحوه برنامه‌نویسی به زبان اسمبلی در ویندوز اختصاص یافته است. این قسمت می تواند به‌منظور درک بهتر ساختارهای متداول مورد استفاده در نرم‌افزارها به کار گرفته شود. داشتن تجربه کافی دراین زمینه برای درک کدهای اسمبلی لازم نیست ولی در صورت وجود می تواند بسیار مفید بوده و شخص را در درک و تحلیل دقیق و صحیح آنها یاری دهد. توجه داشته باشید که تنها برنامه‌نویسان یک زبان می توانند تحلیل دقیق و درستی از کدهای نوشته شده به آن زبان داشته باشند.

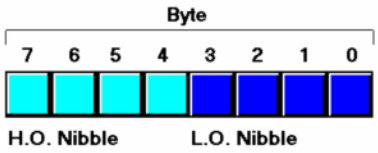
در این بخش ابتدا مروری بر برخی از مفاهیم کلیدی در برنامه‌های اسمبلی خواهیم داشت و سپس ساختارهای متداول مورد استفاده در زبان‌های سطح بالا را در مرحله کامپایل مورد بررسی دقیق قرار خواهیم داد.

سیستم های عددی

همان‌طور که می‌دانید کامپیوترها روش‌های گوناگونی را برای ذخیره و نمایش داده‌ها مورد استفاده قرار می دهند. در هریک از این روش‌ها از مجموعه از بیت‌ها برای ذخیره و نمایش داده‌هایی خاص استفاده می‌شود. در زیر به برخی از انواع استاندارد داده‌ها که در اکثر کامپیوترها از آنها استفاده می‌شود، اشاره خواهیم کرد. توجه داشته باشید نحوه عملکرد و داده‌های ذخیره شده در هر نوع، بستگی به تفسیر نرم‌افزار موردنظر دارد و نمی‌توان قانون خاصی را برای داده‌های ذخیره شده در آنها در نظر گرفت.

Nibble

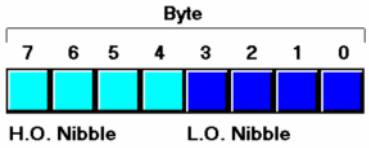
مجموعه‌ای از چهار بیت متوالی یک Nibble نامیده می‌شود که معمولاً برای ذخیره داده‌های BCD (Binary Coded Decimal) و یا Hexadecimal مورد استفاده قرار می‌گیرد. با توجه به این که چهار بیت می توانند ۱۶ حالت مختلف را ایجاد کنند، می‌توان آن را به‌منظور ذخیره و نمایش داده‌های هرگونه سیستمی که کمتراز ۱۶ حالت سازنده دارد مورد استفاده قرار داد.



شکل (۷-۱)

Byte

مجموعه از هشت بیت متوالی یک Byte نامیده می‌شود که می‌توان آن را به عنوان پراستفاده‌ترین نوع داده در کلیه کامپیوترها به حساب آورد. مهمترین نوع داده مورد استفاده در پردازنده‌های 80x86 بایت است. این پردازنده از آن به‌منظور آدرس‌دهی دستگاه‌های ورودی خروجی و حافظه اصلی استفاده می‌کند. در حقیقت کوچکترین قطعه اطلاعات که می‌تواند توسط این پردازنده‌ها مورد استفاده قرار بگیرد یک بایت است. همان‌طور که در شکل (۷-۲) مشاهده می‌کنید، بیت‌های تشکیل دهنده یک بایت به‌صورت زیر شماره‌گذاری می‌شوند که در آن بیت شماره 0 به‌عنوان کم ارزش‌ترین و بیت شماره 7 به عنوان پرارزش‌ترین بیت در نظر گرفته می‌شود.

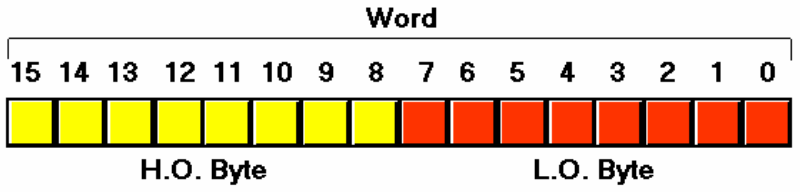


شکل (۷-۲)

با توجه به اینکه این ۸ بیت می‌توانند ۲۵٦ حالت مختلف را پوشش دهند، می‌توانند به‌منظور ذخیره و نمایش کاراکترهای اسکی، تصاویر نقشه بیتی و حتی فریم‌های صوتی مورد استفاده قرار بگیرند.

Word

مجموعه‌ای از ١٦ بیت متوالی یک word نامیده می‌شود که مانند شکل (۷-۳) شماره‌گذاری می‌شوند.



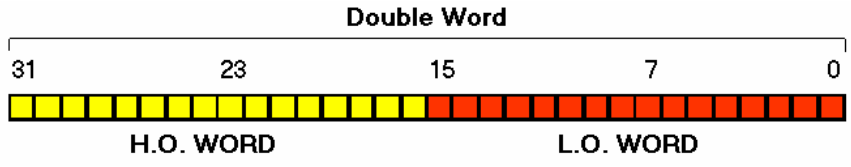
شکل (۷-۳)

همان‌طور که مشاهده می‌کنید، یک Word از دو بایت تشکیل شده است. به مجموعه بیت‌های 7 ... 0 بایت کم ارزش و به مجموعه بیت‌های 15...8 بایت پر ارزش یا High order Byte گفته می‌شود.

با توجه به این که ١٦ بیت می‌تواند 65536 حالت مختلف را پوشش دهند، از آنها به‌منظور ذخیره نوع داده عدری Integer در زبان‌های برنامه‌نویسی مختلف و یا کاراکترهای Unicode استفاده می‌شود.

Double Word یا DWord

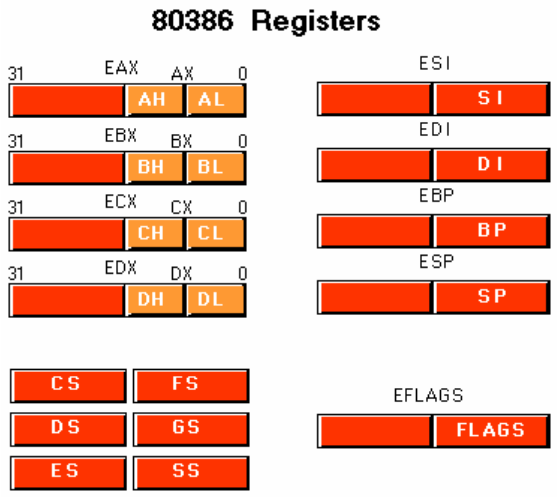
مجموعه‌ای از ۳۲ بیت یا دو word متوالی، DWord نامیده می‌شود. این مجموعه به‌صورت زیر شماره‌گذاری می‌شود. به مجموعه بیت‌های 15...0 ، word کم ارزش و به مجموعه بیت‌های 31...16، word پرارزش گفته می‌شود. درزبان‌های برنامه‌نویسی ازDWord به‌منظور ذخیره‌سازی انواع عددی Long و یا اعداد اعشاری Float استفاده می‌گردد.



شکل (۷-٤)

ثبات‌ها

همان‌طور که می‌دانید از ثبات‌ها در پردازشگرهای مختلف به‌منظور ذخیره‌سازی و استفاده از داده‌ها جهت انجام محاسبات، کنترل اجرای دستورات، دستیابی به آدرس‌های حافظه و I/O استفاده می‌شود. در شکل (۷-۵) ثبات‌های اصلی موجود در پردازنده‌های +80386 را مشاهده می‌کنید.



شکل (۷-۵)

ثبات‌ها برحسب روش معمول استفاده از آنها به گروه‌های زیر تقسیم‌بندی می‌شوند:

ثبات های عمومی

AX ، BX , EAX ، CX , EBX ، DX , ECX ، EDX ، ثبات‌های عمومی هستند که مبنای کاری سیستم محسوب می‌شوند. این ثبات‌ها، در این که می‌توانید تمام یا نیمی از فضای آنها را مورد استفاده قرار دهید منحصر به فرد هستند. ثبات‌های ۱۶ بیتی (AX , BX , CX , DX) به دو قسمت ۱ بایتی کم ارزش و پر ارزش تقسیم می‌شوند. به‌عنوان مثال ثبات AX شامل دو قسمت AH (پر ارزش) و AL (کم ارزش) می‌باشد که هر قسمت با نام آن قابل دسترسی است. پردازشگرهای 80386 و بالاتر علاوه بر این ثبات‌های ۱۶ بیتی، یک نسخه ۳۲ بیتی توسعه یافته آنها را نیز در خود دارند که عبارتند از: EAX , EBX , ECX , EDX.

ثبات EAX / AX

این ثبات در دستورالعمل‌های محاسباتی به‌عنوان عملگر اصلی محسوب می‌شود. علاوه بر آن از این ثبات در اعمال ورودی خروجی و برخی از اعمال رشته‌ای نیز استفاده می‌شود. در برخی از دستورالعمل‌ها در صورت استفاده از این ثبات، کد ماشین کوچکتری نسبت به سایر ثبات‌ها تولید می‌شود.

ثبات EBX / BX

از این ثبات در عملیات محاسباتی و نیز به عنوان شاخص برای توسعه آدرس‌دهی‌ها استفاده می‌گردد.

ثبات ECX / CX

معمولاً از این ثبات به‌عنوان شمارش گر در حلقه‌ها استفاده می‌شود. علاوه بر آن از این ثبات در برخی اعمال محاسباتی و شیفت‌ها استفاده می‌گردد.

ثبات EDX / DX

این ثبات به ثبات داده معروف است که برخی از اعمال ورودی ، خروجی نیازمند استفاده از آن هستند. در اعمال ضرب و تقسیم که مستلزم استفاده از مقادیر بزرگ هستند ، از جفت ثبات AX , DX و یا EAX , EDX استفاده می‌شود.

هر یک از ثبات‌های عمومی می‌توانند برای اعمال جمع و تفریق ۸ بیتی، ١٦ بیتی، و ۳۲ بیتی مورد استفاده قرار بگیرند.

ثبات‌های سگمنت

از این ثبات‌ها معمولاً به‌منظور آدرس‌دهی نواحی حافظه استفاده می‌شود.

ثبات CS

آدرس شروع سگمنت کد را در خود دارد. این آدرس به‌علاوه مقدار آفست در اشاره‌گر دستورالعمل (IP / EIP)، مشخص کننده آدرس دستورالعملی است که جهت اجرا از حافظه واکشی می‌شود.

ثبات DS

ثبات DS دارای آدرس شروع سگمنت داده‌ها است. به بیان ساده‌تر این آدرس به علاوه یک آفست در یک دستورالعمل، سبب ارجاع به مکان مشخصی از سگمنت داده‌ها می‌شود.

ثبات SS

این ثبات آدرس شروع پشته (Stack) را در خود دارد. این آدرس به علاوه یک آفست در ثبات اشاره‌گر پشته (SP/ESP) کلمه جاری در در پشته را آدرس‌دهی می‌کند.

ثبات ES

در برخی از اعمال رشته‌ای از این ثبات به‌منظور دستکاری آدرس‌دهی حافظه استفاده می‌شود.

ثبات های GS , FS

ثبات‌های سگمنت اضافی هستند که در پردازنده‌های 80386 و بالاتر در نظر گرفته شده‌اند.

ثبات‌های اشاره‌گر

ثبات اشاره گر دستورات (IP / EIP)

این ثبات حاوی آدرس آفست دستور بعدی جهت اجرا می‌باشد. همان‌طور که ذکر شد با استفاده از این آفست و آدرس شروع سگمنت کد (CS) آدرس دستورالعمل بعدی که باید از حافظه واکشی شود مشخص می‌گردد.

ثبات اشاره‌گر پشته (SP / ESP)

مقدار این ثبات یک آفست را فراهم می‌کند که وقتی با ثبات SS ترکیب شود، به کلمه جاری در پشته اشاره می‌کند.

ثبات اشاره‌گر پایه (BP / EBP)

این ثبات عمل ارجاع به پارامترها، شامل آدرس‌ها و داده‌هایی که از یک برنامه به پشته فرستاده می‌شوند را تسهیل می‌کند.

ثبات های شاخص

ثبات (SI / ESI)

این ثبات به‌عنوان شاخص مبدأ شناخته می‌شود که در برخی از اعمال رشته‌ای مورد نیاز است.

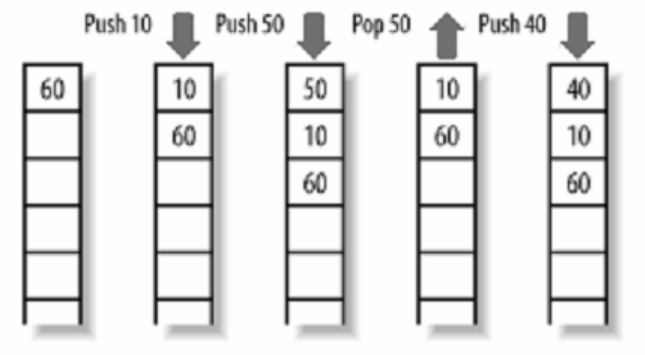
ثبات (DI/ EDI)

این ثبات به عنوان شاخص مقصد شناخته می‌شود که در برخی از اعمال رشته‌ای مورد نیاز است.

Stack

همان‌طور که می‌دانید میزان داده‌هایی که می‌تواند در ثبات‌های پردازنده ذخیره شود بسیار محدود است. به‌منظور غلبه بر این محدودیت از حافظه‌های جانبی مانند RAM و Cache به‌منظور ذخیره و مدیریت داده‌های مورد نیاز نرم‌افزارها در هنگام اجرا کمک گرفته می‌شود.

Stack یا پشته چیزی جز تکه ای از حافظه RAM نیست که می‌تواند توسط دو عمل pop , push اطلاعات مورد نیاز یک نرم‌افزار را ذخیره و بازیابی کند. در شکل (۷-٦) ساختار کلی Stack و نحوه عملکرد push , pop را مشاهده می‌کنید.



شکل (۷-٦)

همان‌طور که در شکل (۷-٦) مشاهده می‌کنید، همیشه داده‌ها به ابتدای stack اضافه می‌شوند. در نتیجه همیشه آخرین داده وارد شده اولین داده‌ای است که خارج خواهد شد. به این نوع ساختمان داده‌ها اصطلاحاً LIFO (Last In First Out) گفته می‌شود.

آدرس دهی‌ها در پردازنده‌های 80x86

در پردازنده‌های 80x86 روش‌های گوناگونی برای دسترسی و مدیریت حافظه در نظر گرفته شده است. این روش‌ها می‌توانند به‌منظور ایجاد و مدیریت ساختمان داده‌های مختلفی از قبیل متغیرها، آرایه‌ها، رکوردها، و سایر انواع دیگر به کار گرفته شوند. یادگیری انواع آدرس‌دهی‌ها، اولین قدم در یادیگری اسمبلی در 80x86 محسوب می‌شود.

- **آدرس‌دهی ثبات‌ها**

اکثر دستورالعمل‌های 80x86 می‌توانند برروی ثبات‌های عمومی عمل کنند. تنها کار لازم این است که از نام ثبات موردنظر به‌عنوان یکی ازعملوندهای دستورالعمل استفاده شود. در زیر نحوه به‌کارگیری دستورالعمل MOV را مشاهده می‌کنید.

```
MOV destination , source
```

این دستورالعمل داده‌ها را از عملوند مبدأ به مقصد کپی می‌کند. ثبات‌های 32,16,8 بیتی می‌توانند به‌عنوان عملوند این دستورالعمل به کار گرفته شوند. تنها نکته‌ای که باید مورد توجه قرار گیرد یکسان بودن اندازه دو عملوند مبدأ و مقصد است. در زیر چند نمونه از نحوه کاربرد آن را مشاهده می‌کنید.

```
Mov al , bl
```

```
Mov ax , cx
Mov ebx , edx
```

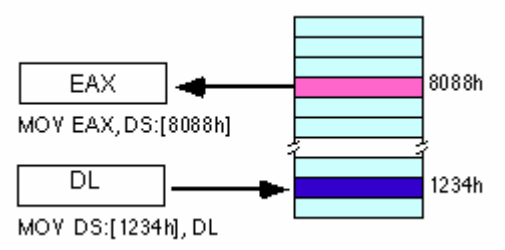
- **آدرس دهی حافظه**

درپردازنده‌های 80x86 و بالاتر، بیش از ۲۰ روش مختلف برای دسترسی به حافظه در نظر گرفته شده است که اکثر آنها مشابه یکدیگر هستند و می‌توان آنها را در ۵ گروه طبقه‌بندی کرد. در زیر برخی از آنها را مورد بررسی قرار خواهیم داد.

جانشینی ساده

مهمترین آدرس‌دهی محسوب می‌شود که کاربرد آن نیز از سایر مدها بیشتر است. به عنوان مثال دستورالعمل Mov eax, ds: [8088h]، ثبات EAX را با مقدار موجود در آدرس 8088h از حافظه

داده برنامه مقدار دهی می‌کند و یا دستورالعمل Mov ds : [1234h], dl مقدار موجود در ثبات dl را در آدرس 1234h از حافظه داده قرار می‌دهد.

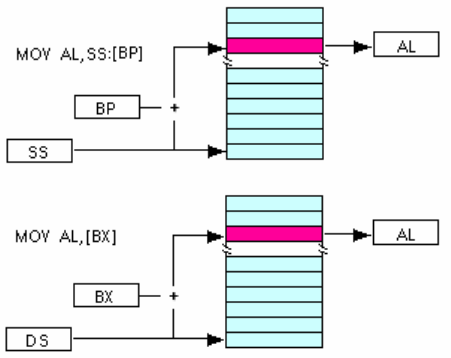


شکل (۷-۷)

آدرس‌دهی غیر مستقیم با استفاده از ثبات‌ها

با استفاده از این مد می‌توانید به آدرس مشخصی از حافظه به‌صورت غیرمستقیم و از طریق ثبات‌ها دسترسی پیدا کنید. به عنوان مثال نگاهی به دستورالعمل‌های ساده زیر بیاندازید.

```
Mov eax , [ebx]
Mov al , [bx]
Mov  al , ss : [bp]
```



شکل (۷-۸)

در صورت لزوم برای دریافت اطلاعات بیشتر راجع به دستورالعمل‌های استاندارد پردازنده‌های 80x86 می‌توانید به ضمیمه آخر کتاب مراجعه کنید.

شناسایی ساختارهای کلیدی مورد استفاده در زبان‌های سطح بالا

شناسایی عملگرهای ریاضی

شناسایی عملگر جمع

در اکثر موارد عملگر + به دستور ماشین ADD ترجمه می‌شود. از دستور ADD برای اعداد طبیعی و از fADD برای پردازش اعداد ممیز شناور استفاده می‌شود. کامپایلرهایی که بهینه‌سازی کد انجام می‌دهند دستور INC XXX را جایگزین ADD XXX , 1 می‌کنند. همچنین ممکن است عبارت c = a +b + Const را به دستور ماشین LEA C , [a + b + Const] ترجمه کنند. این روش باعث می‌شود که چندین متغیر در یک حرکت جمع شوند. همچنین این امکان را می‌دهد که حاصل جمع از طریق یکی از ثبات‌های همه منظور بدست آید. هدف اصلی دستورالعمل LEA بدست آوردن offset مؤثر می‌باشد ولی در بعضی موارد در کامپایلرهایی که بهینه‌سازی کد را انجام می‌دهند کار دستور ADD را نیز انجام می‌دهد.

به مثال زیر توجه کنید:

مثال: شناسایی عملگر" +"

```
main()
{
       int a, b, c;
       c = a + b;
       printf("%x\n", c);
       c=c+1;
       printf("%x\n", c);
}
```

کد disassemble شده عملگر" +" در ویژوال C++

```
main          proc near               ; CODE XREF: start+AF↓p

var_c         = dword ptr - 0Ch
var_b         = dword ptr -8
var_a         = dword ptr -4

       push   ebp
       mov    ebp, esp
```

قاب پشته باز می‌شود.

```
sub     esp, 0Ch
```

حافظه به متغیرهای محلی اختصاص داده می‌شود.

```
mov     eax, [ebp+var_a]
```

مقدار متغیر Var_a در EAX بارگذاری می‌شود.

```
add     eax, [ebp+var_b]
```

مقدار متغیر Var_a با EAX جمع می‌شود و حاصل در ثبات EAX قرار داده می‌شود.

```
mov     [ebp+var_c], eax
```

حاصل جمع Var_a و Var_b در Var_c کپی می‌شود.

```
mov     ecx, [ebp+var_c]
push    ecx
push    offset asc_406030 ; "%x\n"
call    _printf
add     esp, 8
```

Printf("%\n", Var_C)

```
mov     edx, [ebp+var_c]
```

مقدار متغیر Var_cدر EDX بارگذاری می‌شود.

```
add     edx, 1
```

مقدار 0x1 با EDX جمع شده و حاصل در EDX قرار داده می‌شود.

```
mov     [ebp+var_c], edx
```

مقدار Var_c به روزرسانی می‌شود Var_c = Var_c + 1

```
mov     eax, [ebp+var_c]
push    eax
push    offset asc_406034 ; "%x\n"
call    _printf
add     esp, 8
```

Printf("%x\n", Var_C)

```
mov     esp, ebp
pop     ebp
```

قاب پشته بسته می‌شود.

```
        retn
main            endp
```

شناسایی عملگر تفریق

عملگر تفریق به‌طورکلی به دستور SUB برای اعداد طبیعی و fSUBX برای اعداد ممیز شناور ترجمه می‌شود. کامپایلرهایی که بهینه‌سازی کد انجام می‌دهند معمولاً دستور SUB XXX ,1 را با دستور DEC XXX جایگزین می‌کنند. برخی اوقات نیز ممکن است به جای SUB A , Const از ADD a , -Const استفاده کنند چون ADD سریعتر از Sub می‌باشد.

مثال: شناسایی عملگر تفریق "–"

```
main()
{
        int a, b, c;

        c = a - b;
        printf("%x\n", c);

        c = c - 10;
        printf("%x\n", c);
}
```

کد disassemble شده با عملگر تفریق

```
main            proc near               ; CODE XREF: start+AF↓p

var_c           = dword ptr -0Ch
var_b           = dword ptr -8
var_a           = dword ptr -4

        push    ebp
        mov     ebp, esp
```

قاب پشته باز می‌شود.

```
        sub     esp, 0Ch
```

حافظه به متغیرهای محلی اختصاص داده می‌شود.

```
        mov     eax, [ebp+var_a]
```

مقدار متغیر Var_a در EAX بارگذاری می‌شود.

```
sub     eax, [ebp+var_b]
```

مقدار متغیر Var_b از Var_a کم می‌شود و حاصل در EAX قرار داده می‌شود.

```
mov     [ebp+var_c], eax
```

تفاوت Var_b و Var_a در Var_c قرار داده می‌شود.

```
mov     ecx, [ebp+var_c]
push    ecx
push    offset asc_406030 ; "%x\n"
call    _printf
add     esp, 8
```

```
Printf ("%x\n", Var_C)
```

```
mov     edx, [ebp+var_c]
```

مقدار متغیر Var_c در EDX بارگذاری می‌شود.

```
sub     edx, 0Ah
```

مقدار 0xA از Var_c کم می‌شود و حاصل در EDX قرار داده می‌شود.

```
mov     [ebp+var_c], edx
```

مقدار متغیر Var_c بروز رسانی می‌شود.

```
mov     eax, [ebp+var_c]
push    eax
push    offset asc_406034 ; "%x\n"
call    _printf
add     esp, 8
```

```
Printf ("%x\n", Var_C)
```

```
mov     esp, ebp
pop     ebp
```

قاب پشته بسته می‌شود.

```
        retn
main            endp
```

شناسایی عملگر تقسیم

عملگر"/" (تقسیم) معمولاً به دستور ماشین DIV برای اعداد بدون علامت و یا IDIV برای اعداد علامت‌دار یا fDIVX برای اعداد ممیز شناور ترجمه می‌شود. اگر تقسیم‌کننده از توان‌های ۲ باشد دستور DIV با دستور SHR a,N جایگزین می‌شود چون شیفت به راست بسیار سریع‌تر از تقسیم می‌باشد. در این دستور a تقسیم شونده و N شمارهٔ توان ۲ است

روش سریع تقسیم اعداد علامت‌دار کمی پیچیده است. شیفت حسابی به راست برای این کار کافی نیست. چون باعث می‌شود که با ارزش‌ترین بیت که بیت علامت عدد است تغییر کند. به‌طورکلی شیفت راست حسابی برای تقسیم اعداد علامت‌دار باعث گرد شدن حاصل می‌شود. برای این که این گرد کردن رخ ندهد باید عدد 2^N-1 قبل از تقسیم، به تقسیم شونده اضافه شود. این کار باعث می‌شود که به تمام بیت‌های شیفت داده شده یک واحد اضافه شود و رقم نقلی با، با ارزش‌ترین بیت جمع شود. متذکر می‌شویم که به‌طورکلی عملیات تقسیم سرعت کمتری از ضرب دارد اجرای دستور Div بیشتر از ٤٠ سیکل طول می‌کشد اما ضرب حدود ٤ سیکل زمان نیاز دارد. به این دلیل است که کامپایلرهای پیشرفته عملیات تقسیم را با ضرب جایگزین می‌کنند. برای این کار فرمول‌های زیادی وجود دارد که معروف‌ترین آن این است $a / b = 2^N/b \times a/2^N$. در اینجا N تعداد بیت‌های اعداد است. با این وجود تفاوت بین تقسیم و ضرب بسیار ناچیز می‌باشد و همین مسئله شناسایی آن دو از هم را مشکل ساخته است.

به مثال زیر توجه کنید:

شناسایی عملگر تقسیم "/" :

```
main()
{
       int a;
       printf("%x %x\n", a/32, a/10);
}
```

کد disassemble شده با عملگر تقسیم (/) در ویژوال C++.

```
main           proc near               ; CODE XREF: start+AF↓p

var_a          = dword ptr -4

       push    ebp
       mov     ebp, esp
```

قاب پشته باز می‌شود

```
push    ecx
```

حافظه به متغیرهای محلی اختصاص داده می‌شود

```
mov     eax, [ebp+var_a]
```

مقدار متغیر Var_a در EAX کپی می‌شود

```
cdq
```

مقدار EAX به اندازه ۲ DWORD (EDX:EAX) گسترده می‌شود.

```
mov     ecx, 0Ah
```

مقدار 0xA در ECX قرار داده می‌شود

```
idiv    ecx
```

مقدار EDX:EAX بر 0xA تقسیم شده و خارج قسمت در EAX قرار داده می‌شود. EAX = Var_a / 0xA

```
push    eax
```

حاصل محاسبات به تابع Printf فرستاده می‌شود

```
mov     eax, [ebp+var_a]
```

مقدار متغیر Var_a در EAX بارگذاری می‌شود

```
cdq
```

مقدار EAX به اندازه ۲ DWORD (EDX:EAX)گسترده می‌شود.

```
and     edx, 1Fh
```

۵ بیت کم ارزش EDX انتخاب می‌شود

```
add     eax, edx
```

علامت عدد برای از بین بردن گرد شدن، به عدد طبیعی کوچکتر اضافه می‌شود

```
        sar     eax, 5
```

شیفت حسابی ۵ بیت معادل تقسیم عدد به 2^5 (۳۲) می‌باشد. یعنی ٤ دستور بالا نشان‌دهندهٔ عملیات EAX = Var_a / 32 می‌باشد.

```
        push    eax
        push    offset aXX          ; "%x %x\n"
        call    _printf
        add     esp, 0Ch
```

```
Printf("%x %x", Var_a/5xA, Var_a/32)
```

```
        mov     esp, ebp
      pop     ebp
```

قاب پشته بسته می‌شود.

```
        retn
main            endp
```

شناسایی عملگر ضرب

عملگر ضرب معمولاً توسط کامپایلر به دستور ماشین MUL برای اعداد بدون علامت و IMUL برای ضرب اعداد با علامت و FMULX برای ضرب اعداد با ممیز شناور ترجمه می‌شود. اگر ضرب‌کننده توان ۲ باشد دستورات MUL یا IMUL معمولاً با دستور SHL جایگزین می‌شوند. این دستور، اعداد را شیفت به چپ می‌دهد یا دستور LEA که محتوای ثبات‌ها را ضرب در ۲ ، ٤ ، ۸ می‌کند. این دو دستور بیان شده برای اجرا به دو سیکل زمانی نیاز دارند. اما خود دستور MUL بین ۲ تا ۹ سیکل زمانی نیاز دارد. این مسئله بسته به نوع پردازنده می‌باشد. با اضافه کردن خود مقدار به حاصل، توسط دستور LEA می‌توان ضرب در ۳ و ۵ و ۹ را نیز ایجاد کرد. البته از دستور LEA برای مقاصد دیگری استفاده می‌شود و در ضرب کاربرد چندانی ندارد.

به مثال زیر توجه کنید:

مثال: شناسایی عملگر ضرب "*"

```
main ()
{
        int a;
        printf("%x %x %x\n", a*16, a*4+5, a*13);
}
```

کد disassemble شده با عملگر * ضریب توسط ویژوال C++

```
main            proc near               ; CODE XREF: start+AF↓p

var_a           = dword ptr -4

        push    ebp
        mov     ebp, esp
```

قاب پشته باز می‌شود.

```
        push    ecx
```

حافظه به متغیر محلی Var_a اختصاص داده می‌شود.

```
        mov     eax, [ebp+var_a]
```

مقدار متغیر Var_a در EAX بارگذاری می‌شود.

```
        imul    eax, 0Dh
```

این دستور متغیر Var_a را در 0xD ضرب کرده و حاصل را در EAX قرار می‌دهد.

```
        push    eax
```

حاصل 0xD * Var_a به تابع Printf فرستاده می‌شود.

```
        mov     ecx, [ebp+var_a]
```

مقدار Var_a در ECX بارگذاری می‌شود.

```
        lea     edx, ds:5[ecx*4]
```

مقدار ECX در ٤ ضرب می‌شود. سپس مقدار ٥ به آن اضافه شده و حاصل در EDX قرار داده می‌شود. این کار در یک سیکل زمانی انجام می‌شود.

```
        push    edx
```

مقدار عبارت 5+4 * Var_a به تابع Printf فرستاده می‌شود.

```
        mov     eax, [ebp+var_a]
```

مقدار متغیر Var_a در EAX بارگذاری می‌شود.

```
        shl     eax, 4
```

مقدار متغیر Var_a در ۱۶ ضرب می‌شود.

```
        push    eax
```

حاصل عبارت Var_a * 16 به تابع Printf فرستاده می‌شود.

```
        push    offset aXXX             ; "%x %x %x\n"
        call    _printf
        add     esp, 10h
```

```
Printf("%x %x %x", Var_a * 16, Var_a * 4+5, Var_a *0xD)
```

```
        mov     esp, ebp
        pop     ebp
```

قاب پشته بسته می‌شود.

```
        retn
main            endp
```

عملگرهای ++ و - -

زبان‌های C و C++ از دو عملگر ++ و - - پشتیبانی می‌کنند. این عملگرها به همان صورت اصلی a = a + b یا a = a - b ترجمه می‌شوند و شناسایی آنها به‌همان روش قبل است. درحالت کلی دستور a ×= b در زمان کامپایل به a=a×b ترجمه می‌شود. عملگرهای خاص ++ و - - نیز به‌صورت a=a+1 و a=a-1 تبدیل می‌شوند که بسیار ساده می‌باشند.

شناسایی رشته‌ها

در نگاه اول به نظر می‌رسد شناسایی رشته‌ها مشکلات جدیدی را ایجاد کند. رشته‌ها به سادگی با یک نگاه ساده به کدهای خام برنامه شناخته شده و کشف می‌شوند اما پیچیدگی‌های خاصی وجود دارد که به بررسی آنها می‌پردازیم. اولین کار جستجوی سریع رشته‌ها در برنامه می‌باشد. الگوریتم‌های مختلفی برای شناسایی رشته‌ها وجود دارد. ساده‌ترین آنها که زیاد قابل اعتماد هم نمی‌باشد براساس نظر زیر است.

رشته‌ها از تعداد محدودی کاراکتر تشکیل شده‌اند.

معمولاً کاراکترها ترکیبی از اعداد، حروف و علامات کنترلی هستند. یک رشته حداقل از چند کاراکتر ایجاد شده است. اگر فرض کنیم که حداقل طول یک رشته N باشد کافی است که بتوانیم ترتیبی از N کاراکتر معتبر را پیدا کنیم. اگر N کوچک باشد مانند ۲ و ۳ تعداد زیادی اشتباه خواهیم داشت اما اگر N بزرگ باشد مانند ۷ و ۸ تعداد پاسخ‌های اشتباه تقریباً به صفر می‌رسد. در تعداد کم عباراتی مثل ok و yes و no قابل شناسایی نیستند. به‌طورکل رشته می‌تواند از هر چیزی که در جدول اسکی وجود دارد تشکیل شده باشد. وقتی رشته‌ای در برنامه وجود دارد ارجاعاتی نیز باید به آن وجود داشته باشد. ممکن است با جستجو در میان مقادیر بلافصل برای اشاره‌گر به رشته شناخته شده، بتوانیم رشته را شناسایی کنیم و میان رشته و ترتیب اتفاقی از بایت‌ها تمایز قائل شویم.

کار به این سادگی نیست به مثال زیر توجه کنید:

یک رشته در یک برنامه:

```
BEGIN
WriteLn ('Hello, Sailor!');
END
```

برنامه با استفاده از یک کامپایلر پاسکال کامپایل شده است. پس از کامپایل، وارد بخش داده شده و اطلاعات زیر را مشاهده می‌کنیم.

محتوای بخش داده مثال کامپایل شده

```
.data:00404040 unk_404040   db 0Eh ;
.data:00404041              db 48h ; H
.data:00404042              db 65h ; e
.data:00404043              db 6Ch ; l
.data:00404044              db 6Ch ; l
.data:00404045              db 6Fh ; o
.data:00404046              db 2Ch ; ,
```

```
.data:00404047                db 20h ;
.data:00404048                db 53h ; s
.data:00404049                db 61h ; a
.data:0040404A                db 69h ; i
.data:0040404B                db 6Ch ; 1
.data:0040404C                db 6Fh ; o
.data:0040404D                db 72h ; r
.data:0040404E                db 21h ; !
.data:0040404F                db 0 ;
.data:00404050 word_404050    dw 1332h
```

می‌دانیم که این یک رشته است و هیچ شکی در آن نیست حال می‌خواهیم ببینیم که چگونه به آن ارجاع شده است. در IDA Pro این کار با ترکیب دو کلید <Alt>+<I> و وارد کردن offset شروع رشته انجام می‌شود یعنی مقدار 0x404041 . پاسخ منفی است؟ پس چه چیز به تابع writeln فرستاده شده است؟ علت شکست این است که در پاسکال یک بایت در ابتدای رشته وجود دارد که حاوی طول رشته می‌باشد. مقدار 0xE در offset ، 0x404040 قرار دارد یعنی تعداد کاراکترهای رشته "Hello, Salilor!". فشار دادن دوباره کلید <Alt> + <I> و جستجوی عملوند بلافصل 0x404040 پاسخ زیر را باز می‌گرداند.

نتیجهٔ جستجوی عملوند بلافصل:

```
.text:00401033       push  404040h
.text:00401038       push  [ebp+var_4]
.text:0040103B       push  0
.text:0040103D       call  FPC_WRITE_TEXT_SHORTSTR
.text:00401042       push  [ebp+var_4]
.text:00401045       call  FPC_WRITELN_END
.text:0040104A       push  offset loc_40102A
.text:0040104F       call  FPC_IOCHECK
.text:00401054       call  FPC_DO_EXIT
.text:00401059       leave
.text:0040105A       retn
```

شناسایی یک رشته کافی نمی‌باشد بلکه باید محدوده آن هم شناسایی شود. انواع زیر معروف‌ترین نوع رشته‌ها هستند.

رشته‌های C

این رشته‌ها ASCIIZ نیز نامیده می‌شوند که Z به معنی Zero در انتها می‌باشد. رشته‌های C به‌طور بسیار وسیعی در سیستم عامل‌های ویندوز و یونیکس مورد استفاده قرار گرفته‌اند. کاراکتر "\0" وظیفه خاصی انجام می‌دهد و به عنوان قطع‌کننده و یا ختم‌کننده رشته نامیده می‌شود. اندازه

رشته‌های ASCIIZ به اندازه فضایی که به فرآیند اختصاص داده شده یا به طول Segment، محدود می‌باشد. اندازه رشته در سیستم عامل‌های NT/9X کمتر از ۲ گیگابایت و در Dos و ویندوز 3.1 کمتر از 64k بایت می‌باشد. رشته‌های ASCIIZ یک بایت بزرگتر از رشته‌های ASCII می‌باشند. این نوع از رشته‌ها معایبی نیز دارند که به بیان آنها می‌پردازیم. این رشته‌ها نمی‌توانند شامل صفر بایت باشند پس برای پردازش اطلاعات باینری مناسب نمی‌باشند. اجرای عملیات کپی، مقایسه، قطع و اتصال با رشته‌های C سربار زیادی را به سیستم تحمیل می‌کند. برای پردازشگرهای امروزی مناسب‌تر است که با double word کار کنند تا به با یک بایت. اما متأسفانه طول رشته‌های C از قبل قابل پیش‌بینی نبوده و باید تک تک بایت‌ها به‌منظور شناسایی کاراکتر خاتمه مورد بررسی قرار گیرند. کامپایلرهای جدید از یک حیله استفاده می‌کنند، آنها رشته را با هفت صفر ختم می‌کنند. با این کار می‌توانند به سادگی مقایسه‌ها را به double word تبدیل کنند.

رشته‌های پاسکال

رشته‌های پاسکال کاراکتر مختوم‌کننده ندارند. آنها دارای فیلدی می‌باشند که حاوی طول رشته است. با استفاده از این روش می‌توان رشته‌هایی به طول صفر داشت. دیگر اینکه می‌توان پردازش سریع‌تری برروی متغیرهای رشته‌ای انجام داده و دیگر نیازی به بررسی تک تک بایت‌های حافظه برای پیدا کردن کاراکتر خاتمه، نیست. وقتی طول رشته شناخته شده باشد به سادگی می‌توان با نوع double word که نوع طبیعی پردازشگرهای ۳۲ بیتی می‌باشد کار کرد. سئوالی که مطرح می‌شود این است که این فیلد چند بایت است. در پاسکال این فیلد یک بایت است و طول رشته‌ها حداکثر ۲۵۵ می‌باشد. این نوع رشته‌ها به رشته‌های پاسکال معروف می‌باشند که به‌طور دقیق‌تر به آنها رشته‌های کوتاه پاسکال می‌گویند.

رشته‌های دلفی

در دلفی طول فیلد اندازه رشته به دو بایت افزایش یافته است. یعنی حداکثر طول رشته‌های دلفی ٦٥٥٣٥ بایت می‌باشد. رشته‌هایی با این مشخصه را رشته‌های دو بایتی پاسکال نیز می‌نامند. رشته‌های دلفی بهترین خصوصیات رشته‌های C و پاسکال را ترکیب کرده است (تقریباً طول نامحدود و سرعت پردازش بالا). این رشته‌ها بهترین نوع برای استفاده می‌باشند.

رشته‌های گستردهٔ پاسکال

در این نوع رشته‌ها حجم فیلد نگه‌داری طول رشته به ٤ بایت افزایش یافته است. یعنی طول این رشته‌ها تقریباً می‌تواند تا ٤ گیگا بایت باشد. این اندازه تقریباً بیشتر از آن چیزی است که ویندوز NT و 9X از حافظه به برنامه‌ها و فرآیندها اختصاص می‌دهند. البته دراکثر موارد سه بایت از این فیلد خالی می‌ماند. این نوع رشته‌ها باعث افزایش سربار سیستم می‌شوند و از آنها به ندرت استفاده می‌شود.

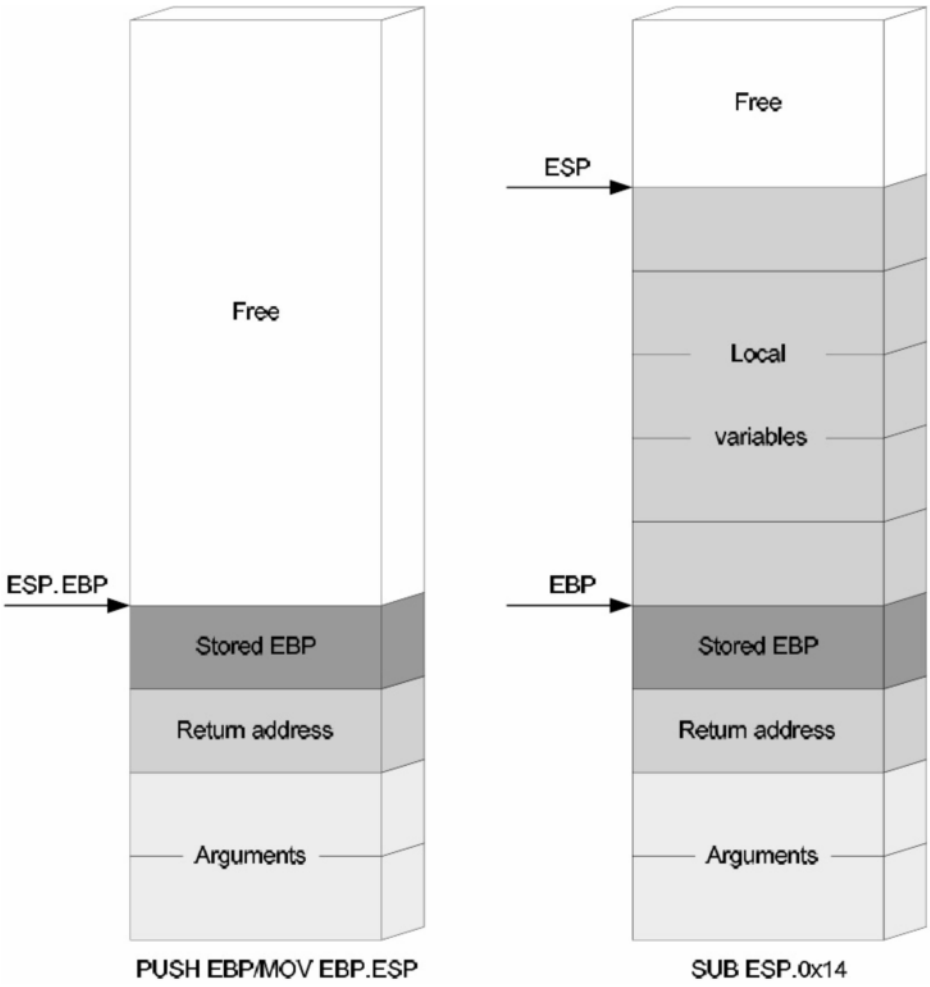
انواع ترکیبی

در برخی از کامپایلرها از ترکیبی از رشته‌های C و پاسکال استفاده می‌شود. رشته‌های ترکیبی C و پاسکال به شما اجازه پردازش سریع و نگه‌داری طول دلخواه را می‌دهند. این رشته‌ها می‌توانند به سادگی توسط توابع کتابخانه‌ای C که با رشته‌های ASCIIZ کار می‌کنند مورد استفاده قرار گیرند. تمام رشته‌های ترکیبی حتماً با صفر خاتمه می‌یابند. اما این صفر در رشته ظاهر نمی‌شود. کتابخانه‌های معمولی با آنها به عنوان رشته‌های پاسکال برخورد می‌کنند. اما هنگام استفاده از توابع کتابخانه‌ای C، کامپایلر به جای فرستادن اشاره‌گر به شروع واقعی رشته، اشاره‌گر به اولین کاراکتر رشته را به تابع می‌فرستد.

شناسایی متغیرهای محلی

متغیرهای محلی در پشته قرار داده می‌شوند و پس از پایان تابع از بین می‌روند. ابتدا هر آرگومانی که به تابع فرستاده می‌شود در پشته قرار داده می‌شود. دستور Call که تابع را فراخوانی می‌کند، آدرس بازگشت را در بالای آرگومان‌ها قرار می‌دهد. پس از گرفتن کنترل، تابع قاب پشته را باز می‌کند مقدار قبلی EBP را ذخیره کرده و آن را برابر ESP قرار می‌دهد (ESP به بالای پشته اشاره می‌کند). فضای آزاد پشته، بالای EBP قرار دارد (در آدرس‌های پایین‌تر). داده‌های خدماتی مثل مقدار ذخیره شده EBP و آدرس بازگشت مانند دیگر آرگومان‌ها در زیر این فضای آزاد قرار دارند. فضای بالای اشاره‌گر ESP می‌تواند پاک شود. برای مثال می‌تواند توسط کنترل‌کننده وقفه‌های سخت‌افزاری در هر زمانی مورد استفاده قرار گیرد. اگر در چنین شرایطی تابعی از پشته استفاده کند نتیجهٔ آن خرابی پشته خواهد بود. تنها راه برای جلوگیری از این اتفاق این است که اشاره‌گر پشته را تا جایی که نیاز داریم بالا بریم تا فضای مورد نیاز را اشغال کند. بعد از اتمام مراحل اجرا، تابع باید مقدار ESP را به مقدار قبلیش باز گرداند در غیر این صورت دستور RET نمی‌تواند آدرس بازگشت را از پشته بردارد و به جای آن آخرین متغیر محلی را برداشته و کنترل را به جایی ناشناخته می‌فرستد.

تذکر: *بخش چپ شکل (۷-۹) پشته را در زمان فراخوانی تابع نمایش می‌دهد. تابع، قاب پشته را باز کرده، مقدار قبلی EBP را ذخیره می‌کند و آن را برابر ESP قرار می‌دهد. بخش راست شکل (۷-۹) اختصاص 0x14 بایت از حافظه را به متغیرهای محلی نشان می‌دهد. این عمل با بالا بردن ثبات ESP صورت می‌گیرد. یعنی حرکت به مکان‌هایی باآدرس پایین‌تر. حافظهٔ موردنیاز برای متغیرهای محلی توسط دستور pushاز فضای پشته اختصاص داده می‌شود. بعد از اجرا، تابع مقدار ثبات ESP را افزایش داده و آن را به مقدار قبلیش باز می‌گرداند و با این کار حافظه اشغال شده توسط متغیرهای محلی را آزاد می‌کند سپس مقدار EBP را بازیابی کرده و قاب پشته را می‌بندد.*

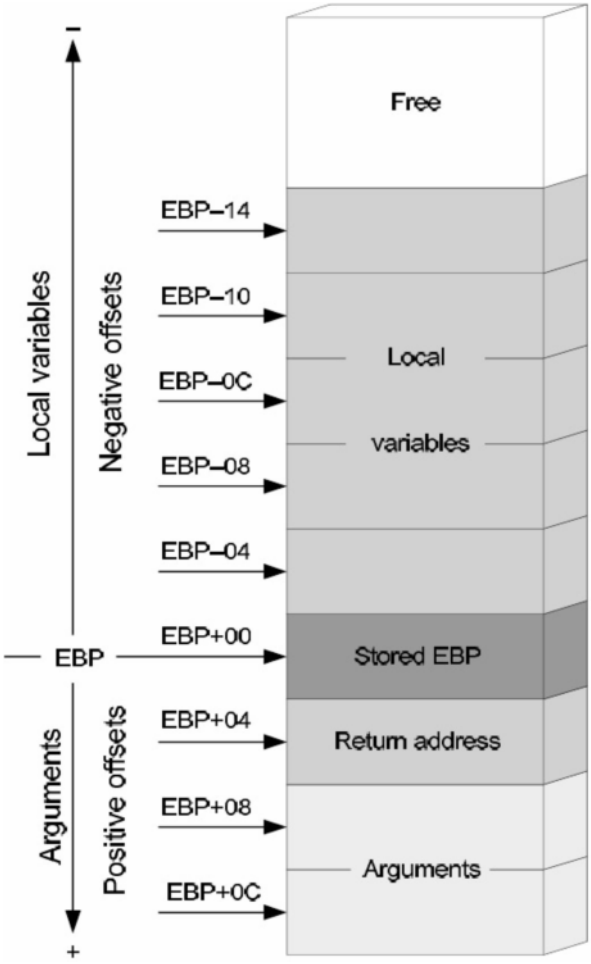


شکل(۷-۹) مکانیزم اختصاص فضا در پشته به متغیرهای محلی

آدرس‌دهی متغیرهای محلی

متغیرهای محلی و آرگومان‌های پشته به‌صورت مشابه آدرس‌دهی می‌شوند. تنها فرق بین این دو این است که آرگومان‌ها پایین EBP هستند و متغیرهای محلی بالای EBP می‌باشند. به بیان دیگر آرگومانها offset مثبت نسبت به EBP داشته و متغیرهای محلی offset منفی دارند. با این وجود، این دو به سادگی قابل تمایز هستند. برای مثال [EBP+XXX] یک آرگومان است و [EBP-XXX] یک متغیر محلی می‌باشد.

در حقیقت ثباتی که به قاب پشته اشاره می‌کند به عنوان حصاری به خدمت گرفته می‌شود و آرگومان‌های تابع یک طرف و متغیرهای محلی در طرف دیگر آن هستند. هنگامی‌که قاب پشته باز می‌شود مقدار ESP در EBP کپی می‌شود. اگر این کپی انجام نشود آدرس‌دهی متغیرهای محلی و آرگومان‌ها پیچیده و مشکل می‌شود. طراحان و توسعه‌دهندگان کامپایلرها نیز مانند سایر انسان‌ها هیچ علاقه‌ای به پیچیده کردن کارها ندارند. به هر حال کامپایلرهایی که بهینه‌سازی کد را انجام می‌دهند توانایی این را دارند که آدرس‌دهی متغیرهای محلی و آرگومان‌ها را از طریق ESP انجام داده و ثبات EBP را برای کارهای مفیدتر آزاد بگذارند.



شکل(۷-۱۰) آدرس‌دهی متغیرهای محلی

جزئیات پیاده‌سازی

روش‌های بسیار زیادی به‌منظور اختصاص و آزادسازی فضا، برای متغیرهای محلی وجود دارد. برای مثال SUB ESP ,XXX می‌تواند در ابتدا و ADD ESP, XXX در انتها قرار گیرد. الگوریتم‌های آزادسازی حافظه نیز تقریباً مبهم هستند. بعداز مواجهه شدن با افزایش ثبات اشاره‌گر پشته (ESP) توسط دستور ADD ESP, XXX، ممکن است ساختار MOV ESP, EBP را ببینید. زیرا وقتی که قاب پشته باز می‌شود مقدار ESP در EBP کپی می‌شود و در طول اجرای تابع مقدار EBP تغییری نمی‌کند. در آخر حافظه با دستور POP آزاد می‌شود و به این صورت متغیرهای محلی را تک تک برداشته و در یک ثبات بلا استفاده می‌ریزیم. این روش زمانی مورد استفاده قرار می‌گیرد که تعداد متغیرهای محلی کم باشد.

Table 8.14: Allocating and Clearing Memory for Local Variables

Action	Implementation variants		
Allocating memory	`SUB ESP, xxx`	`ADD ESP, - xxx`	`PUSH reg`
Releasing memory	`ADD ESP, xxx`	`SUB ESP, - xxx`	`POP reg`
	`MOV ESP, EBP`		

جدول(۷-۱) اختصاص و آزادسازی حافظه برای متغیرهای محلی

شناسایی مکانیزم به کاربرده شده برای اختصاص حافظه با استفاده از دستورات SUB و ADD نشان می‌دهد که اختصاص حافظه بدون ابهام ، قابل توضیح و تفسیر است. اگر اختصاص حافظه بااستفاده از دستور Push و آزادسازی با استفاده از دستور POP باشد این ساختار از یک اختصاص حافظه ساده یا ذخیره سازی ثبات‌ها در پشته غیرقابل تمایز است. یک تابع ممکن است شامل دستوراتی برای اختصاص ثبات‌ها و همچنین شامل دستوراتی برای اختصاص حافظه باشد. برای یافتن تعداد بایت‌های اختصاص داده شده به متغیرهای محلی جستجوی ارجاعات به فضای بالای موقعیت ثبات EBP مفید می‌باشد.

شناسایی مکانیزم اختصاص حافظه

```
push ebp          push ebp
push ecx          push ecx
xxx           xxx
mov [ebp-4], 0x666          xxx
xxx           xxx
pop ecx           pop ecx
pop ebp           pop ebp
ret                     ret
```

در مثال سمت چپ هیچ ارجاعی به متغیرهای محلی وجود ندارد اما در مثال راست ساختار MOV [EBP - 4], OX666 مقدار 0X666 را در متغیر محلی Var_4 کپی می‌کند. اگر متغیر محلی وجود داشته باشد باید فضایی به آن اختصاص یابد. از آنجایی که در بدنه تابع دستوراتی مانند SUB ESP , XXX یا ADD ESP , XXX وجود ندارد اختصاص فضا توسط دستور Push ECX صورت گرفته است و محتوای ثبات ECX در ٤ بایت بالای EBP ذخیره می‌شود.

مقدار حافظه اختصاص یافته، توسط بالاترین offset متغیرهای محلی در بدنه تابع مشخص می‌گردد. به بیان دیگر درمیان کلیه عبارت [EBP-XXX] بزرگترین offset برابر با تعداد بایت‌های اختصاص داده شده به متغیرهای محلی می‌باشد. البته متغیرهای محلی ممکن است معرفی شوند اما استفاده نگردند.

مقداردهی اولیه متغیرهای محلی

دو راه برای مقداردهی اولیه متغیرهای محلی وجود دارد اختصاص مقدار با استفاده از دستور Mov مانند Mov [EBP_04], OX666 یا وارد کردن مستقیم مقدار در پشته با استفاده از دستور Push در اکثر موارد کامپایلرها مقداردهی را با استفاده از دستور Mov انجام می‌دهند اما اسمبلرها معمولاً علاقمند به استفاده از دستور Push هستند.

اختصاص حافظه به رکوردها و آرایه‌ها

رکوردها و آرایه‌ها در حافظه به ترتیب و پشت سرهم قرار می‌گیرند. اندیس‌های کوچکتر یک آرایه در آدرس کوچکتر قرار داشته ولی در offset دورتری نسبت به اشاره‌گر ESP قرار دارند. زیرا همان‌طور که ذکر شد متغیرهای محلی توسط offset منفی آدرس‌دهی می‌شود [EBP - 0x4] < [EBP - 0x10].

ایجاد متغیرهای موقت برای ذخیره مقدار بازگشتی یک تابع و نتیجه عبارات محاسباتی

اکثر زبان‌های سطح بالا اجازه استفاده از توابع و عبارات را به عنوان آرگومان می‌دهند. برای مثال myfunc(a+b, myfunc_2(c)] قبل از فراخوانی تابع myfunc کامپایلر باید مقدار عبارت a+b را محاسبه کند. مشکل اینجاست که حاصل این جمع در کجا باید ذخیره شود؟

کد disassemble شده نشان می‌دهد که کامپایلرها چگونه مقدار حاصل از محاسبهٔ عبارات و مقادیر بازگشتی توابع را ذخیره می‌کنند.

```
mov    eax, [ebp+var_C]
```

متغیر موقتی tEAX ایجاد شده و مقدار متغیر محلی Var_c در آن کپی می‌شود.

```
push   eax
```

متغیر موقتی tEAX در پشته ذخیره شده و مقدار متغیر محلی Var_c به عنوان آرگومان به تابع myfunc فرستاده می‌شود.

```
call   myfunc
add    esp, 4
```

مقدار تابع myfunc به داخل ثبات EAX باز می‌گردد. EAX را می‌توان نوعی متغیر موقتی در نظر گرفت.

```
push   eax
```

مقدار بازگشتی توسط تابع myfunc به عنوان آرگومان به تابع myfunc_2 فرستاده می‌شود.

```
mov    ecx, [ebp+var_4]
```

مقدار متغیر محلی Var_4 در ECX کپی می‌شود. ECX همچنان به عنوان یک متغیر موقتی در نظر گرفته می‌شود. هنوز روشن نیست که چرا کامپایلر از ثبات EAX استفاده نکرده است. زیرا متغیر موقتی قبلی دیگر مورد نیاز نیست.

```
add    ecx, [ebp+var_8]
```

```
ECX:=Var_4+Var_8
push   ecx
```

حاصل جمع دو متغیر محلی به تابع myfunc_2 فرستاده می‌شود.

```
call   _myfunc_2
```

حوزهٔ متغیرهای موقتی

متغیرهای موقتی به اندازه متغیرهای محلی هستند و در اکثر موارد حوزهٔ آنها محدود به چند خط از کد برنامه بوده و در خارج آن بی‌معنی هستند. متغیرهای محلی معمولاً باعث نامفهوم شدن کد می‌شوند برای مثال myfunc(a+b) قابل فهم تر از tmp=a+b ; myfunc(tmp) است.

یک مثال از توضیح مناسب برنامه:

```
mov eax, [ebp+var_4]     ; var_8 := var_4
                         ; ^ tEAX := var_4
add eax, [ebp+var_8],    ; ^ tEAX += var_8

push eax                 ; MyFunc (var_4 + var_8)
call MyFunc
```

شناسایی متغیرهای سراسری

شناسایی متغیرهای سراسری در زبان‌های سطح بالا راحت‌تر از شناخت هر چیز دیگری می‌باشد. متغیرهای سراسری بطور مستقیم آدرس‌دهی می‌شوند برای مثال Mov EAX , [401066] که 0x401066 آدرس متغیر سراسری می‌باشد. شناخت هدف و کاربرد یک متغیر سراسری و همچنین مقدار آن در یک لحظه با استفاده از بررسی‌های ایستا کار بسیار مشکلی است. متغیرهای محلی توسط تابع پدر خود مقداردهی اولیه شده مورد استفاده قرار می‌گیرند. اما متغیرهای سراسری را درهمه جا می‌توان استفاده کرده و مقدار آنها را تغییر داد.

آدرس‌دهی غیرمستقیم متغیرهای سراسری

اگر یک متغیر سراسری با ارجاع به یک تابع فرستاده شود به‌صورت غیرمستقیم از طریق یک اشاره‌گر آدرس‌دهی می‌شود. حال این سئوال مطرح می‌شود که چرا باید متغیرهای سراسری به تابع فرستاده شوند؟ چون توابع می‌توانند متغیرهای سراسری را بدون آنکه به آنها فرستاده شوند آدرس‌دهی کنند. این مطلب زمانی درست است که تابع قبلاً اطلاعاتی درباره آن داشته باشد. حال فرض کنید تابعی به نام xchg داریم که مقادیر آرگومان‌های خود را با هم جایگزین می‌کند و می‌خواهیم دو متغیر سراسری را با هم تعویض کنیم. تابع xchg به متغیرهای سراسری دسترسی دارد اما نمی‌داند که کدام یک از آنها را باید تغییر دهد. اینجا همان جایی است که باید متغیرهای سراسری را به‌صورت آشکار به عنوان آرگومان به توابع بفرستیم. مطالب گفته شده نشان می‌دهد که با یک جستجوی ساده نمی‌توان تمام ارجاعات به متغیرهای سراسری را به‌دست آورد.

فرستادن آشکار یک متغیر سراسری

```
#include <stdio.h>
int a; int b;
```

متغیرهای سراسر a و b

```
xchg (int *a, int *b)
```

تابعی که مقادیر آرگومان‌ها را جا به جا می‌کند.

```
{
        int c; c=*a; *b=*a; *b=c;
 }
```

آرگومان‌ها به‌صورت غیرمستقیم با استفاده از اشاره‌گر آدرس‌دهی می‌شوند. اگر آرگومان‌های تابع متغیرهای سراسری باشند به‌صورت غیرمستقیم آدرس‌دهی می‌شوند.

```
main ( )
{
       a=0x666; b=0x777;
       xchg (&a, &b);
}
```

کد disassemble شده فرستادن آشکار متغیرهای سراسری با استفاده از کامپایلر ویژوال C++.

```
main          proc near         ; CODE XREF: start+AF↓p
       push   ebp
       mov    ebp, esp
```

قاب پشته باز می‌شود.

```
       mov    dword_405428, 666h
```

متغیر سراسری dword_405428 مقداردهی اولیه می‌شود. آدرس‌دهی غیرمستقیم نشان می‌دهد که این یک متغیر سراسری است.

```
       mov    dword_40542C, 777h
```

متغیر سراسری dword_40542c مقداردهی اولیه می‌شود.

```
       push   offset dword_40542C
```

offset متغیر سراسری dword_40542c به عنوان آرگومان به تابع فرستاده می‌شود. این بدان معنی است که تابع متغیرهای سراسری را با استفاده از اشاره‌گر به‌صورت غیرمستقیم همانند متغیرهای محلی آدرس‌دهی خواهد کرد.

```
       push   offset dword_405428
```

Offset متغیر سراسری dword_405428 به تابع فرستاده می‌شود.

```
       call   xchg
       add    esp, 8

       pop    ebp
       retn
main          endp
xchg          proc near       ; CODE XREF: main+21↑p
```

```
var_4           = dword ptr -4
arg_0           = dword ptr 8
arg_4           = dword ptr 0Ch

        push    ebp
        mov     ebp, esp
```

قاب پشته باز می‌شود.

```
        push    ecx
```

حافظه به متغیر محلی Var_4 اختصاص می‌یابد.

```
        mov     eax, [ebp+arg_0]
```

محتوای آرگومان arg_0 در EAX بارگذاری می‌شود.

```
        mov     ecx, [eax]
```

متغیر سراسری به‌صورت غیرمستقیم آدرس‌دهی می‌شود. فقط تجزیه و تحلیل تابع فراخواننده روشن می‌کند که یک متغیر سراسری آدرس‌دهی شده است.

```
        mov     [ebp+var_4], ecx
```

مقدار arg_0* در متغیر محلی Var_4 کپی می‌شود.

```
        mov     edx, [ebp+arg_4]
```

محتوای آرگومان arg_4 در EDX بارگذاری می‌شود.

```
        mov     eax, [ebp+arg_0]
```

آرگومان arg_0* در EAX بارگذاری می‌شود.

```
        mov     ecx, [eax]
```

آرگومان arg_0* در ECX بارگذاری می‌شود.

```
        mov     [edx], ecx
```

مقدار arg_0[0] در [arg_4] کپی می‌شود.

```
        mov     edx, [ebp+arg_4]
```

مقدار arg_4 در EDX بارگذاری می‌شود.

```
        mov     eax, [ebp+var_4]
```

مقدار متغیر محلی Var_4 در EAX بارگذاری می‌شود.

```
        mov     [edx], eax
```

مقدار arg_0* در arg_4* بارگذاری می‌شود.

```
        mov     esp, ebp
        pop     ebp
        retn
xchg            endp

dword_405428  dd 0                    ; DATA XREF: main+3↑w main+1C↑o
dword_40542C  dd 0                    ; DATA XREF: main+D↑w main+17↑o
```

IDA تمام ارجاعات به دو متغیر سراسری را پیدا کرده است. اولین دو ارجاع ,Main+D↑W Main+3↑W به مقداردهی اولیه ارجاع می‌کنند (W به معنی "Wirte " است یعنی آدرس‌دهی برای نوشتن). دومین دو ارجاع، Main+1C↑O و Main+17↑O هستند (O به معنی offset است یعنی بدست آوردن offset متغیرهای سراسری). Offset به این معنی است که متغیرهای سراسری با ارجاع فرستاده شده‌اند و فرستادن با ارجاع هم به معنی آدرس‌دهی غیرمستقیم می‌باشد.

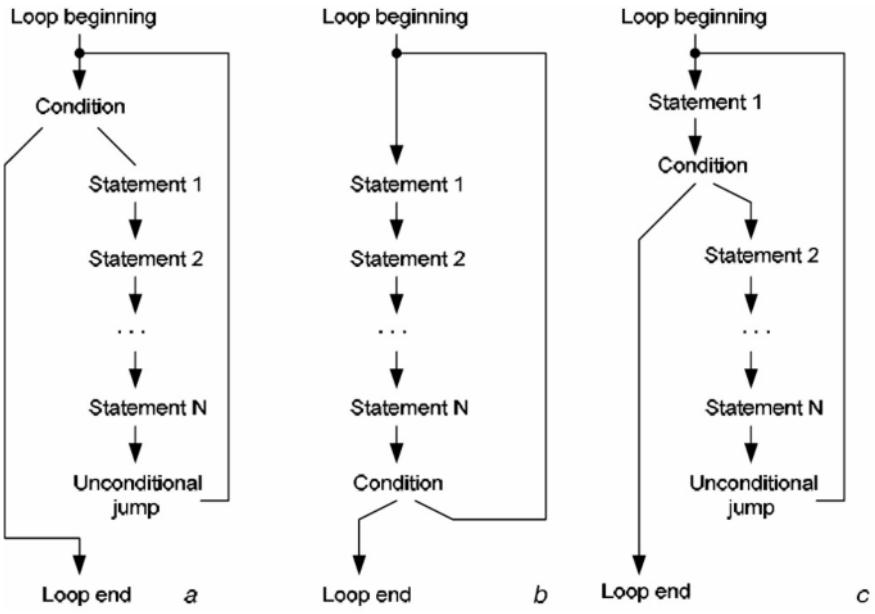
متغیرهای ایستا

متغیرهای ایستا همانند متغیرهای سراسری هستند با این تفاوت که حوزهٔ محدود شده‌ای دارند. آنها فقط از داخل تابعی که آنها را معرفی کرده است قابل دسترسی هستند. متغیرهای سراسری و ایستا همانند هم هستند. هر دو در بخش داده قرار می‌گیرند هر دو به‌صورت مستقیم آدرس‌دهی می‌شوند (جز در زمان آدرس‌دهی با ارجاع).

شناسایی حلقه های تکرار

حلقه‌ها در زبان‌های سطح بالا تنها ساختاری هستند که ارجاع به عقب دارند. همهٔ انواع پرشها مثل if-then-else یا switch که به‌صورت شرطی هستند حرکت به سوی جلو دارند. با دیدن درخت منطقی آنها در می‌یابیم که در نگاه اول می‌توان آنها را شناسایی کرد.

سه نوع اصلی از انواع حلقه‌ها وجود دارد: ۱- حلقه‌ها با شرط در ابتدا ۲- حلقه با شرط در انتها ۳- حلقه‌ها با شرط در وسط.



شکل(۷-۱۱)

شرطهای حلقه نیز دو نوع هستند: ۱- شرط اختتام حلقه ۲- شرط ادامه حلقه. در حالت اول اگر شرط اختتام حلقه درست باشد یک پرش به انتهای حلقه انجام می‌شود در غیراین صورت اجرای حلقه ادامه می‌یابد. در حالت دوم اگر شرط اجرای حلقه غلط باشد یک پرش به انتهای حلقه انجام می‌شود و در غیراین صورت اجرای حلقه ادامه می‌یابد. شرایط ادامه حلقه معکوس شرایط اختتام حلقه می‌باشند. پس کافی است که هر کامپایلر فقط یکی از آنها را پشتیبانی کند. دستورات do، while و for در زبان C فقط با شرطهای ادامه حلقه کار می‌کنند. تنها استثناء دستور repeat-until است که شرط اختتام حلقه را می‌پذیرد.

حلقه‌ها با شرط در ابتدا

در زبان C و پاسکال حلقه‌ها با شرط در ابتدا توسط دستور While (expression) فراهم آمده است که expression شرط ادامهٔ حلقه می‌باشد. برای مثال حلقهٔ

```
while (a<10)
a++
```

تا زمانی که شرط $a < 10$ درست باشد ادامه پیدا می‌کند. اما گاهی اوقات ممکن است که کامپایلر شرط ادامه حلقه را به شرط اختتام حلقه تبدیل کند. در کد زیر حلقه با شرط اختتام در سمت چپ قرار دارد و حلقه با شرط ادامه در سمت راست قرار دارد. می‌بینید که حلقه با شرط اختتام به اندازه یک دستور کوتاه‌تر است. پس همهٔ کامپایلرها بهتر می‌دانند که نوع سمت چپ را ایجاد کنند. بعضی از کامپایلرها نیز توانایی تبدیل حلقه‌های شرط در ابتدا را به حلقه‌های شرط در انتها دارند.

مثال: حلقه با شرط اختتام و ادامه

```
while:              while:

cmp a, 10           cmp a, 10
      jae end       jb continue
      inc A         jmp end
      jmp while     continue:
end:                inc a
                    jmp while
                    end:
```

حلقه‌هایی که دارای شرط اختتام هستند به‌طور مستقیم قابل شبیه‌سازی توسط دستور While نیستند. این اشتباه بسیار زیاد دیده می‌شود که مبتدیان این گونه می‌نویسند:

```
while (a >= 0)
a++
```

حلقه با چنین شرطی هیچ گاه اجرا نخواهد شد.

عملگرهای معکوس بدین گونه هستند: برای مثال معکوس کوچکتر از، بزرگتر مساوی می‌باشد. در جدول(۵-۸) لیست کامل عملگرهای معکوس را مشاهده می‌کنید.

عملگر منطقی	معکوس عملگر منطقی
==	!=
!=	==
>	<=
<	>=
<=	>
>=	<

جدول(۷-۲)

حلقه‌ها با شرط در انتها

در زبان C حلقه‌ها با شرط در انتها توسط دستور do-while پیاده سازی می شوند. در زبان پاسکال این کار توسط دستور repeat \ unit صورت می‌گیرد. حلقه‌ها با شرط در انتها به سادگی و بدون هیچ مشکلی به کد ماشین تبدیل می‌شوند.

برای مثال: حلقهٔ

```
do
a++;
while (a < 10)
```

کد کامپایل شده یک حلقه با شرط در انتها به کد زیر تبدیل می‌شود:

```
repeat :
inc a
cmp a,10
Jl repeat
end :
```

در مقایسه کد حلقه با شرط در انتها و کد حلقه با شرط در ابتدا نشان می‌دهد که حلقه‌ها با شرط در انتها سریع‌تر می‌باشند. کامپایلرهای مطمئن مانند ویژوال C++ توانایی تبدیل حلقه‌ها با شرط در ابتدا به حلقه‌ها با شرط در انتها را دارند. تفاوت اجرای حلقه‌ها با شرط در ابتدا و شرط در انتها این است که حلقه‌هایی که شرط آنها در انتها قرار دارد حتماً یک بار اجرا می‌شوند و زمانی که کامپایلر

مطمئن است که حلقه یک بار انجام می‌شود در هر محلی که مناسب می‌داند شرط را بررسی خواهد کرد. در بررسی مثال قبل شرط در ابتدا (a < b) است و زمانی که a= =b باشد حلقه اجرا نخواهد شد اما در تبدیل آن به حلقه با شرط در انتها حلقه یکبار اجرا می‌شود و برای رفع این مشکل در بررسی شرط می‌توانیم یک واحد به a اضافه کنیم do… while((a+1) < b) یا یک واحد از a کم کنیم do… while(a < (b-1)) علت این کار این است که در پردازش‌گرهای پنتیوم پرش به عقب (به آدرس‌های کوچکتر) بهینه‌سازی شده است و به این علت است که حلقه‌ها با شرط در انتها کمی‌سریع‌تر اجرا می‌شوند.

حلقه‌ها با شمارشگر

حلقه‌های با شمارشگر نوع مستقلی از حلقه‌ها نمی‌باشند و قاعده آنها همانند حلقه‌ها با شرط در ابتدا است. برای مثال حلقهٔ for (a = 0 ; a < 10 ; a++) همانند

```
a = 0;
while (a < 10)
(
.
.
.
a++;
)
```

می‌باشد. اما کد کامپایل شده آنها ممکن است یکسان نباشد. کامپایلرهایی که بهینه‌سازی کد انجام می‌دهند بعد از مقداردهی متغیر شمارشگر ، کنترل را به دستوری که شرط اختتام حلقه را بررسی می‌کند می‌فرستند.

مثال حلقه با شمارشگر

```
mov a, xxx
```

متغیر شمارشگر مقداردهی اولیه می‌شود.

```
jmp conditional
```

یک پرش برای آزمایش شرط ادامه حلقه انجام می‌شود.

```
repeat:
```

این قسمت شروع حلقه و در ادامه بدنه حلقه می‌باشد.

```
add a, xxx [sub a, xxx]
```

شمارشگر حلقه دستکاری می‌شود.

```
conditional:
cmp    a, xxx
```

شرط ادامه حلقه بررسی می‌شود.

```
jxx repeat
```

اگر شرط حلقه درست باشد پرش به ابتدای حلقه انجام می‌شود. کامپایلرهای مطمئن و پیش‌رفته سعی در تشخیص این مسئله دارند که آیا این حلقه حداقل برای یک بار انجام می‌شود یا خیر. اگر چنین چیزی رخ داد حلقهٔ for را به یک حلقه با شرط در انتها تبدیل می‌کنند.

مثال: تبدیل شمارشگر

```
mov a, xxx
```

متغیر شمارشگر مقداردهی اولیه می‌شود.

```
repeat:
```

این قسمت آغاز حلقه و بعد از آن بدنهٔ حلقه می‌باشد.

```
add a, xxx [sub a, xxx]
```

شمارشگر تغییر می‌کند.

```
cmp a, xxx
```

شرط ادامهٔ حلقه بررسی می‌شود.

```
jxx repeat
```

اگر شرط حلقه درست باشد پرشی به ابتدای حلقه انجام می‌شود. اکثر کامپایلرهای پیشرفته حلقه‌های افزایشی را به حلقه‌های کاهشی تبدیل می‌کنند. این کار زمانی انجام می‌شود که کامپایلر مطمئن شود از پارامتر حلقه فقط برای کنترل تعداد دفعات انجام حلقه استفاده می‌شود و در بدنه حلقه استفاده دیگری از آن نمی‌شود در چنین حالتی حلقه‌های افزایشی به کاهشی تبدیل می‌شوند حلقه‌های کاهشی بسیار کوتاهتر می‌باشند زیرا دستور DEC فقط مقدار عملوند را کاهش نمی‌دهد

بلکه هنگامی‌که به صفر می‌رسد پرچم صفر (Zero Flag) را یک می‌کند. در این حالت دستور CMP A,XXX غیرضروری خواهد بود.

مثال: تبدیل یک حلقه افزایشی به کاهشی.

```
mov a, xxx
```

متغیر شمارشگر مقدار دهی اولیه می‌شود.

```
repeat:
```

این قسمت ابتدای حلقه و بعد از آن بدنهٔ حلقه می‌باشد.

```
dec a
```

شمارشگر کاهش می‌یابد.

```
jnz repeat
```

این قست هم تکرار While A!=0 می‌باشد.

بسته به تنظیمات کامپایلر حلقه‌های for می‌توانند به حلقه‌هایی با شرط در ابتدا یا حلقه‌هایی با شرط در انتها تبدیل شوند. چنین مسائلی شناخت حلقه‌های for را بسیار مشکل می‌کند. تنها نوعی از حلقه‌هایی که شرط آنها در انتها قرار دارد مطمئناً قابل شناسایی هستند. در بقیه حالات هیچ روش خاصی برای شناخت for وجود ندارد. اگر منطق حلقه شناسایی شد ساده‌تر است که آن را از طریق حلقه for نمایش دهیم درغیراینصورت از while-do یا (repeat \ unil) برای نمایش حلقه با شرط در ابتدا یا در انتها استفاده می‌کنیم.

شناسایی ساختارهای کنترلی

دستورات IF-THEN-ELSE

به‌طورکلی دو نوع الگوریتم وجود دارد. شرطی و غیرشرطی. ترتیب اجرای عملیات در الگوریتم‌های غیرشرطی ثابت است و به داده‌های ورودی وابسته نمی‌باشد. برای مثال a = b + c. ولی ترتیب اجرای عملیات در الگوریتم‌های شرطی وابسته به داده‌های ورودی می‌باشد. برای مثال:

```
if C is Zero Then a= b/c , else send an error
```

کلمات کلیدی else, then, if انشعاب (Branch) نامیده می‌شوند. انشعاب‌ها قلب زبان‌های برنامه‌نویسی هستند و شناسایی صحیح آنها بسیار مهم است. ما به جزئیات پیاده‌سازی آنها در زبان‌های مختلف برنامه‌نویسی نمی‌پردازیم ولی ساختار کلی آنها به‌صورت زیر است.

```
IF (condition) THEN {statement1; statementN;} ELSE {statement11;
statement1M;}
```

وظیفهٔ کامپایلر این است که دستورات را به دستورات ماشین تبدیل کند. این کار باید به این صورت باشد که اگر شرط if درست بود Satement 1 تا Satement N اجرا شود اما اگر شرط غلط بود $Statement1_1$ تا $Statement1_M$ اجرا شود. ریزپردازنده‌های خانواده 80x86 تعداد کمی از دستورات، شرطی را پشتیبانی می‌کنند اما تعداد زیادی پردازنده وجود دارد که دستورات مناسبی را پشتیبانی می‌کنند برای مثال به جای نوشتن:

```
Test   ECX , ECX / JNZ  XXX / MOV  EAX , 0x666
```

می‌توانید بنویسید:

```
Test  ECX , ECX / If Z  MOV  EAX , 0x666
```

این دستور زمانی که پرچم Z صفر باشد، اجازه اجرای دستور Mov EAX , 0x666 را می‌دهد. ریز پردازنده‌های 80x86 را می‌توان با زبان BASIC مقایسه کرد که چیزی جز Goto برای پرش را نمی‌پذیرفت. در مثال زیر این موضوع را مشاهده می‌کنید.

```
IF A=B THEN PRINT "A=B" 10 IF A=B THEN GOTO 30
20 GOTO 40
30 PRINT "A=B"
40 ... // The rest of code
```

اکثر کامپایلرها حتی آنهایی که بهینه‌سازی که هم انجام می‌دهند، ساختار شرط را معکوس می‌کنند به عنوان مثال دستور

```
If (Conditions) then {Statements}
```

به شبه کد زیر تبدیل می‌شود

شبه کد ایجاد شده از انشعاب if-then.

```
IF (NOT condition) THEN continue
        statement1;
        ...
        statementN;
continue:

...
```

برای بازسازی کد اصلی برنامه، شرط باید معکوس شود و بلوک دستورات $\{Statement_1 : Statement_N\}$ به کلمهٔ کلیدی then چسبیده شود.

به مثال زیر توجه کنید:

بازسازی کد اصلی یک برنامه

```
10 IF A<>B THEN 30
20 PRINT "A=B"
30 ...// The rest of the code
```

در این حالت کد اصلی حاوی دستورات زیر است:

```
If A=B then Print "A+B"
```

کامپایلر شرط را معکوس کرده و کد زیر را ایجاد می‌کند.

```
10 IF A=B THEN 30
20 PRINT "A<>B"
30 ...// The rest of the code
```

حال به بررسی نحوه تبدیل عبارت:

```
If (Condition) then {Statement1 ; StatementN;} Else  {Statement11 ;
Statement1M;}
```

خواهیم پرداخت. بعضی کامپایلرها به این صورت عمل می‌کنند.

```
IF (condition) THEN do_it
statement1_1;
...
statement1_N;
GOTO continue:

do_it:
```

پرش if اجرا می‌شود.

```
statement_1;
...
statementM;
continue:
```

بعضی دیگر به این صورت تبدیل می‌کنند.

```
IF (NOT condition) THEN else
statement_1;
...
statement_M;
GOTO continue

else:
```

پرش else اجرا می‌شود.

```
statement1_1;
...
statement1_M;
continue:
```

کامپایلر اخیر مقدار شرط را معکوس می‌کند اما قبلی چنین کاری را انجام نمی‌داد. بنابراین بدون دانستن عملکرد کامپایلر دستیابی به کد اصلی برنامه غیرممکن خواهد بود. اما این مسئله مشکلی ایجاد نمی‌کند چون می‌توان شرط‌ها را به روش‌های گوناگونی نوشت. برای مثال اگر علاقه ندارید که از عبارت زیر استفاده کنید:

```
If (C <> 0) THEN a = b / c  ELSE  PRINT "ERROR"
```

می‌توانید از عبارت:

```
If (C= =0) THEN PRINT "ERROR"  ELSE a = b / c
```

استفاده کنید

انواع شرط‌ها

شرط‌ها می‌توانند ساده و ابتدایی یا پیچیده و ترکیبی باشند. نمونه‌ای از شرط ساده if(a= =b) است و نمونه‌ای از شرط‌های ترکیبی و پیچیده: if ((a= =b) && (a!=o)) می‌باشد. هر شرط ترکیبی و پیچیده قابل تبدیل به چندین شرط ساده و ابتدایی است.

شرط‌های ساده و ابتدایی دو نوع هستند:

۱- شرط‌های نسبی مانند بزرگتر، بزرگتر مساوی ،کوچکتر ، کوچکتر مساوی و مساوی.

۲- شرطی منطقی مانند: XOR , NOT , OR , AND یک عبارت درست یک مقدار بولین True باز می‌گرداند و یک عبارت غلط یک مقدار بولین False باز می‌گرداند. نمایش فیزیکی درونی مقادیر بولین بسته به پیاده‌سازی‌های مختلف فرق می‌کند. بطور معمول عبارت غلط با مقدار صفر نمایش داده می‌شود وعبارت درست با مقادیر غیرصفر نمایش داده می‌شود.

برای مثال If ((a>b)!=0) درست است و If ((a>b)= = 1) وابسته به پیاده‌سازی است و ممکن است با توجه به پیاده سازی درست یا نادرست باشد. به این مثال توجه کنید:

```
If ((666= =777)= = 0) Printf ("Test!")
```

فکر می‌کنید با این مثال چه چیز بر روی صفحه به نمایش در خواهد آمد؟ بدیهی است که عبارت "Test!" نمایش داده خواهد شد. هیچ یک از مقادیر 666 و 777 برابر صفر نیست اما 666=!777 است پس مقدار شرط (666=777) برابر false و صفر است. اما نوشتن If ((a=b)= =0) نتیجهٔ دیگری خواهد داد.

در اینجا مقدار b به a نسبت داده خواهد شد و تساوی آنها با صفر بررسی خواهد شد. از شرط‌های منطقی برای ترکیب شرط‌های ساده و ابتدایی استفاده می‌شود و با استفاده از این شرط‌های منطقی ، شرط‌های پیچیده و ترکیبی ایجاد می‌شود. کامپایلر در زمان کامپایل برنامه شرط پیچیده و ترکیبی برنامه را به شرط‌های ساده و ابتدایی تبدیل می‌کند و در مرحلهٔ بعد عبارت و دستورات شرطی با Goto جایگزین خواهد شد.

تبدیل شرط ترکیبی به ساده:

```
IF a!=b THEN continue
IF a==0 THEN continue
...// The code of the condition
:continue
... // The rest of code
```

چگونگی تبدیل شرط‌های ترکیبی به ساده برعهده خود کامپایلر و به صلاحدید خود آن خواهد بود اما تنها موضوع تضمینی این است که شرط‌های مقید به AND منطقی از چپ به راست بررسی می‌شوند و اگر اولین شرط درست نباشد بقیه بررسی نخواهد شد. این مسئله این اجازه را می‌دهد که مثالی به صورت زیر بنویسیم:

```
If ((filename) and (f = fopen(& filename[o], "rw")))
```

در این حالت اگر اشاره‌گر Filename به فضای اختصاص داده نشده‌ای اشاره کند دستور fopen صدا زده نخواهد شد.

حال به مشکلات شناسایی شرط‌های منطقی و تحلیل عبارات پیچیده می‌پردازیم. به مثال زیر توجه کنید:

```
If ((a= =b) && (a!=0))
```

حال ببینیم وقتی این کد کامپایل می‌شود چه رخ می‌دهد.

نتیجهٔ کامپایل عبارت If ((a= =b) && (a!=0))

```
IF a!=b THEN continue
IF a==0 THEN continue
...
...
...
:continu
...
```

دستور CMP: این دستور معادل دستور Sub برای اعداد طبیعی می‌باشد با یک اختلاف و آن هم اینکه دستور CMP برخلاف Sub مقدار عملوندها را دستکاری نمی‌کند. این دستور فقط برروی پرچم‌های پردازش‌گر اصلی تأثیر می‌گذارد. پرچم صفر، پرچم نقلی، پرچم علامت، پرچم سرریز. اگر نتیجهٔ کاهش صفر شود Zero flag ، یک می‌شود. پرچم نقلی زمانی یک می‌شود که یک رقم قرضی

از با ارزش‌ترین بیت مفروق‌منه گرفته شود. پرچم علامت برابر با ارزش‌ترین بیت حاصل قرار داده می‌شود.

بیت سرریز زمانی یک می‌شود که نتیجهٔ محاسبات بیشتر از تعداد بیت‌های موجود شود و جایی برای بیت علامت نباشد. برای بررسی و تست پرچم‌ها از پرش‌های شرطی استفاده می شود. نتایج حاصل از انجام کار بر روی اعداد طبیعی در جدول (۷-۳) قرار دارد:

Condition		The state of the flags			Instruction		
		Zero flag	Carry flag	Sign flag			
a == b		1	?	?			
a != b		0	?	?	JNZ	JNE	
a < b	Unsigned	?	1	?	JC	JB	JNAE
	Signed	?	?	!=OF	JL	JNGE	
a > b	Unsigned	0	0	?	JA	JNBE	
	Signed	0	?	==OF	JG	JNLE	
a >=b	Unsigned	?	0	?	JAE	JNB	JNC
	Signed	?	?	==OF	JGE	JNL	
a <=b	Unsigned	(ZF == 1) || (CF == 1)		?	JBE	JNA	
	Signed	1		? !=OF	JLE	JNG	

جدول (۷-۳) عملیات نسبی و دستورالعمل‌های پردازشگر آنها

CPU	7	6	5	4	3	2	1	0
	SF	ZF	–	AF	–	PC	–	CF
FPU	15	14	13	12	11	10	9	8
	Busy!	C3(ZF)	TOP			C2(PF)	C1	C0(CF)

جدول (۷-۴) عنوان ارتباط با پرچم‌های fpu با CPU

به‌طورکلی عبارت Then do-it (عبارت شرطی نسبی ساده) if به دستورات ماشین زیر ترجمه می‌شود.

```
cmp a, b
jxx do_it
continue:
```

دستورالعمل‌هایی که بر روی پرچم‌های پردازشگر تأثیری ندارند (مانند Lea یا Move) می توانند بین دستور CMP و پرش‌های شرطی قرار بگیرند.

دستورالعمل‌های شرطی Set: پردازنده‌های جدید اینتل شامل دستورالعمل‌هایی به‌منظور Set کردن یک بایت درهنگام وقوع یک شرط هستند (Set XX). این دستورالعمل‌ها در صورت وقوع شرط، عملوند خود را یک کرده و در غیر این صورت آن را با مقدار صفر پر می‌کنند. دستور Set XX به‌طور گسترده‌ای در کامپایلرهایی که از بهینه‌سازی کد استفاده می‌کنند به‌منظور بهینه‌سازی پرش‌ها و انشعاب‌ها استفاده می‌شود. زیرا پرش‌ها باعث تخلیه pipeline پردازش‌گر و کاهش کارایی سیستم می‌گردند.

Table 22: The Boolean Set-On-Condition Instructions

Instruction			Relationship	Condition
SETA	SETNBE	a>b	Unsigned	CF == 0 && ZF == 0
SETG	SETNLE		Signed	ZF == 0 && SF == OF
SETAE	SETNC SETNB	a>=b	Unsigned	CF == 0
SETGE	SETNL		Signed	SF == OF
SETB	SETC SETNAE	a<b	Unsigned	CF == 1
SETL	SETNGE		Signed	SF !=OF
SETBE	SETNA	a<=b	Unsigned	CF == 1 || ZF == 1
SETLE	SETNG		Signed	ZF == 1 || SF != OF
SETE	SETZ	a==b	—	ZF == 1
SETNE	SETNZ	a!=0	—	ZF == 0

جدول(۷-۵) دستورات بولین شرطی Set-on

دیگر دستورات شرطی

هشت دستور پرش شرطی وجود دارد. JS, JNP, JP, JNO-JO, JECXZ, JCXZ و JNS از اینها فقط JCXZ و JECXZ با عملیات مقایسه‌ای ارتباط مستقیم دارند. کامپایلرهایی که بهینه‌سازی کد انجام می‌دهند معمولاً ساختار CMP [E]CX ,0 \ JZ do-it را با معادل کوتاهتر آن J[E]CXZ do-it جایگزین می‌کنند.

از پرش‌های شرطی JO و JNS در کتابخانه‌های ریاضی برای پردازش اعداد بزرگ استفاده می‌شود. (برای مثال اعداد طبیعی ۱۰۲٤ بیتی). از پردازش‌های شرطی JS و JNZ به‌طور مکرر برای تست مقدار با ارزش‌ترین بیت استفاده می‌شود.

Table 23: The Auxiliary Conditional Jumps

Instruction	Jump if…	Flags
JCXZ	…the CX register equals zero.	CX == 0
JECXZ	…the ECX register equals zero.	ECX == 0
JO	…there is an overflow.	OF == 1
JNO	…there is no overflow.	OF == 0
JP/JPE	…parity of the least significant byte of the result is even.	PF == 1
JNP/JPO	…parity of the least significant byte of the result is odd.	PF == 0
JS	…the sign bit is set.	SF == 1
JNS	…the sign bit is cleared	SF == 0

جدول(۷-۶) پرش‌های شرطی کمکی

دستورات جابه جایی شرطی

پردازش‌گرهای پنتیوم از پنتیوم II دستورات جابه جایی شرطی CMOVXX را پشتیبانی می‌کنند. این دستور در صورت درست بودن شرط، مقداری را از منبع به مقصد می‌فرستد. این دستورالعمل‌ها از پرش‌های اضافی جلوگیری می‌کنند.

این مسئله را بر روی دستور If a<b then a=b بررسی می‌کنیم و سمت چپ این کار را با استفاده از پرش شرطی و در سمت راست با استفاده از جا به جایی شرطی انجام دادیم.

```
cmp a, b        cmp a, b
cmovb a, b       jae continue:
mov a, b
continue:
```

مقایسه‌های بولین

False منطقی برابر با مقدار صفر و True برابر با مقدار غیرصفر می‌باشد. برای مثال if (a) then do-it به عبارت if (a!=0) then do-it ترجمه می‌شود.

معمولاً اکثر کامپایلرها دستور CMPA , 0 را با دستورات کوتاه‌تری مثل Test A , A یا Or A , A جایگزین می‌کنند.در همهٔ حالت‌ها اگر A= =0 باشد Zero flag یک می‌شود و اگر A!=0 باشد Zero flag، صفر می‌شود.با این وجود کدی مانند Test EAX , EAX \ JZ do-it در کد disassemble شده نشان‌دهندهٔ مقایسه بولین می‌باشد.

ساختار ((Condition)? Do-it: Continue)

در زبان C عباراتی مانند a = (condition) ? Do-it:continue به if (condition) THEN a = do-it else a = continue تبدیل می‌شوند. البته حاصل کامپایل این دستورات همیشه یکسان نیست. عملگر " ? "در بهینه‌سازی کد انعطاف‌پذیرتر از if-then-else است. به مثال زیر توجه کنید:

شناسایی عملگرهای شرطی

```
main()
{
int a;
int b;

a=(a>0)?1:-1;

if (b>0)
       b=1;
else
       b=-1;

return a+b;
}
```

کد disassemble شده با استفاده از کامپایلرویژوال C++

```
push    ebp
mov     ebp, esp
```

قاب پشته باز می‌شود.

```
sub     esp, 8
```

فضا به متغیرهای محلی اختصاص داده می‌شود.

```
xor     eax, eax
```

EAX صفر می شود

```
cmp     [ebp+var_a], 0
```

متغیر با صفر مقایسه می‌شود.

```
setle   al
```

اگر Var_ a<= 0 باشد مقدار ox1 در a1 قرار داده می‌شود. اگر مقدار Var_a > 0 باشد مقدار صفر در a1 قرار داده می‌شود.

```
dec     eax
```

از EAX یک واحد کم می‌شود. حال اگر Var_a > 0 باشد مقدار EAX برابر 1- می‌شود. اگر Var_a <=0 باشد مقدار EAX صفر می‌شود.

```
and     eax, 2
```

تمام بیت‌ها بجز بیت دوم از چپ صفر می‌شوند. حال اگر Var_a > 0 باشد مقدار EAX برابر ۲ می‌شود. اگر Var_a <= 0 باشد مقدار EAX صفر می‌شود.

```
add     eax, 0FFFFFFFFh
```

مقدار 0x1 از EAX کم می‌شود. حال اگر Var_a > 0 باشد. مقدار EAX برابر ۱ می‌شود. اگر Var_a <= 0 باشد مقدار EAX برابر 1- می‌شود.

```
mov     [ebp+var_a], eax
```

حاصل در متغیر Var_a ریخته می‌شود. اینجا پایان عملگر ? و آغاز بخش IF-THEN-ELSE می‌باشد.

```
        cmp     [ebp+var_b], 0
```

مقدار متغیر Var_b با صفر مقایسه می‌شود.

```
        jle     short else
```

اگر Var_b <= 0 باشد پرش انجام می شود.

```
        mov     [ebp+var_b], 1
```

مقدار 1 به Var_b نسبت داده می‌شود.

```
        jmp     short continue
```

پرش به برچسب Continue انجام می‌شود.

```
else:                                   ; CODE XREF: _main+1D↑j
        mov     [ebp+var_b], 0FFFFFFFFh
```

مقدار 1- در متغیر Var_b نوشته می‌شود.

```
         continue:
```

این قسمت پایان بخش if-then-else می‌باشد. متذکر می‌شویم که پیاده‌سازی if-then-else ساده‌تر از عملگر ? است اما به دلیل داشتن پرش‌های شرطی کارآیی برنامه را کاهش می‌دهد.

```
        mov     eax, [ebp+var_a]
```

مقدار متغیر Var_a در EAX بارگذاری می‌شود.

```
        add     eax, [ebp+var_b]
```

مقدار متغیر Var_a با مقدار متغیر Var_b جمع می‌شود و حاصل در EAX ریخته می‌شود.

```
        mov     esp, ebp
        pop     ebp
```

قاب پشته بسته می‌شود.

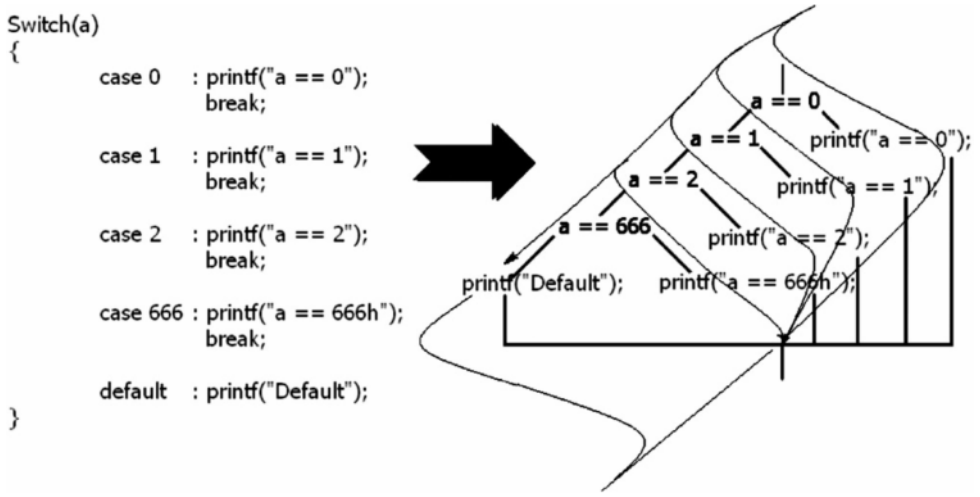
```
        retn
```

دستورات Switch-case-break

برای افزایش خوانایی برنامه دستور Switch در زبان C معرفی شد. در زبان پاسکال نیز دستور case پیاده‌سازی شد که انعطاف بیشتری از Switch در C دارد. به سادگی می‌توان ساختار کلی دستور Switch را نشان داد. دستور Switch معادل دستورات زیر است:

```
IF (a == x1) THEN statement1 ELSE IF (a == x2) THEN statement2
ELSE
IF (a == x3) THEN statement3 ELSE IF (a == x4) THEN statement4
ELSE…
```

نمایش درخت منطقی این بخش را در شکل (۵-۵) مشاهده می‌کنید:



شکل (۷-۱۲) ترجمه کلی دستور Switch

بدون ساختن درخت نمی‌توانیم در میان زنجیر بلند و طولانی تست مقادیر جستجو کرده و ساختار دستور Switch را مشخص کنیم. در دستور Switch فقط تساوی متغیرها با مقادیر بلافصل بررسی می‌شود. کامپایلرها دستور Switch را از بالا به پایین ترجمه می‌کنند. به مثال زیر با استفاده از ویژوال C++ توجه کنید.

کد disassemble شده دستور Switch با استفاده از کامپایلر ویژوال C++ :

```
main            proc near               ; CODE XREF: start+AF↓p

var_tmp         = dword ptr -8
var_a           = dword ptr -4

        push    ebp
        mov     ebp, esp
```

قاب پشته باز می‌شود.

```
        sub     esp, 8
```

فضا به متغیرهای محلی اختصاص داده می‌شود.

```
        mov     eax, [ebp+var_a]
```

مقدار متغیر Var_a در EAX بارگذاری می‌شود.

```
        mov     [ebp+var_tmp], eax
```

دستور Switch متغیر موقت خاص خود را ایجاد می‌کند. مقدار این متغیر در مقایسه‌ها ممکن است تغییر کند اما در چگونگی عملکرد این دستور اثری نخواهد داشت. برای جلوگیری از تداخل میان اینها از متغیر Var_tmp به عنوان Var_a استفاده می‌کنیم.

```
        cmp     [ebp+var_tmp], 2
```

مقدار متغیر Var_a با ۲ مقایسه می‌شود. در کد اصلی case با صفر شروع شده و با 0x666 خاتمه می‌یابد.

```
        jg      short loc_401026
```

اگر Var_a > 2 باشد پرش انجام می‌شود. چنین چیزی در کد اصلی وجود نداشته است. این پرش منتهی به فراخوانی تابع Printf نمی‌شود.

```
        cmp     [ebp+var_tmp], 2
```

مقدار متغیر Var_a با ۲ مقایسه می‌شود. مسیر و جریان کار کامپایلر مشخص است. این کار قبلاً انجام شده و هیچ پرچمی‌تغییر نکرده است.

```
        jz      short loc_40104F
```

اگر متغیر Var_a برابر ۲ باشد این پرش به فراخوانی تابع Printf ("a= = 2") منتهی می‌شود. این قسمت از کد از ترجمه بخش Case 2: Printf ("a= = 2") بدست آمده است.

```
cmp     [ebp+var_tmp], 0
```

مقدار متغیر Var_a با 0 مقایسه می‌شود.

```
jz      short loc_401031
```

اگر مقدار متغیر Var_a برابر صفر باشد این پرش به فراخوانی تابع Printf("a= = 0") منتهی خواهد شد. این قسمت از کد از ترجمه بخش Case 0 : Printf("a= = 0") بدست آمده است.

```
cmp     [ebp+var_tmp], 1
```

مقدار متغیر Var_a با ۱۰ مقایسه می‌شود.

```
jz      short loc_401040
```

اگر مقدار Var_a برابر ۱ باشد این پرش به فراخوانی تابع Printf("a= = 1") منتهی می‌شود. این قسمت از کد از ترجمه بخش Case 1: Printf("a= = 1") بدست آمده است.

```
jmp     short loc_40106D
```

این پرش به فراخوانی Printf("Default") منتهی خواهد شد. این قسمت از کد از ترجمه بخش default: Printf("Default") بدست آمده است.

```
loc_401026:                             ; CODE XREF: main+10↑j
```

اگر مقدار Var_a بزرگتر از ۲ باشد کنترل به این بخش منتقل خواهد شد.

```
Cmp [ebp+var_tmp] , 666H
```

مقدار متغیر Var_a با مقدار 666H مقایسه می‌شود.

```
jz      short loc_40105E
```

اگر مقدار متغیر Var_a برابر با 666h باشد این پرش به فراخوانی تابع Printf ("a= = 666h") منتهی خواهد شد.این قسمت از کد از ترجمه بخش Case 666 : Printf ("a= = 666h") بدست آمده است.

```
        jmp     short loc_40106D
```

این پرش به فراخوانی تابع Printf("Default") منتهی خواهد شد که از ترجمه بخش default : Printf ("Default") بدست آمده است.

```
loc_401031:                             ; CODE XREF: main+1C↑j
        ; printf("A == 0")
        push    offset aA0 ; "A == 0"
        call    _printf
        add     esp, 4
        jmp     short loc_40107A
```

در این قسمت اولین دستور break قرار دارد. از این قسمت کنترل به خارج از دستور Switch فرستاده می‌شود. اگر دستور break وجود نداشته باشد کلیه شاخه‌ها و بخش‌های case اجرا می‌شود.

```
loc_401040:                             ; CODE XREF: main+22↑j
        ; printf("A == 1")
        push    offset aA1 ; "A == 1"
        call    _printf
        add     esp, 4
        jmp     short loc_40107A

loc_40104F:                             ; CODE XREF: main+16↑j
        ; printf("A == 2")
        push    offset aA2 ; "A == 2"
        call    _printf
        add     esp, 4
        jmp     short loc_40107A

loc_40105E:                             ; CODE XREF: main+2D↑j
        ; printf("A == 666h")
        push    offset aA666h ; "A == 666h"
        call    _printf
        add     esp, 4
        jmp     short loc_40107A

loc_40106D:                             ; CODE XREF: main+24↑j main+2F↑j
        ; printf("Default")
        push    offset aDefault ; "Default"
        call    _printf
        add     esp, 4

loc_40107A:                             ; CODE XREF: main+3E↑j main+4D↑j
...
```

پایان دستور Switch.

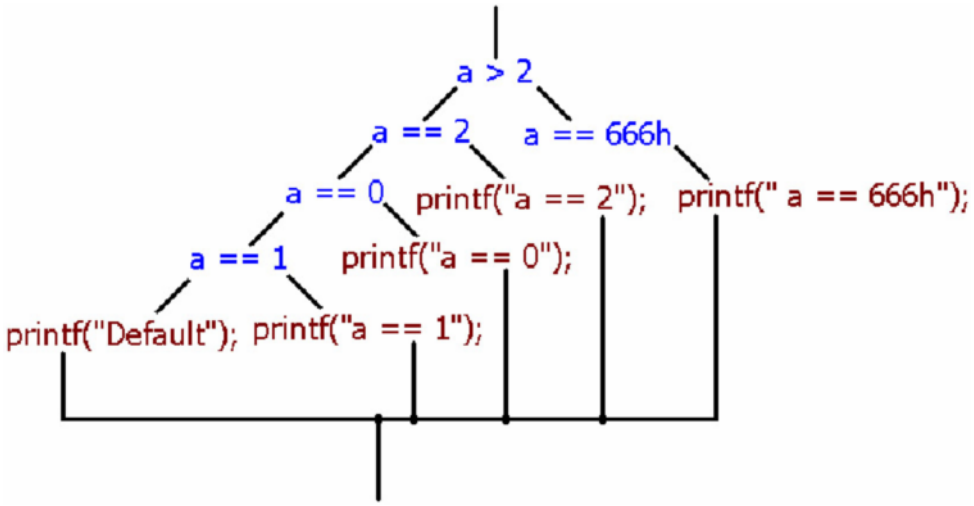
```
        mov     esp, ebp
        pop     ebp
```

قاب پشته بسته می‌شود.

```
        retnm
main            endp
```

با ساختن درخت منطقی شکلی شبیه شکل (۷-۱۳) ایجاد خواهد شد.

ابتدا به شرط a>2 توجه کنید که در کد اصلی وجود ندارد. بعد از آن به تغییر در ترتیب پردازش caseها توجه کنید. البته ترتیب فراخوانی‌های تابع Printf به همان صورتی که تعریف شده است می‌باشد. چرا کامپایلر این گونه رفتار می‌کند؟



شکل (۷-۱۳) ترجمه دستور Switch با استفاده از میکروسافت ویژوال C++

هدف از a>2 به سادگی قابل توضیح است: پردازش پشت سرهم همه caseهای Switch مناسب نیست. این کار برای ٤ یا ٥ case مناسب است. اما اگر صدها case باشد چه رخ خواهد داد؟ در این حالت پردازشگر در بررسی تک تک آنها از پای درخواهد آمد. به خاطر حل این مشکل کامپایلر درخت‌ها را فشرده می‌کند تا از طول آن کاسته شود و به جای ایجاد یک شاخه، دو شاخه ایجاد می‌کند کامپایلر اعداد کوچکتر از ۲ را در طرف چپ قرار می‌دهد و اعداد بزرگتر از ۲ را در طرف راست می‌گذارد. در این حالت بخش 666h از آخر به اول درخت منتقل شد. این روش بهینه‌سازی جستجوی مقدار، الگوریتم فورک نامیده می‌شود.

کامپایلر حق تغییر در ترتیب اجرای caseها را دارد. در استانداردها چیزی در این باره گفته نشده است و هر پیاده‌سازی در این باره به گونهٔ خاص خود عمل می‌کند. اما اگر شرط case درست بود دستورات بعد از آن باید دقیقاً به همان صورتی که تعریف شده‌اند قرار بگیرند.

با قبول کردن مطالب زیر می‌توانیم دستور Switch را شناسایی کنیم:

در صورت حذف نود شرطی مرکزی و ترکیب زیر نودهای چپ با راست و رسیدن به یک درخت شرطی معادل، مطمئناً با یک دستورالعمل Switch روبرو هستیم.

در مثال بالا نودهای شرطی در زیر شاخه‌های چپ (a = = 2) ، (a = = 0) و (a = = 1) و در زیر شاخه‌های سمت راست (a= =0x666) است. بدیهی است که در صورتی که a= = 0x666 باشد مطمئناً a =! 1 و a =! 0 است.

در نتیجه بدون هیچ مشکلی می‌توانیم زیر شاخهٔ راست را با زیر شاخه‌های چپ ترکیب کرده و درخت معادلی بدست آوریم. پس از انجام این عملیات درخت مذکور ساختار مناسبی برای یک دستور Switch خواهد داشت.

شناسایی آرایه‌ها و اشیاء

ساختار داخلی اشیاء در زبان C همانند رکوردها می‌باشد. در این قسمت روش‌های شناسایی هر دوی آنها را بررسی خواهیم کرد. رکوردها در میان برنامه‌نویسان محبوب تر از اشیاء هستند. آنها به برنامه‌نویسان اجازه می‌دهند تا داده‌های وابسته به هم را در یک‌جا جمع کرده و برنامه را خواناتر و قابل فهم‌تر سازند. شناخت رکوردها در طول disassemble کردن یک کد باعث آسان‌تر شدن تجزیه و تحلیل آن می‌شود. مهم‌ترین مشکل این است که رکوردها فقط در کدهای برنامه هستند و در زمان کامپایل به‌طور کامل از بین رفته و اجزاء آنها همانند متغیرهای معمولی هیچ‌گونه وابستگی به یکدیگر ندارند.

حال به مثال زیر توجه کنید:

حذف رکوردهادر زمان کامپایل :

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

struct zzz
{
	char s0[16];
	int a;
	float f;
};

func(struct zzz y)
```

بهتر است که از فرستادن رکورد با مقدار به یک تابع جلوگیری کنیم. اما در این قسمت عمداً این کار را انجام می‌دهیم تا بتوانیم عملیات مخفیانه ایجاد یک متغیر محلی را به نمایش بگذاریم.

```
{
    printf("%s %x %f\n", &y.s0[0], y.a, y.f);
}


main()
{
	struct zzz y;
	strcpy(&y.s0[0], "Hello, Sailor!");
	y.a = 0x666;
	y.f = 6.6;
	func(y);
```

```
}
```

کد disassemble شده حذف رکورها در زبان کامپایل

```
main          proc near           ; CODE XREF: start+AF↓p
var_18        = byte ptr -18h
var_8         = dword ptr -8
var_4         = dword ptr -4
```

اعضای رکورد از متغیرهای محلی معمولی قابل تمایز نیستند.

```
push   ebp
mov    ebp, esp
sub esp, 18h
```

فضایی در پشته اختصاص داده می‌شود.

```
push   esi
push   edi
push   offset aHelloSailor ; "Hello, Sailor!"

lea eax, [ebp+var_18]
```

این یک اشاره‌گر به متغیر محلی var_18 است. متغیر بعد از این در offset هشت قراردارد. از این جهت 0x18-0x8=0x10 که معادل ۱۶ بایت است. این حجم فضایی است که متغیر var_18 اشغال می‌کند و مشخص می‌کند که این متغیر از نوع رشته‌ای نیست.

```
push   eax
call   strcpy
```

رشته از بخش داده در متغیر محلی که یکی از اعضای رکورد است کپی می‌شود.

```
add    esp, 8
mov    [ebp+var_8], 666h
```

مقدار 0x666 به متغیری از نوع DWord اختصاص اختصاص داده می‌شود.

```
mov    [ebp+var_4], 40D33333h
```

این مقدار معادل 6.6 در فرمت Float می‌باشد.

```
sub esp, 18h
```

در اینجا اختصاص فضا توسط کامپایلربه یک متغیر محلی موقتی برای فرستادن اعضای رکورد با مقدار به یک تابع را مشاهده می‌کنید.

```
        mov     ecx, 6
```

Double word 6 که معادل ۲٤ بایت است کپی خواهد شد. ۱٦ بایت برای متغیرهای رشته‌ای (string) و ٤ بایت برای هر کدام از متغیرها ، از نوع float , int اختصاص می‌یابد.

```
        lea esi, [ebp+var_18]
```

اشاره‌گر به رکوردی که قصد کپی برداری از آن را داریم به دست می‌آوریم.

```
        mov     edi, esp
```

اشاره گر به متغیر محلی موقتی ایجاد شده به دست آمد.

```
repe movsd

        call    func
```

اشاره گر متغیر محلی موقتی فرستاده نمی‌شود زیرا در بالای پشته قرار دارد و در حقیقت قبلاً فرستاده شده است.

```
        add     esp, 18h
        pop     edi
        pop     esi
        mov     esp, ebp
        pop     ebp
        retn
main            endp
```

حال رکورد را با تعریف اعضاء داخلی آن جایگزین می‌کنیم.

مقایسه بین رکوردها و متغیرهای معمولی

```
main()
{
        char s0[16];
        int a;
        float f;

        strcpy(&s0[0], "Hello, Sailor!");
        a=0x666;
        f=6.6;
```

```
}
```

حال این مثال را با کد مرحله قبل مقایسه می‌کنیم. مقایسه بین رکوردها و متغیرهای معمولی

```
main            proc near           ; CODE XREF: start+AF↓p

var_18          = dword ptr -18h
var_14          = byte ptr -14h
var_4           = dword ptr -4
```

تفاوت‌هایی را در اینجا مشاهده می‌کنید. متغیرهای محلی به‌ترتیبی در پشته قرار می‌گیرند که کامپایلر ترجیح می‌دهد نه به‌صورتی که در کد برنامه تعریف شده اند. اما اعضاء یک رکورد باید به‌صورتی که دربرنامه تعریف شده‌اند در پشته قرار گیرند. به هر حال در زمان disassemble کردن، ترتیب متغیرها ناشناخته است و ما نمی‌توانیم تشخیص دهیم که آیا درست به ترتیب هستند یا خیر؟

```
push    ebp
mov     ebp, esp
sub     esp, 18h
```

0x18 بایت از پشته همانند مثال قبل اختصاص داده می‌شود.

```
push    offset aHelloSailor ; "Hello, Sailor!"
lea eax, [ebp+var_14]
push    eax
call    strcpy
add     esp, 8
mov     [ebp+var_4], 666h
mov     [ebp+var_18], 40D33333h
```

کدها بایت به بایت همانند هم هستند و نمی‌توان بین یک رکورد و یک دسته از متغیرهای محلی معمولی تمایزی قائل شد.

```
mov     esp, ebp
pop     ebp
retn
main                        endp

func            proc near           ; CODE XREF: main+36↑p

var_8           = qword ptr -8
arg_0           = byte ptr 8
arg_10          = dword ptr 18h
arg_14          = dword ptr 1Ch
```

اگرچه فقط یک آرگومان که همان نمونه‌ای از کورد است به تابع فرستاده می‌شود اما در کد disassemble شده نمی‌توانیم بین این عمل و انتقال ترتیبی تعدادی از متغیرها محلی به پشته تمایزی قائل شویم. پس نمی‌توانیم پیش تعریف اصلی تابع را بازسازی کنیم.

```
push   ebp
mov    ebp, esp
fld [ebp+arg_14]
```

عددی از نوع Floating point که در آفست 0x14 نسبت به اشاره‌گر EAX قرار دارد در پشته FPU بارگذاری می‌شود.

```
sub esp, 8
```

۸ بایت برای متغیرهای محلی اختصاص داده می‌شود.

```
fstp [esp+8+var_8]
```

مقدار Floating – point در متغیر محلی ذخیره می‌شود.

```
mov    eax, [ebp+arg_10]
push   eax
```

متغیر Real ذخیره شده ، خوانده شده و در پشته قرار می‌گیرد.

```
lea    ecx, [ebp+arg_0]
```

اشاره‌گر به اولین آرگومان به دست می‌آید.

```
push   ecx
push   offset aSXF        ; "%s %x %f\n"
call   printf
add    esp, 14h
pop    ebp
retn
func           endp
```

به‌نظر می‌رسد که نخواهیم توانست تمایزی میان یک رکورد و متغیرهای معمولی قائل شویم. اما برخی تحلیل‌گران کد با یافتن ارتباطات بین داده‌ها می‌توانند این کار را انجام دهند اما بعضی اوقات آنها نیز دچار اشتباه می‌شوند و به درستی نمی‌توانند ساختار کد منبع را بازسازی کنند.

علت این است که نمونه‌هایی که از یک رکورد ایجاد می‌شوند در زمان کامپایل و ترجمه همانند متغیرهای مستقل عمل می‌کنند و ارتباطی برای آنها در نظر گرفته نمی‌شود. آنها به‌طور واقعی آدرس‌دهی می‌شوند و گاهی آدرس‌دهی آنها به‌طور غیرمستقیم می‌باشد. اما در این حالت که یک نمونه از رکورد می‌باشد هر حوزه یک اشاره گر به آن نمونه از رکورد دارد و تمام فراخوانی‌های اعضای نمونه رکورد از طریق این اشاره‌گر صورت می‌گیرد.

احتمالاً شما می‌دانید که اعضای آرایه هم به همین صورت آدرس‌دهی می‌شوند. اشاره‌گر پایه به شروع آرایه اشاره می‌کند و offset لازم برای شماره عنصر خاصی از آرایه در صورت نیاز اضافه می‌شود و حاصل این محاسبات اشاره‌گر واقعی به یک عنصر از آرایه خواهد بود.

فرق اساسی بین آرایه و رکورد این است که عناصر آرایه از یک نوع هستند اما عناصریک رکورد از انواع متفاوتی هستند. بنابراین آرایه‌ها و رکوردها به سادگی از طریق آدرس‌دهی مکان‌های حافظه توسط اشاره‌گر مشترک پایه و شناسایی نوع متغیرها قابل شناسایی هستند. اگر با بیش از یک نوع متغیر مواجهه شدیم احتمالاً بایک رکورد سر و کار داریم و اگر تنها یک نوع را مشاهده کردیم تفاوتی میان رکورد و آرایه نیست و بر حسب شرایط باید قضاوت کنیم.

حال فرض کنیم که یک برنامه‌نویس می‌خواهد میزان مصرف چای روزانه خود در روزهای هفته را ذخیره کند. او ممکن است از یک آرایه به‌صورت day [7] استفاده کند یا اینکه از یک رکورد به صورت struct week {int Monday; int Tuesday: …} برای ذخیره اطلاعات استفاده کند. اما در هر دو حالت کد ایجاد شده توسط کامپایلر یکسان خواهد بود. نه فقط کد بلکه مفهوم هم یکسان خواهد بود. در این قسمت از نظر فیزیکی تفاوتی میان آرایه رکورد نیست و به‌نظر و انتخاب خود برنامه‌نویس بستگی دارد. در ذهن داشته باشید که آرایه‌ها معمولاً طول زیادی دارند و هنگامی‌که عناصر آنها آدرس‌دهی می‌شوند همیشه با عملیات ریاضی متنوعی برروی اشاره‌گر مواجه هستیم. به‌علاوه به‌طور معمول عناصر آرایه در یک حلقه مورد پردازش قرار می‌گیرند. اما عناصر یک رکورد به صورت تک تک برداشته می‌شوند. چیزی که در اینجا ناخواسته است این است که C++ و C به سادگی اجازه تغییرنوع را میان انواع مختلف می‌دهند. ما زمانی که با کد disassemble شده مواجهه هستیم نمی‌دانیم که با یک رکورد سر و کار داریم یا یک آرایه که به‌طور دستی در آن تغییر نوع داده شده است. برای رفع این مشکل مسئله را به این صورت درنظر می‌گیرم که چنین تغییر نوعی آرایه را به رکورد تبدیل می‌کند چون آرایه در تعریف خود یک نوع را پشتیبانی می‌کند و نمی‌تواند داده‌هایی از انواع مختلف را نگه دارد.

حال مثال قبل را به این صورت تغییرمی‌دهیم که یک اشاره‌گر را به تابع می‌فرستیم نه یک رکورد، و می‌خواهیم ببینیم که کامپایلر چه کدی را ایجاد خواهد کرد.

فرستادن اشاره‌گر رکورد به یک تابع:

```
funct         proc near           ; CODE XREF: sub_0_401029+29↓p

var_8         = qword ptr -8
arg_0         = dword ptr 8
```

تابع فقط یک آرگومان می‌گیرد.

```
push    ebp
mov     ebp, esp
mov     eax, [ebp+arg_0]
```

در این قسمت در حال بارگذاری آرگومان‌های ارسال شده، در ثبات EAX هستیم.

```
fld dword ptr [eax+14h]
```

مقدار floating – point که در offset، 0x14 نسبت به اشاره‌گر EAX قرار دارد در پشته FPU بارگذاری می‌شود.

اولاً EAX که همان آرگومان فرستاده شده به تابع می‌باشد یک اشاره‌گر است. دوماً EAX یک اشاره‌گر ساده نمی‌باشد بلکه ازنوع پایه است و برای دسترسی به دیگر عناصر یک رکورد یا آرایه مورد استفاده قرار می‌گیرد.

```
sub     esp, 8
```

هشت بایت برای متغیرهای محلی اختصاص داده می‌شود.

```
fstp [esp+8+var_8]
```

در این قسمت در حال ذخیره کردن مقدار واقعی هستیم که از متغیر محلی var_8 خواندیم.

```
mov     ecx, [ebp+arg_0]
```

مقدار اشاره گری که به تابع فرستاده شده در ECX بارگذاری می‌شود.

```
mov     edx, [ecx+10h]
```

مقداری که در offset ، 0x10 قرار دارد، در EDX بارگذاری می‌شود. این مقدار مطمئناً از نوع floating–point نمی‌باشد. پس ما با یک رکورد سر و کار داریم.

```
        push    edx
```

مقدار قبلی خوانده شده در پشته وارد می‌شود.

```
        mov     eax, [ebp+arg_0]
        push    eax
```

در این قسمت اشاره‌گر به رکورد که همان اشاره‌گر به اولین عضو آن است را به دست آوردیم و آن را در پشته قرار دادیم. نزدیک ترین عنصر در offset، 0x10 قرار دارد. اولین عنصر این رکود تمام 0x10 بایت را اشغال کرده است. بقیه اعضای رکورد مورد استفاده واقع نمی‌شوند. حال تقریباً می‌توانیم ساختار رکورد را حدس بزنیم.

```
struct xxx{
char x[0x10] || int x[4] || ___int16[8] || __int64[2];
int y;
float z;
}
```

```
        push    offset aSXF         ; "%s %x %f\n"
```

رشته مورد استفاده برای فرمت بندی رشته به ما این اجازه را می‌دهد که نوع داده را ثابت کنیم. اولین عنصر بدون شک char x[x010] است زیرا خروجی آن یک رشته بوده و فرض اولیه ما که این یک رکورد است، درست می‌باشد.

```
        call    printf
        add     esp, 14h
        pop     ebp
        retn
funct           endp

main            proc near           ; CODE XREF: start+AF↓p

var_18          = byte ptr -18h
var_8           = dword ptr -8
var_4           = dword ptr -4
```

در نگاه اول به نظر می‌رسد که با چندین متغیر محلی سر و کار داریم.

```
        push    ebp
        mov     ebp, esp
        sub     esp, 18h
```

پشته باز است یعنی برای ذخیره کردن اطلاعات جا دارد.

```
push    offset aHelloSailor; "Hello, Sailor!"
lea     eax, [ebp+var_18]
push    eax
call    unknown_libname_1
```

unknown_libname_1 یک strcpy است و ما این را بدون تجزیه و تحلیل کد، به سادگی در می‌یابیم. این تابع دو آرگومان می‌گیرد. یک اشاره گر به 0x10 بایت از بافر محلی که این اندازه از کاهش offset نزدیک‌ترین متغیر از offset آفست این متغیرنسبت به فریم پشته به دست آمده است. تابع strcmp نیز چنین پیش تعریفی دارد اما این تابع نمی‌تواند strcmp باشد زیرا بافر محلی مقداردهی اولیه نشده است و فقط می‌تواند یک بافر دریافتی باشد.

```
add     esp, 8
```

آرگومان‌ها از پشته برداشته می‌شوند.

```
mov     [ebp+var_8], 666h
```

مقداردهی اولیه متغیر محلی var_8 از نوع Dword.

```
mov     [ebp+var_4], 40D33333h
```

این قسمت مقداردهی اولیه متغیر var_4 می‌باشد که نوع آن را نمی‌دانیم.اما نوع آن بسیار شبیه Dword است. حال با بررسی و تحلیل عملکرد تابعی که این متغیر به آن فرستاده شده است در می‌یابیم که این متغیر یک مقدار floating point با اندازه ٤ بایت است.

```
lea     ecx, [ebp+var_18]
push    ecx
```

در اینجا به مسئله اساسی رسیدیم. تابع، یک اشاره‌گر به متغیر محلی var_10 که یک بافر رشته‌ای با اندازه 0x10 بایت است را دریافت می‌کند. اما با بررسی و تحلیل تابع فراخوانده شده متوجه می‌شویم که این تابع نه فقط0x10 بایت اول پشته پدر بلکه کل 0x10 بایت آن را آدرس‌دهی می‌کند. پس اشاره‌گر فرستاده شده به تابع یک اشاره‌گر به بافر رشته نمی‌باشد بلکه اشاره گر به یک رکورد است که به تابع فرستاده شده است.

```
struct x{
        char var_18[10];
        int var_8;
        float var_4
}
```

از آنجایی که نوع داده‌ها فرق می‌کند، این ساختار یک رکورد است نه یک آرایه.

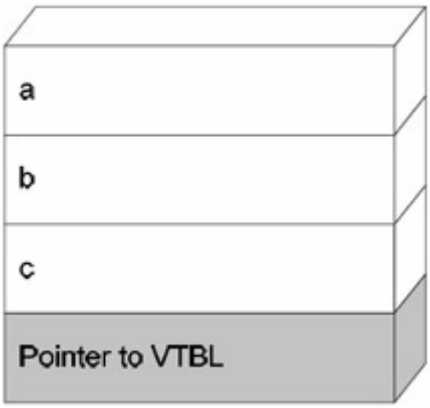
```
        call    funct
        add     esp, 4
        mov     esp, ebp
        pop     ebp
        retn
sub_0_401029  endp
```

شناسایی اشیاء

اشیاء در زبان C++ در واقع همان رکوردها هستند که شامل داده‌ها، متدهای پردازشی (توابع) و خصوصیات حفاظتی (مانند Friend, Public ...) می‌باشند. عناصر داده‌ای یک شیء همانند اعضای معمولی رکورد توسط کامپایلر مورد پردازش قرار می‌گیرند. توابع غیرمجازی توسط offset خود فراخوانی می‌شوند و مانند این است که در اشیاء قرار ندارند. توابع مجازی توسط یک اشاره‌گر خاص به جدول مجازی که در شیء قرار دارد فراخوانی می‌شوند و خصوصیات حفاظتی در زمان کامپایل از بین رفته و پاک می‌شوند. برای تمایز قائل شدن میان یک تابع protected وpublic می‌توانیم این موضوع را یادآور شویم که توابع protected فقط از درون شیء خود صدا زده می‌شوند اما یک تابع public از طریق دیگراشیاء نیز فراخوانی می‌شود.

```
class MyClass{
        void demo_1 (void);
        int a;
        int b;

public:
        virtual void demo_2(void);
        int c;
};

MyClass zzz;
```

یک نمونه شیء توسط کامپایلر به ساختار شکل (۷-۱٤) خرد می‌شود.



شکل(۷-۱٤) نمایش یک شیء در حافظه

حال این سوال پیش می‌آید که چگونه باید بین اشیاء و یک رکورد ساده تمایز قائل شویم؟چگونه می‌توانیم اندازه شیء را پیدا کنیم؟ چگونه تشخیص دهیم که چه تابعی به چه شیء تعلق دارد؟ حال به ترتیب به این سوالات پاسخ می‌دهیم عموماً غیر ممکن است که بتوانیم بین شیء و رکورد تمایز قائل شویم. به طور پیش فرض یک شیء یک رکورد است که فقط عناصر private (خصوصی) دارد. برای تعریف اشیاء می‌توانیم از کلمات کلیدی struct یا class استفاده کنیم.

برای کلاس‌هایی که اعضای حفاظت شده ندارند بهتر است از struct استفاده کنیم زیرا اعضای یک رکورد به‌طور پیش‌فرض public هستند. این دو مثال را با هم مقایسه کنید:

تعریف شیء با استفاده از کلمات کلیدی struct و class.

```
struct MyClass{
       void demo(void);
       int x;
private:
       void demo_private(void);
       int y;
};
class MyClass{
       void demo_private(void);
       int y;
public:
 void demo(void);
       int x;
};
```

کدهایی که مشاهده کردید از نظر نحوی با هم فرق دارند اما کدی که در نهایت توسط کامپایلر ایجاد می‌شود برای هر دوی آنها یکسان است. هر رکوردی که حداقل شامل یک تابع باشد را به عنوان یک شیء در نظر می‌گیریم.

چگونه دریابیم که چه تابعی مربوط به چه شیء است؟ برای توابع مجازی این کار نسبتاً ساده است زیرا آنها به‌صورت غیرمستقیم از طریق اشاره‌گر به جدول مجازی (Virtual table) فراخوانی می‌شوند. کامپایلر این اشاره‌گر را در تمام نمونه‌های اشیاء برای ارتباط با توابع مجازی آن قرار می‌دهد.

اما توابع معمولی با استفاده از آدرس واقعی خود فراخوانی می‌شوند دقیقاً همانند توابعی که متعلق به هیچ شیء نیستند. هر تابع یک شیء، یک آرگومان مجازی دارد که همان اشاره‌گر this است. این اشاره‌گر مرتبط با نمونه شیء آن تابع است. نمونه شیء، خود شیء نیست بلکه چیزی وابسته به آن است. پس شناسایی ساختار اولیه اشیاء در کد disassemble شده آن ممکن است.

اندازه اشیاء نیز با استفاده از همین اشاره‌گر this مشخص می‌شود. این اندازه حاصل تفاوت بین اشاره‌گر همجوار this و خود آن در پشته یا در بخش داده می‌باشد. اگر نمونه شیء با استفاده از علگر new ساخته شده باشد کد شامل فراخوانی از تابع new است که تعداد بایت‌هایی از حافظ را که باید اختصاص دهد به عنوان آرگومان می‌گیرد. این اندازه دقیقاً همان اندازه شیء است. به این نکته توجه داشته باشید که اکثر کامپایلرها وقتی نمونه شیء را ایجاد می‌کنند اگر داده یا توابع مجازی نداشته باشد حداقل میزان حافظه را که ۱ بایت است به آن اختصاص می‌دهند. اما باید بدانیم که حداقل میزان حافظه بسته به پیاده‌سازی‌های مختلف فرق دارد. حداقل فضای قابل اختصاص به پیاده‌سازی‌های heap بستگی داشته که از ٤ بایت تا ٤ کیلو بایت متغیر است. یعنی نمی‌توان ۱بایت از heap را اختصاص داد بلکه بلوک، بسته به پیاده‌سازی heap اختصاص می‌یابد. این حافظه حتماً باید اختصاص داده شود. زیرا کامپایلر باید اشاره‌گر this را تعریف کند و این اشاره‌گر نمی‌تواند از نوع null باشد چون در این صورت با اولین فراخوانی یک خطا (exception) رخ می‌دهد و همچنین عملگر delete باید چیزی را پاک کند یا از بین ببرد. پس ابتدا باید چیزی اختصاص داده شود تا بعد بتوان آن را از بین برد.

حال به ارائه یک مثال می‌پردازیم:

شناسایی یک شیء و ساختار آن :

```
#include <stdio.h>

class MyClass{
public:
       void demo(void);
```

```
        int x;
private:
        demo_private(void);
        int y;
};

void MyClass: :demo_private(void)
{
        printf ("Private\n");
}

void MyClass: :demo(void)
{
        printf("MyClass\n");
        this->demo_private();
        this->y=0x666;
}

main()
{
        MyClass *zzz = new MyClass;
        zzz->demo();
        zzz->x=0x777;
}
```

کد disassemble شده این برنامه به صورت زیر است :

کد disassemble شده شناسایی اشیاء و ساختار آنها

```
main            proc near               ; CODE XREF: start+AF↓p
        push    esi
        push    8
        call    ??2@YAPAXI@Z            ; operator new(unit)
```

با استفاده از عملگر new ، ۸ بایت از حافظه را به نمونه شیء اختصاص می‌دهیم. هنوز قطعاً معلوم نیست که حافظه برای یک شیء اختصاص داده شده است و ممکن چیزی شبیه به این مثال باشد. Char * x = new char [8] این موضوع به‌عنوان یک فرض بوده و اصل نمی‌باشد. تحلیل‌های بعدی چگونگی بررسی صحت این مساله را نشان می‌دهند.

```
        mov     esi, eax
        add     esp, 4

        mov     ecx, esi
```

اشاره this آماده شده و از طریق ثبات به تابع فرستاده می‌شود. ECX یک اشاره گر به نمونه شیء است.

```
        call    demo
```

در این قسمت به فراخوانی تابع demo رسیدیم. هنوز معلوم نیست که این تابع چه کاری انجام می‌دهد. اما می‌دانیم که متعلق شیء ای است که ECX به آن اشاره می‌کند. این نمونه را a می‌نامیم. تابعی که تابع demo را فراخوانی کرده است متعلق به a نمی‌باشد چون خود این تابع شیء را ایجاد کرده است. تابع demo یک تابع public است.

```
        mov     dword ptr [esi], 777h
```

یادآوری می‌کنیم که ESI به نمونه شیء اشاره می‌کند و در می‌یابیم که یک عضو دیگر Public نیز در این شیء وجود دارد که یک متغیر از نوع int می‌باشد.

```
        class myclass{
        public:
        void demo(void)

        int x;
        }
```

علت نوشتن void این است که این تابع نه چیزی می‌گیرد نه چیزی باز می‌گرداند.

```
        pop     esi
        retn
main            endp

demo            proc near                   ; CODE XREF: main+F↑p
```

حال در تابع demo که عضوی از شیء a است قرار داریم.

```
        push    esi
        mov     esi, ecx
```

اشاره گر this در ثبات ECX بارگذاری می‌شود.

```
        push    offset aMyclass             ; "MyClass\n"
        call    printf
        add     esp, 4
```

یک تابع دیگر فراخوانی می‌شود. این تابع از اعضای شیء می‌باشد و احتمالاً خصوصیت private دارد چون از طریق یک تابع شیء فراخوانی شده است.

```
        mov     ecx, esi
        call    demo_private
```

یک متغیر دیگر در این شیء وجود دارد این متغیر نیز احتمالاً private است. برطبق دانسته‌های کنونی ساختار شیء باید به صورت زیر باشد.

```
class myclass{
void demo_provate(void);
int y;
public:
void demo(void);
int x;
}
```

نتوانسته‌ایم شیء را شناسایی کنیم اما ساختار آن را کشف کرده‌ایم. نمی‌توانیم تضمین کنیم که این کار خالی از هرگونه خطا و اشتباه است برای مثال در مورد خصوصیت private تابع demo_private و متغیر y به این اصل بسنده کردیم که چون این‌ها از درون هیچ تابع دیگری خارج از شیء a فراخوانی نشده‌اند پس باید دارای خصوصیت private باشند. با این فرضیات تقریباً می‌توان کد برنامه را دوباره بازسازی کرد.

```
        pop     esi
        retn
demo            endp

demo_private proc near                  ; CODE XREF: demo+12↑p
```

این تابع demo_private می‌باشد.

```
        push    offset aPrivate         ; "Private\n"
        call    printf
        pop     ecx
        retn
demo_private    endp
```

اشیاء و نمونه‌ها

در کدی که توسط کامپایلر ایجاد می‌شود شیء وجود ندارد بلکه فقط نمونه شیء وجود دارد. به نظر می‌رسد که تفاوتی بین این دو نباشد. اما یک تفاوت اساسی بین شیء و نمونه شیء وجود دارد. شیء نمایش‌گر یک کلاس است و نمونه شیء که در کد ایجاد می‌شود یک زیر ساخت از آن است.

فرض کنید که کلاسی به نام A داریم و این کلاس دارای دو تابع a1 , a2 می‌باشد. دو نمونه از این کلاس ایجاد می‌کنیم. از یکی از آنها تابع a1 و از دیگری تابع a2 را فراخوانی می‌کنیم. با استفاده از اشاره‌گر this فقط می‌توانیم توضیح دهیم که یک نمونه تابع a1 را در اختیار دارد و دیگری تابع a2 را در اختیار دارد. اما کشف این مسئله که این نمونه‌ها از یک شیء یا چندین شیء هستند غیر ممکن است. توابع وراثتی شرایط را بدتر هم می‌کنند. زیرا آنها در کلاس‌های مشتق شده، کپی نمی‌شوند و ابهام زیاد می‌شود.

فرض کنید توابع a1 ,a2 به یک نمونه متصل هستند. و a1 , a2 و تابع دیگری که آن را a3 می‌نامیم به نمونه دیگری متصل هستند. در این حالت دونمونه می‌توانند از یک کلاس باشند که درنمونه اول تابع a3 فراخوانی نشده است یا این که نمونه دوم مشتق شده از کلاس اول باشد. در هر دو حالت کد ایجاد شده توسط کامپایلر یکسان می‌باشد.

ارتباط و وراثت کلاس‌ها بسته به هدف توابع متعلق به آنها می‌باشد و تنها با برداشت‌های ذهنی می‌توان کد اصلی را به‌صورت تقریبی به دست آورد.

به‌طور خلاصه هرگز بین شیء و نمونه شیء اشتباه نکنید و گیج نشوید. بدانید که شیء فقط در کد اصلی قرار دارد و در زمان کامپایل از بین می‌رود.

شناسایی توابع

همان‌طور که می‌دانید توابع بخش اصلی ساختار زبان‌های برنامه‌نویسی محسوب می‌شوند. بنابراین disassemble کردن یک کد با شناسایی توابع و پارامترهای آنها شروع می‌شود.

منظور ما از function روالی جدا از برنامه اصلی است که در بخش‌های مختلف برنامه قابل فراخوانی باشد. یک تابع می‌تواند یک یا چند آرگومان بگیرد. می‌تواند آنها را باز پس بفرستد. می‌تواند حاصل عملیات اجرایی خود را بازگرداند ویا می‌تواند هیچ مقداری را باز نگرداند. این نکات هیچ کدام مهم نیستند. نکته مهم و کلیدی برگرداندن کنترل به محلی است که تابع را فراخوانی کرده است. چگونه یک تابع می‌داند که کنترل را به کجا باید بازگرداند؟ آشکار است که کدی که این تابع را فراخوانی می‌کند باید آدرس بازگشت را ذخیره کند و آن را همراه آرگومان‌ها به تابع بفرستد. راه‌های بسیاری زیادی برای حل این مشکل وجود دارد. برای مثال با یک دستور پرش غیرشرطی در انتهای تابع به آدرس پس از فراخوانی آن، می‌توان این کار را انجام داد.

همچنین می‌توانیم آدرس بازگشت را در یک متغیر خاص ذخیره کنیم و بعد از این که تابع عملیات خود را به پایان رسانید با یک پرش غیرمستقیم با استفاده از این متغیر به‌عنوان عملگر دستور JUMP به آن آدرس باز گردیم. بدون اینکه بخواهیم مزایا و معایب این روش‌ها را بیان کنیم، این نکته را متذکر می‌شویم که بیشتر کامپایلرها با استفاده از دستورات ماشین Call و Ret عملیات فراخوانی و بازگشت از توابع را انجام می‌دهند.

دستور CALL آدرس دستور بعدی را در بالای پشته قرار می‌دهد و دستور RET آن را از بالای پشته بر می‌دارد و روند اجرایی را به آنجا منتقل می‌کند. آدرسی که دستور CALL به آن اشاره می‌کند آدرس شروع تابع می‌باشد و دستور RET تابع را پایان می‌دهد. اما باید بدانید که همیشه دستور RET پایان تابع نیست. با این وجود توابع را به دو روش می‌توانیم شناسایی کنیم. یکی با استفاده از ارجاعات که توسط دستور ماشین CALL صورت می‌پذیرد یا با epilog آنها که با دستور RET پایان می‌پذیرد.

روش ارجاعات و epilog به ما کمک می‌کنند که آدرس‌ها شروع و پایان توابع را شناسایی کنیم. در اینجا متذکر می‌شویم که شروع بسیاری از توابع با یک ترتیب خاص از دستورالعمل‌ها شروع می‌شود که به آن Prolog می‌گویند. این حالت برای شناسایی توابع بسیار مناسب است.

شناسایی فراخوانی‌ها

کد disassemble شده را جستجو کرده و تمام دستورات CALL را پیدا می‌کنیم. عملگر مورد نیاز این دستور آدرس شروع تابع می‌باشد. آدرس توابع معمولی که با نام خود فراخوانی می‌شوند در طول عملیات کامپایل محاسبه می‌شوند. عملگر دستور CALL در چنین حالاتی یک مقدار بلافصل می‌باشد.

حال به یک مثال می‌پردازیم.

فراخوانی مستقیم تابع :

```
func();
main()
{
	int a;
	func();
	a=0x666;
	func();
}

func()
{
	int a;
	a++;
}
```

نتیجه کامپایل تقریباً به صورت زیر است.

```
.text:00401000 push ebp
.text:00401001 mov ebp, esp
.text:00401003 push ecx
.text:00401004 call 401019
```

در این قسمت به دستور CALL رسیدیم. در اینجا عملگر بلافصل این دستور، آدرس شروع تابع می‌باشد. یا به عبارت دقیق‌تر offset بخش کد آن می‌باشد. حال در این قسمت به خط text:00401019. می‌رویم و تابع را نام‌گذاری می‌کنیم یعنی عملگر دستور CALL را با offset نام تابع جایگزین می‌کنیم.

```
.text:00401009 mov dword ptr [ebp-4], 666h
.text:00401010 call 401019
```

در این بخش نیز یک فراخوانی دیگر از تابع داریم. با مشاهده خط text:401019. در می‌یابیم که این ترکیب از دستورات قبلاً به‌صورت تعریف تابع بیان شده است و کار ما در این قسمت این است است که call 401019 را با call offset Function_name جایگزین کنیم.

```
.text:00401015 mov esp, ebp
.text:00401017 pop ebp
.text:00401018 retn
```

در این قسمت با یک دستور برای بازگشت از تابع مواجه شدیم. به هر حال این قسمت حتماً پایان تابع نیست و یک تابع می‌تواند محل‌های خروج متفاوتی داشته باشد. بعد از این خط، شروع دستورات تابع خود را می‌بینیم. این شناسایی توسط عملگر دستور CALL صورت گرفته است.

```
.text:00401019 push ebp
```

عملگر چندین دستور CALL به این آدرس ارجاع کرده‌اند. این قسمت آدرس شروع تابع می‌باشد. هر تابع باید دارای نام خاص خود باشد. این تابع را my function می‌نامیم.

```
.text:0040101A mov ebp, esp
.text:0040101C push ecx
.text:0040101D mov eax, [ebp-4]
.text:00401020 add eax, 1
.text:00401023 mov [ebp-4], eax
.text:00401026 mov esp, ebp
.text:00401028 pop ebp
.text:00401029 retn
```

این قسمت بدنه تابع می‌باشد. و خط آخر هم نشان‌دهنده انتهای تابع است.

همان‌طور که مشاهده کردید همه چیز بسیار ساده می‌باشد. البته اگر برنامه نویس از فراخوانی غیرمستقیم تابع استفاده کند کار کمی‌پیچیده خواهد شد. در این صورت آدرس‌ها از طریق ثبات فرستاده می‌شود و عملیات محاسبه آدرس به صوت پویا در زمان اجرای برنامه صورت می‌گیرد. در هر حالتی کامپایلر باید به‌صورتی آدرس تابع را ذخیره کند و ما باید آن را پیدا کرده و محاسبه کنیم.

حال به مثال زیر توجه کنید.

فراخوانی تابع با استفاده از اشاره گر:

```
func();
main()
{
        int (a*) ();
```

```
        a=func;
        a();
}
```

به‌صورت کلی نتیجه کامپایل باید به‌صورت زیر باشد:

کد disassemble شده برای فراخوانی یک تابع با استفاده از اشاره گر :

```
.text:00401000      push      ebp
.text:00401001      mov       ebp, esp
.text:00401003      push      ecx
.text:00401004      mov       dword ptr [ebp-4], 401012
.text:0040100B      call      dword ptr [ebp-4]
```

در این قسمت یک دستور CALL قرار دارد که فراخوانی غیرمستقیم تابع را پیاده سازی کرده است و آدرس تابع در خانه [EBP-4] قرار دارد.

در دو خط بالا دستور mov dword ptr [ebp-4], 401012 را می‌بینید. کنترل برنامه به خط text:401012. فرستاده می‌شود که دقیقاً آدرس شروع تابع می‌باشد. حال این خط را با این دستور جدید جایگزین می‌کنیم.

```
.  mov dword ptr [ebp-4], offset Function_name
```

```
.text:0040100E       mov      esp, ebp
.text:00401010       pop      ebp
.text:00401011       retn
```

سخت‌ترین حالت زمانی است که تابع به‌صورت دستی و با استفاده از دستور jmp فراخوانی می‌شود و مقدار آدرس بازگشت در stack قرار داده می‌شود طرح کلی این روش به این ترتیب است:

```
.PUSH ret_addrr/JMP func_addr
```

Ret_addr آدرس بازگشت مستقیم یا غیرمستقیم است و func_addr آدرس شروع مستقیم یا غیرمستقیم تابع است.

توجه داشته باید که این دو دستور همیشه پشت سر هم نمی‌آیند و دستورات دیگری ممکن است آنها را از یکدیگر جدا کنند.

حال ممکن است این سوال مطرح شود که دستور CALL چه عیبی داشت که ما باید از دستور JMP استفاده کنیم. با استفاده از دستور CALL همیشه پس از پایان یافتن تابع، کنترل به خط بعد از فراخوانی تابع در تابع پدر باز می‌گردد اما در حالتی ممکن است بخواهیم که بعد از اجرای تابع

کنترل اجرای برنامه به محل متفاوتی رفته و از آنجا ادامه یابد، در چنین حالتی باید به‌صورت دستی آدرس بازگشت مورد نیاز را مشخص کنیم و در تابع فرزند از دستور JMP استفاده کنیم.

شناسایی چنین توابعی کار بسیار مشکلی است. جستجوی رشته هیچ نتیجه خاصی نمی‌دهد چون برنامه از دستورات JMP برای پرش‌های کوتاه پر است. اگر نتوانیم تابع را شناسایی کنیم دو چیز را از دست خواهیم داد. ابتدا تابع فراخوانی کننده و دیگری تابعی که کنترل برنامه پس از بازگشت به آن داده می‌شود. متاسفانه برای این مشکل هیچ راه حل ساده و راحتی وجود ندارد. تنها روش این است که دستورات ما قبل JMP را مورد بررسی قرار دهیم.

به مثال زیر توجه کنید:

فراخوانی دستی تابع با استفاده از دستور JMP

```
funct();

main()
{
	__asm
	{
		LEA ESI, return_addr
		PUSH ESI
		JMP funct
	return_addr:
	}
}
```

کد نهایی کامپایل باید به‌صورت زیر باشد:

کد disassemble شده برای فراخوانی دستی تابع با استفاده از دستور JMP

```
.text:00401000 push  ebp
.text:00401001 mov   ebp, esp
.text:00401003 push  ebx
.text:00401004 push  esi
.text:00401005 push  edi
.text:00401006 lea   esi, [401012h]
.text:0040100C push  esi
.text:0040100D jmp 401017
```

این خط نمی‌تواند یک پرش معمولی باشد. بلکه یک فراخوانی تابع می‌باشد. چگونه ما این موضوع را در می‌یابیم؟

برای این کار به آدرس 0x401017 رفته و می‌بینیم.

```
.text:00401017   push   ebp
.text:00401018   mov    ebp, esp
.text:0040101A   pop    ebp
.text:0040101B   retn
```

شما چه فکر می‌کنید؟ این دستور ret کنترل اجرای برنامه را به کجا خواهد فرستاد؟ به‌طور طبیعی به آدرسی که در بالای stack قرار دارد می‌فرستد در آن جا چه چیز قرار دارد؟ مقدار قرار داده شده در stack توسط دستور push ebp ، توسط دستور pop ebp برداشته می شود. حال به همان جایی که دستور JMP قرار داشت باز می‌گردیم و فراخوانی‌های stack را بررسی می‌کنیم در خط 401000C دستور push ESI را می‌بینیم. در این جا مقدار ثبات ESI در بالای پشته قرار داده شده است و مقدار آن 0x401012 می‌باشد. این مقدار آدرس شروع تابعی است که توسط دستور JMP فراخوانی شده است.

```
.text:00401012 pop   edi
.text:00401013 pop   esi
.text:00401014 pop   ebx
.text:00401015 pop   ebp
.text:00401016 retn
```

شناسایی خودکار توابع با استفاده از IDA Pro

IDA Pro توانایی تحلیل عملگرهای دستور CALL را دارد و می‌تواند برنامه را به توابع جدا از هم تقسیم کند. البته هنوز نمی‌تواند فراخوانی‌های پیچیده و دستی را که با استفاده از دستور JMP صورت می‌گیرد، مدیریت کند.

باید بدانید که حالت‌های پیچیده، بسیار کم رخ می‌دهند. تقریباً آمار آنها در حدود یک درصد از توابعی هستند که IDA می‌تواند تشخیص دهد.

Prolog : اکثر کامپایلرهایی که کد را بهینه‌سازی نمی‌کنند کدهای زیر را در شروع تابع خود قرار می‌دهند که Prolog نامیده می‌شود.

```
push        ebp
mov         ebp, esp
sub         esp, xx
```

حاصل کار پرولوگ به‌صورت زیر است:

اگر از ثبات EBP برای آدرس‌دهی متغیرهای محلی استفاده شده است ابتدا مقدار آن باید در پشته ذخیره شود. حال مقدار ثبات اشاره‌گر به پشته یعنی ESP ، در EBP کپی می‌شود. پشته باز است و مقدار ESP به اندازه بلوک حافظه اختصاص داده شده به متغیرهای محلی کاهش می‌یابد.

از این ترتیب دستورات: `PUSH EBP/MOV EBP, ESP/SUB ESP, xx` برای شناسایی توابع در یک فایل استفاده می‌شود و IDA از این تکنیک استفاده می‌کند، اما کامپایلرهایی که از بهینه‌سازی کد استفاده می‌کنند می‌دانند که چگونه از خود ESP به‌منظور آدرس‌دهی به متغیرهای محلی استفاده کنند واز ثبات EBP همانند دیگر ثبات‌های همه منظوره استفاده می‌کنند. پرولوگ توابع بهینه‌سازی شده فقط از دستور Sub ESP , XXX تشکیل شده است که متاسفانه این یک خط دستور برای شناسایی توابع بسیار کم می‌باشد.

Epilog : بعد از اتمام کار، هر تابع پشته را می‌بندد. و اشاره‌گر آن را به پایین می‌برد. و مقدار قبلی EBP را جایگزین می‌کند.

Epilog با دو روش عمل می‌کند: یکی با افزایش ESP توسط دستور ADD به مقدار کافی ویا این که مقدار EBP که به پایین پشته اشاره می‌کرد، در esp کپی می‌شود.

Epilog های موجود :

```
Epilog 1
pop         ebp
add         esp, 64h
retn


Epilog 2
mov         esp, ebp
pop         ebp
retn
```

متذکر می‌شویم که دستورات POP EBP / ADD ESP, xxx و MOV ESP, EBP / POP EBP نیازی نیست که به‌صورت پشت سر هم باشند و توسط دستورات دیگری می‌توانند از هم جدا شوند. بنابراین جستجوی رشته نامناسب می باشد و باید از جستوی mask استفاده کرد..

در کامپایلرهایی که از تبدیلات پاسکال استفاده می‌کنند تابع باید پشته را از آرگومان‌های خود خالی سازد و در اکثر مواقع این کار توسط دستور RETn انجام می‌شود. n برابر است با تعداد بایت‌هایی که هنگام بازگشت باید از پشته برداشته شود. اما کامپایلرهایی که از تبدیلات C استفاده می‌کنند تخلیه پشته را بر عهده کد فراخوانی کننده گذاشته و با دستور RET متوقف می‌شوند.

توابع API در windows از ترکیبی از تبدیلات C و پاسکال استفاده می‌کنند. آرگومان‌های توابع در پشته از راست به چپ قرار می‌گیرند. اما پاک‌سازی پشته توسط خود تابع انجام می‌پذیرد در بخش آرگومان‌های توابع این مبحث را به‌طور کامل تشریح خواهیم کرد.

پس RET برای نمایش epilog تابع کافی می‌باشد. اما هر epilog به معنای پایان تابع نیست. اگر یک تابع دارای چندین دستورRET باشد، کامپایلر برای هر کدام از آنها یک epilog ایجاد می‌کند. همیشه چک کنید که آیا بعد از این epilog ، epilog جدید دیگری هم وجود دارد یا خیر؟

همیشه بدانید کامپایلرها کدی که هیچ گاه کنترل برنامه را در دست نمی‌گیرد، در فایل اجرایی قرار نمی‌دهند. به بیان دیگر هر تابع فقط دارای یک epilog می‌باشد و هر چیزی که بعد ازاولین RET، به عنوان کد غیر ضروری باشد دور ریخته می‌شود.

دور ریختن کد بعد از عملگر غیر شرطی

```
int func(int a)          push        ebp
{                        mov         ebp, esp
                         mov         eax, [ebp+arg_0]
       return a++;       mov         ecx, [ebp+arg_0]
       a=1/a;            add         ecx, 1
       return a;         mov         [ebp+arg_0], ecx
                         pop         ebp
}                        retn
```

تابعی با چندین epilog

```
int func(int a)
{
       if (!a) return a++;
       return 1/a;
}
```

کد disassemble شده برای تابع کامپایل شده با چندین epilog

```
       push       ebp
       mov        ebp, esp
       cmp        [ebp+arg_0], 0
       jnz     short loc_0_401017
       mov        eax, [ebp+arg_0]
       mov        ecx, [ebp+arg_0]
       add        ecx, 1
       mov        [ebp+arg_0], ecx
       pop        ebp
       retn
```

این قسمت به‌صورت آشکار epilog تابع می‌باشد اما پس از آن، تابع بدون prolog جدید دنبال شده است.

```
loc_0_401017:
```

این ارجاعات، کد ما را به یک پرش منتهی می‌کند و این نشان می‌دهد که این کد، ادامه تابع قبل بوده و شروع یک تابع جدید نمی‌باشد. توابع نرمال با دستور CALL فراخوانی می‌شوند نه JMP. حال ببینیم که این یک تابع نرمال است یا نه؟ برای این کار باید ببینیم که آیا مقدار بازگشتی در بالای پشته قرار دارد یا خیر ؟ در اینجا فرض ما درباره ادامه کد درست بود.

```
mov         eax, 1
cdq
Idiv        [ebp+arg_0]

loc_0_401020:         ; CODE XREF: sub_0_401000+15↑j
pop         ebp
retn
```

نکته: *وقتی که پردازشگر 80286 آمد دو دستور LEAVE , ENTER در مجموعه دستورات پدید آمدند. آنها برای باز کردن و بستن قاب پشته ایجاد شده بودند اما امروزه در هیچ کدام از کامپایلرها از آنها استفاده نمی‌شود. زیرا این دو دستور بسیار کند عمل می‌کردند، در یک پنتیوم دستور ENTER در ۱۰ سیکل زمانی انجام می‌شد که به‌صورت معمول در ۵ سیکل زمانی انجام می‌شود. به همین صورت LEAVE در ۵ سیکل زمانی انجام می‌شد ولی درحالت معمولی و با دستورات معمولی در دوسیکل زمانی انجام می‌شود.*

به جای ENTER از PUSH EBP / MOV EBP, ESP / SUB ESB, XXX استفاده می‌کنیم و به جای LEAVE از MOV ESP , EBP / POP EBP استفاده می‌کنیم.

توابع Naked : ویژوال C++ نوعی از توابع با خاصیت naked را پشتیبانی می‌کند این روش به برنامه‌نویس اجازه می‌دهد که تابعی بدون epilog و Prolog ایجاد کند. یعنی کامپایلر در انتهای تابع دستور RET قرار نمی‌دهد. و شما باید این کار را به‌صورت دستی با اضافه کردن دستورات اسمبلی انجام دهید.

```
. __asm{ret}
```

استفاده از Return به تنهایی نتیجه دلخواه را ایجاد نمی‌کند. به‌طورکلی و عمومی پشتیبانی از توابع به‌صورت Naked برای نوشتن درایور در C طراحی شده است و در میان طراحان و برنامه‌نویسان مکانیزم‌های امنیتی موضوعی غیر منتظره بود. اما واقعاً این قابلیت خوبی است که امکان ایجاد تابعی را داشته باشیم، بدون نگرانی این که کامپایلر با چه روش غیرقابل پیش‌گویی آن را انجام خواهد داد. این بدان معنی است که در برنامه‌ها امکان وجود توابعی بدون epilog و Prolog وجود دارد.

شناسایی آرگومان‌های تابع

شناسایی آرگومان‌های تابع یک بخش کلیدی تحلیل برنامه‌های disassemble شده است.این بحث کمی‌طولانی و خسته‌کننده است ولی برای یادگیری هر مطلبی باید هزینه آنرا نیز پرداخت کنید.

سه روش برای فرستادن آرگومان‌ها به یک تابع وجود دارد :

۱- از طریق پشته

۲- از طریق ثبات‌ها

۳- از طریق پشته و ثبات‌ها

آرگومان‌ها یا با مقدار منتقل می‌شوند یا با ارجاع. در حالت انتقال با مقدار یک کپی از متغیر به تابع فرستاده می‌شود اما در حالت انتقال با ارجاع یک اشاره گر به آن فرستاده می‌شود.

قراردادهای فرستادن آرگومان‌ها

برای این که کار به خوبی انجام شود تابع فراخوانی کننده علاوه بر پیش تعریف تابع فراخوانی شده باید با روشی که آرگومان‌ها به آن فرستاده می‌شوند نیز هماهنگی داشته باشد. اگر آرگومان‌ها از طریق ثبات‌ها فرستاده می‌شوند باید نشان دهد که چه آرگومانی در چه ثباتی قرار دارد. اگر آرگومان‌ها از طریق پشته فرستاده می‌شوند باید ترتیبی تعریف کند که آرگومان‌ها در کجا و چگونه قرار گرفته‌اند. همچنین باید معین کنید که بعد از این که تابع فراخوانی شده اجرا شده چه کسی مسئول خالی کردن پشته از آرگومان‌ها است.

ابهامات مکانیزم فرستادن آرگومان‌ها یکی از دلایل نارسازگاری بین کامپایلرهای مختلف است. حال به این فکر می‌افتیم که چرا سازندگان کامپایلرها را مجبور به پیروی از یک متد خاص نمی‌کنند. چون متاسفانه این روش بیش ازآن که مشکلاتی را حل کند مشکلات جدیدی را به وجود می‌آورد.

چون طراحان کامپایلرها به یک نظر مشترک و واحد نرسیدند تصمیم گرفتند که از همه روش‌های ممکن برای فرستادن آرگومان‌ها استفاده کنند. و این کار را با پذیرفتن تعدادی قرارداد بین خود انجام دادند. حال این قراردادها را بررسی می‌کنیم:

قرارداد C : با گذاشتن cdecl__ آرگومان‌ها از راست به چپ در پشته به ترتیبی که تعریف شده‌اند قرارمی‌گیرند. در اینجا تخلیه پشته بر عهده تابع فرزند است.

هنگامی‌که کامپایلر تابعی را می‌بیند که از قرارداد C استفاده کرده است، «_» به عنوان پیشوند به نامش اضافه می‌کند و اشاره‌گر this نیز در آخر از طریق پشته فرستاده می‌شود.

قرارداد pascal : با گذاشتن PASCLA آرگومان‌ها از چپ به راست به ترتیبی که تعریف شده‌اند قرار می‌گیرند. در این قسمت تخلیه پشته برعهده تابع پدر است.

قرارداد استاندارد: با گذاشتن stdcall__ آرگومان‌ها از راست به چپ در پشته قرارمی‌گیرند اما تخلیه پشته توسط تابع پدر صورت می‌گیرد. نام توابعی که از این قرارداد استفاده می‌کنند، با پیشوند «_» شروع شده و با پسوند «@» خاتمه می‌یابد و پس از آن تعداد بایت‌هایی که به تابع منتقل شده است آورده می‌شود اشاره‌گر this در نهایت از طریق پشته منتقل می‌شود.

قرارداد fastcall : این قراداد بیان می‌کند که آرگومان‌ها از طریق ثبات‌ها فرستاده خواهند شد. کامپایلرهای Borland و مایکروسافت کلمه کلیدی fastcall__ را پشتیبانی می‌کنند اما به طرق مختلفی آن را تفسیر می‌کنند. توابعی که به قرارداد fastcall پیوسته‌اند یعنی قبل از نام آنها کلمه کلیدی fastcall__ قرار داده شده است در زمان کامپایل قبل از نام آنها کاراکتر «@» توسط کامپایلر قرار می‌گیرد.

قرارداد پیش‌فرض: هیچ توضیح روشن و صریحی از نوع فراخوانی در این روش وجود ندارد و کامپایلر بنا به صلاح دید خود یکی از روش‌ها را انتخاب می‌کند. اکثر کامپایلرها اشاره‌گر this را از طریق ثبات منتقل می‌کنند که در مایکروسافت ثبات ECX و برلند EAX می‌باشد.

دیگر آرگومان‌ها نیز می‌توانند از طریق ثبات انتقال یابند اما این صلاح دید از جانب قسمت بهینه سازی (Optimizer) باید صورت بگیرد. مکانیزم انتقال و منطق آن در هر کامپایلر نسبت به کامپایلر دیگر فرق می‌کند. این مسئله بسیار ناخوشایند است اما ما ناچار به بررسی آن هستیم.

اهداف و وظایف

وقتی که یک تابع را تحلیل و بررسی می‌کنیم، یک تحلیل گر کد با عملیات زیر مواجهه می‌شود:

او باید بداند که از چه قراردادی برای فراخوانی استفاده شده است؟ تعداد آرگومان‌هایی که به تابع فرستاده می‌شود چند تا است؟ کار و نوع هر آرگومان چیست؟ حال می‌خواهیم با مطرح شدن سوالات فوق به شرح آنها بپردازیم.

شناسایی نوع قرارداد از طریق روش تخلیه و پاک سازی پشته به سختی صورت می‌گیرد. اگر پشته توسط تابع فراخوانی شده خالی گردد ما با قرارداد cdecl سر و کار داریم.در غیر این صورت با PASCAL یا StdCall سر و کار خواهیم داشت. علت این عدم قطعیت این است که پیش تعریف تابع برای ما ناشناخته بوده و ترتیب قرار گرفتن آرگومان‌ها در پشته قابل شناسایی نیست. اما گر نوع کامپایلر شناخته شده باشد، و برنامه‌نویس از روش پیش‌فرض فراخوانی استفاده کرده باشد، احتمالاً می‌توانیم نوع فراخوانی توابع را مشخص کنیم. در برنامه‌های ویندوز دو روش StdCall و PASCAL به‌طور گسترده‌ای مورد استفاده قرار می‌گیرد اما با این وجود همچنان عدم قطعیت ادامه دارد. برای شناسایی روش فراخوانی تابع، وقتی هر دو تابع پدر و فرزند را در اختیار داشته باشیم، می‌توانیم با یک تناظر بین آرگومان‌های فرستاده شده و آرگومان‌های دریافتی شناخت خوبی نسبت به روش فراخوانی پیدا کنیم. مورد دیگر توابع کتابخانه ای هستندکه پیش تعریف آنها برای ما شناخته شده است. در این موارد اگر بدانیم که آرگومان‌ها به چه ترتیب در پشته قرار می‌گیرند به راحتی می‌توانیم نوع و عملکرد هر یک را به سادگی شناسایی کنیم. این اطلاعات را از پیش تعریف تابع به دست می‌آوریم.

شناخت تعداد آرگومان‌ها و روش ارسال آنها

همان‌طور که در بالا اشاره کردیم آرگومان‌ها می‌توانند از طریق پشته، ثبات و یا ترکیبی از این دو فرستاده شوند. آرگومان‌های مجازی نیز می‌توانند از طریق متغیرهای سراسری انتقال یابند. اگر از پشته تنها به‌منظور ارسال آرگومان‌ها استفاده می‌شد، شمارش آنها کار آسانی بود ولی از پشته به‌منظور نگه‌داری موقت داده‌های ثبات‌ها نیز استفاده می‌شود. همین امر کار ما را دشوار می‌کند. پس اگر با دستور PUSH مواجهه شدید تصور نکنید که یک آرگومان راشناسایی کرده‌اید. شناسایی تعداد بایت‌هایی که به‌عنوان آرگومان به تابع فرستاده می‌شوند غیرممکن است اما به‌سادگی می‌توان تعداد بایت‌هایی که پس از تکمیل عملیات تابع از پشته POP می‌شوند را شناسایی کرد.

اگر تابع از قرارداد PASCAL استاندارد پیروی کند پشته با دستور RETn خالی خواهد شد. همان‌طور که ذکر شد مقدار n تعداد بایت‌هایی است که باید از پشته برداشته شود. در مورد استاندارد cdecl بررسی‌ها به این سادگی نیست. به‌طورکلی در این استاندارد بعد از فراخوانی، دستور ADD ESP , n قرار دارد.

می‌توانیم فرض کنیم تعداد بایت‌هایی که در پشته قرار می‌گیرند برابر تعداد بایت‌هایی هستند که از پشته برداشته می‌شوند در غیر این صورت بعد از این که تابع اجرا شد توازن پشته به هم خواهد خورد و برنامه متوقف می‌شود. برخی از کامپایلرهایی که بهینه‌سازی کد را انجام می‌دهند، در بعضی شرایط خاص اجازه عدم توازن پشته را می دهند.

باید بدانیم که تعداد آرگومان‌های فرستاده شده به تابع برابر تعداد بایت‌های فرستاده شده به آن نمی‌باشد برای مثال نوع double هشت بایت مصرف می‌کند. یک رشته کاراکتری با مقدار فرستاده شده و هر چقدر که فضا نیاز دارد مصرف می‌کند همچنین یک رشته، رکورد داده، آرایه، یا شیء می‌توانند وارد پشته شوند و به جای دستور PUSH از دستور MOVS استفاده کنند. (استفاده از دستور MOVS نشان دهنده فرستادن آرگومان‌ها با مقدار است).

در این قسمت مطالب گفته شده را منظم کرده و مروری برروی آنها خواهیم داشت. با تحلیل کد تابع فراخوانی کننده (پدر) یک تابع، غیرممکن است که بتوانیم تعداد آرگومان‌های فرستاده شده به تابع از طریق پشته را تشخیص دهیم. در برخی موارد حتی تعداد بایت‌های فرستاده شده را نیز نمی‌توان به دقت مشخص کرد و نوع انتقال نیز مبهم خواهد بود. حال به بررسی یک مثال کوچک می‌پردازیم.

```
PUSH         0x404040
CALL MyFuct :0x404040
```

یک آرگومان که با مقدار فرستاده شده (مقدار ثابت 0x404040)، یا یک اشاره‌گر به شیء و یا رکوردی که در آفست (Offset) 0x404040 قرار داده شده است (فرستادن با ارجاع) همان‌طور که در این مثال ساده مشاهده کردید بدون تجزیه و تحلیل تابع فراخوانی شده (فرزند) نمی‌توان هیچ نظر قطعی در این زمینه بیان کرد و باید تابع فرزند نیز مورد بررسی قرار بگیرد. این که این تابع چگونه با آرگومان‌هایی که به آن فرستاده شده کار می‌کند، مشخص می‌کند که نوع، عملکرد و مقدار آنها چه بوده است؟

به‌منظور آشنایی بیشتر با آدرس‌دهی متغیرها در پشته به مثال زیر توجه کنید:

مکانیزم فرستادن آرگومان‌ها

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

struct XT{
       char s0[20];
       int x;
};

void MyFunc(double a, struct XT xt)
```

```
{
       printf("%f, %x, %s\n", a, xt.x, &xt.s0[0]);
}

main()
{
       struct XT xt;
       strcpy(&xt.s0[0], "Hello, World!");
       xt.x = 0x777;
MyFunc(6.66, xt);
}
```

کد disassemble شده این برنامه با کامپایلر ویژوال C++ با تنظیمات پیش‌فرض، به‌صورت زیر است.

کد disassemble شده برای فرستادن آرگومان‌ها با استفاده از ویژوال C++.

```
main           proc near           ; CODE XREF: start+AF↓p

var_18         = byte ptr -18h
var_4          = dword ptr -4
       push    ebp
       mov     ebp, esp
       sub     esp, 18h
```

اولین PUSH مربوط به پرولوگ تابع است و نه ارسال یک آرگومان

```
       push    esi
       push    edi
```

عدم مقداردهی صریح ثبات‌ها نشان می‌دهد که احتمالاً در پشته ذخیره شده‌اند و به عنوان آرگومان فرستاده نمی‌شوند. اگر برای ارسال آرگومان‌ها به این تابع، علاوه بر پشته از ثبات‌های EDI , ESI نیز استفاده شد، قرار دادن آنها در پشته ممکن است بیانگر این مطالب باشدکه آرگومان‌ها به تابع بعدی فرستاده می شوند.

```
       push    offset aHelloWorld ; "Hello, World!"
```

در این قسمت اشاره گر به یک رشته، به‌عنوان آرگومان فرستاده می‌شود. از نظر تئوری ممکن است که یک مقدار ثابت به طور موقت در پشته ذخیره شود و در مراحل بعد از پشته برداشته شده و در یک ثبات آماده ریخته شود. همچنین ممکن است که به‌طور مستقیم آدرس پشته آن داده شود. با این وجود می‌دانیم که هیچ کامپایلری توانایی استفاده از این روش‌ها را نداشته و قرار دادن یک مقدار ثابت در پشته همیشه نشان از فرستادن یک آرگومان می‌باشد.

```
lea     eax, [ebp+var_18]
```

اشاره‌گر به یک بافر محلی در EAX قرار می‌گیرد.

```
push    eax
```

EAX در پشته ذخیره می‌شود. همه آرگومان‌ها قابل شناسایی هستند. اولین آرگومان ، شناسایی شده و می‌توانیم مطمئن باشیم که از این به بعد هر چه در پشته وارد می‌شود یک آرگومان است.

```
call    strcpy
```

پیش تعریف تابع strcpy (char , char) به ما اجازه شناسایی ترتیب ارسال آرگومان‌ها را نمی‌دهد. ولی همه توابع کتابخانه ای C از قرارداد cdecl استفاده می‌کنند یعنی آرگومان‌ها از راست به چپ قرار می‌گیرند. در نتیجه کد شناسایی شده به این صورت می‌باشد:

```
Strcpy ( & buff [o], "Hello , World!")
       add    esp, 8
```

۸ بایت از پشته برداشته می‌شود. می‌توانیم نتیجه بگیریم که تنها دو آرگومان به پشته فرستاده شده است. در نتیجه PUSH ESI و PUSH EDI از آرگومان‌های تابع نبودند.

```
mov     [ebp+var_4], 777h
```

مقدار ثابت 0x777 در متغیر محلی قرار داده می‌شود. مطمئناً این یک ثابت است و نه یک اشاره‌گر زیرا در ویندوز داده‌های کاربر نمی‌تواند در این محدوده از حافظه قرار گیرد.

```
sub     esp, 18h
```

حافظه برای متغیرهای موقتی اختصاص داده می‌شود. متغیرهای محلی زمانی ایجاد می‌شوندکه آرگومان‌ها با مقدار به تابع فرستاده شوند.

```
mov     ecx, 6
```

مقدار ثابت 0x6 در Ecx قرار داده می‌شود اما هنوز علت این کار را نمی‌دانیم.

```
lea     esi, [ebp+var_18]
```

اشاره‌گر به بافرمحلی که حاوی یک کپی از رشته "Hello, world!" است، در ثبات ESI قرار داده می‌شود.

```
        mov     edi, esp
```

اشاره‌گر به داده ذخیره شده در بالای پشته، در ثبات EDI قرار می‌گیرد.

```
        repe movsd
```

این قسمت فرستادن یک رشته با مقدار است. کل رشته در پشته کپی شده و (٤×٦) بایت از فضای آن را اشغال می‌کند. (٦ اندازه ثبات شمارش گر ECX است و ٤ اندازه هر double word است). این آرگومان ۲۰ بایت از فضای پشته را اشغال می‌کند. از این مقدار به‌منظور شناسایی تعداد آرگومان‌ها استفاده می‌کنیم و انجام این کار بر اساس تعداد بایت‌هایی است که از پشته برداشته می‌شود. اطلاعات از آدرس [ebp + var_18] تا [ebp + var_18 + 0x14] در پشته کپی می‌شود. اما متغیر var_4 شامل مقدار ثابت 0x777 می‌باشد. پس این متغیر نیز همراه رشته به تابع فرستاده می‌شود. حال می‌توانیم ساختار رکورد را به‌صورت زیر بازسازی کنیم.

```
struct x{
      char s0[20]
      int x
}
```

```
        push    401AA3D7h
        push    0A3D70A4h
```

دو آرگومان دیگر در پشته قرار داده می‌شوند. این دو ممکن است یک آرگومان از نوع int 64 یا یک double باشند. با استفاده از این کدها تشخیص نوع ممکن نیست.

```
        call    MyFunc
```

myfunc فراخوانی می‌شود. متاسفانه نمی‌توانیم پیش تعریف تابع را کشف کنیم. تنها مسئله‌ای که روشن است این است که اولین آرگومان از چپ یا از راست یک رکورد است.

```
        add     esp, 20h
        pop     edi
        pop     esi
        mov     esp, ebp
        pop     ebp
        retn
sub_401022      endp
```

آدرس دهی آرگومان‌ها در پشته

مفهوم اساسی پشته در دو عمل خاصه می‌شود: یکی گذاشتن عناصر در پشته و دیگری برداشتن عناصر از پشته. ولی در صورتی که با عنصر سوم کار داریم نیازی نیست که دو عنصر اول را از پشته برداریم. یک پشته همانند یک آرایه می‌باشد با دانستن موقعیت اشاره‌گر پشته و اندازه عنصر می‌توانیم offset آن را محاسبه کرده و به سادگی مقدار متغیر موردنظر را به دست آوریم. پشته هم مانند آرایه مشکل نوع متغیر را دارد. به عنوان مثال در صورتی که بخواهیم یک بایت در آن قرار دهیم با توجه به محدودیت ٤ بایت قطعات پشته، مجبور خواهیم بود که آن را به اندازه یک متغیر double گسترش دهیم و اگر متغیری داریم که ۸ بایت فضا نیاز دارد، باید دو عنصر از پشته را به آن اختصاص دهیم. در کنار فرستادن آرگومان‌ها، آدرس بازگشت توابع را نیز می‌توان در پشته قرار داد. برای این کار یک یا دو عنصر از پشته را بسته به نوع فراخوانی (دور یا نزدیک) اختصاص می‌دهیم. فراخوانی نزدیک در محدوده یک segment صورت می‌گیرد و فقط offset دستور بعد از Call را ذخیره می‌کنیم. اگر تابع فراخواننده و فراخوانی شده در segment متفاوت واقع شده باشند، برای بازگشت باید علاوه بر offset، شماره section را نیز داشته باشیم. آدرس بازگشت بعد از آرگومان‌ها در پشته قرار می‌گیرد. با توجه به مدل حافظه Flat در ویندوز 9X و NT می‌توانیم موضوع section را فراموش کرده و همه جا از پرش‌های نزدیک استفاده کنیم. کامپایلرهایی که بهینه‌سازی کد را انجام نمی‌دهند، از یک ثبات خاص (معمولاً EBP) برای آدرس‌دهی آرگومان‌ها استفاده می‌کنند.

از آنجایی که پشته از آدرس بزرگتر به کوچکتر رشد می‌کند offset همه آرگومان‌ها مثبت است. Offset آرگومان n ام از این فرمول محاسبه می‌شود:

```
arg_offset = N*size_element + size_return_address
```

آرگومان شماره N از بالای پشته یعنی صفر شمارش می شود.

Size_element اندازه قطعات پشته را نشان می‌دهد که در ویندوزهای خانواده 9X و NT برابر ٤ بایت می‌باشد.

Size_return_ Address اندازه فضای اختصاص داده شده به‌منظور ذخیره آدرس بازگشت می‌باشد. این اندازه هم در ویندوزهای NT , 9x برابر ٤ بایت است.

معمولاً ما کاری برعکس این انجام می‌دهیم. یعنی با دانستن offset یک عنصر بتوانیم تعداد آرگومان‌هایی که آدرس‌دهی شده‌اند را به دست آوریم. فرمول زیراز فرمول قبلی به دست آمده است.

$$N = \frac{arg\_offset - size\_return\_address}{size\_element}.$$

به هرحال چون مقدار قبلی EBP قبل از آنکه مقدار ESP در آن کپی شود باید در همان پشته ذخیره شود، با اضافه کردن ظرفیت ثبات EBP به اندازه آدرس بازگشت این فرمول را تصحیح می‌کنیم.

از نقطه نظر تحلیل‌گران کد، یک نکته خوب در این روش آدرس‌دهی آرگومان‌ها وجود دارد. با دیدن دستوری مانند MOV EAX , [EBP + 0x10] می‌توانیم محاسبه کنیم که چه آرگومانی آدرس‌دهی شده است. کامپایلرهایی که بهینه‌سازی کد را انجام می‌دهند به‌منظور آزادسازی ثبات EBP، آدرس‌دهی را به‌طور مستقیم از طریق ESP انجام می‌دهند. مقدار ESP در طول اجرای تابع تغییر می‌کند. هر زمان که متغیری وارد پشته شده یا از آن خارج می‌شود مقدار آن تغییر می‌کند. یعنی offset آرگومان‌ها نسبت به ESP ثابت نمی‌ماند. برای این که بدانیم چه آرگومانی آدرس‌دهی شده است، باید در آن نقطه از برنامه مقدار ESP را بدانیم. برای دانستن این موضوع باید همه تغییرات را از ابتدای برنامه دنبال کنیم. به مثال زیر توجه کنید:

کد Disassemble شده یک تابع در حال دریافت آرگومان‌ها

```
MyFunc          proc        near ; CODE XREF: main+39↑p

arg_0           = dword ptr 8
arg_4           = dword ptr 0Ch
arg_8           = byte ptr 10h
arg_1C          = dword ptr 24h
```

IDA چهار آرگومان ارسالی به تابع را شناسایی کرده است. البته اگر آرگومانی برای مثال int 64 باشد IDA آن را به عنوان چند کلمه مختلف در نظر می‌گیرد. به همین دلیل همیشه نمی‌توان به IDA اعتماد کرد. کد ایجاد شده توسط IDA باید تفسیر و بررسی شود. هیچ چیزی نمی تواند جلوی دسترسی تابع فراخوانی شده را به پشته پدر بگیرد. یعنی ممکن است هیچ آرگومانی به تابع فرستاده نشده باشد و آن تابع مقداری را از پشته بردارد و یا آن را دستکاری کند. این کار اساساً باعث خطاهای برنامه‌نویسی می‌شود. البته ما گاهی به این امکانات و کارها نیاز داریم. عددی که بعد از arg قرار گرفته offset نسبتی آن از شروع قاب پشته می‌باشد. قاب (frame) پشته ۸ بایت نسبت به EBP جا به جا می‌شود. ٤ بایت آدرس بازگشت را نگه می‌دارد و ٤ بایت اضافه برای نگه‌داری مقدار قبلی ثبات EBP استفاده می‌شود.

```
push    ebp
mov     ebp, esp
```

```
lea      eax, [ebp+arg_8]
```

اشاره‌گر به یک آرگومان بدست آمده، حال باید کشف کنیم که اشاره گر مربوط به کدام آرگومان است. IDA محاسبه کرده است که این آرگومان در آفست ۸ نسبت به آدرس شروع قاب پشته قرار داده شده است.حال به سراغ تابع فراخواننده می‌رویم تا ببینیم که چه مقداری را وارد کرده‌ایم. آخرین آیتم‌ها ۲ آرگومان نامشخص می‌باشند. قبل از آن یک رشته و یک متغیر از نوع int در پشته قرار داده شده است. پس EBP + ARG_8 به یک رشته اشاره می‌کند.

```
push     eax
```

اشاره‌گر به دست آمده در پشته قرار می‌گیرد. احتمالاً این اشاره‌گر به‌عنوان آرگومان به تابع بعدی فرستاده خواهد شد.

```
mov      ecx, [ebp+arg_1C]
```

محتوای آرگومان EBP + ARG_1 C در ECX قرار داده می‌شود. یادآور می‌شویم که نوع int در رکورد، درoffset 0x14 از شروع قرار دارد و ARG_8 شروع آن می‌باشد. به این ترتیب است آفست آن0x1c = 0x8 + 0x14 می‌باشد.

```
push     ecx
```

متغیر به دست آمده در پشته قرار داده می‌شود.

```
mov      edx, [ebp+arg_4]
```

حال یکی از دو متغیر نا مشخص را که در پشته وارد شده اند، به دست آوردیم.

```
push     edx
```

آن را دوباره در پشته وارد می‌کنیم تا آرگومان را به تابع بعدی بفرستیم.

```
mov      eax, [ebp+arg_0]
push     eax
```

دومین آرگومان نامشخص نیز وارد پشته می‌شود.

```
push     offset aFXS ; "%f,%x,%s\n"
call     _printf
```

در این قسمت فراخوانی تابع printf را داریم. فرمت رشته به آن فرستاده می‌شود. همان‌طور که می‌دانید تابع printf تعداد آرگومان‌های متغیری دارد. تعداد و نوع آنها توسط یک رشته کنترلی مشخص می‌شود. بیاد بیاورید که ابتدا اشاره گر به این رشته کنترلی را در پشته قرار دادیم. اولین مشخص کننده از سمت راست" %s "می‌باشد که فرمت خروجی رشته را معین می‌کند. سپس یک متغیر از نوع int در پشته قرار داده شده است. دومین مشخص کننده" %x " می‌باشد که نمایش یک عدد integer به‌صورت hex می‌باشد. سومین مشخص کننده"%f"می‌باشد که مربوط به محل قرار گرفتن دو آرگومان در پشته می‌باشد. اگر به راهنمای ویژوال C++ مراجعه کنیم خواهیم دید که مشخص کننده "%f" برای خروجی نوع Floating–point می‌باشد. این مقدار بسته به نوع خود یعنی Float یا double می‌تواند ٤ یا ٨ بایت فضا اشغال کند. در اینجا به روشنی ٨ بایت فضا را اشغال می‌کند. در این قسمت تقریباً توانسته ایم که پیش تعریف تابع را بازسازی کنیم این بازسازی را مشاهده می‌کنید:

```
Cdecl myfunc (double a , struct b)
```

نوع فراخوانی cdecl می‌باشد. یعنی پشته توسط تابع فراخواننده خالی خواهد شد. اما ترتیب اصلی متغیرها را نتوانستیم کشف کنیم. به هرحال ترتیب اصلی آرگومان‌ها در پیش تعریف تابع نقش مهمی را ایفا نمی‌کنند. بلکه مسئله اصلی برقراری ارتباط بین آرگومان‌های فرستاده شده و دریافت شده می‌باشد. این کار با بررسی تابع فراخواننده و فراخوانی شده صورت می‌گیرد. تحلیل و بررسی یکی از این دو ما را به نتیجه نخواهد رساند.

```
add         esp, 14h
pop         ebp
retn
MyFunc      endp
```

در اینجا کمی پیش رفت کردیم و توانستیم پیش تعریف اولین تابع را با موفقیت بازسازی کنیم.

آرگومان‌های پیش فرض :Default Arguments

به‌منظور ایجاد سهولت در فراخوانی توابعی که آرگومان‌های زیادی دارند، زبان C++ امکان تعریف و استفاده از مقادیرپیش فرض را فراهم آورده است.

در این قسمت دو سوال مطرح می‌شود:

۱- آیا فراخوانی تابع با مقادیر پیش‌فرض فرقی با فراخوانی توابع معمولی دارد؟

۲- کدام تابع (فراخوانی شده یا فراخواننده) آنها را از بین می‌برند.

وقتی تابعی با مقادیر پیش‌فرض فراخوانی می‌شود کامپایلر آرگومان‌های لازم را همان‌طور که صلاح می‌داند اضافه می‌کند پس فراخوانی توابع با مقدار پیش فرض فرقی با فراخوانی دیگر توابع ندارد.

این بحث را با یک مثال اثبات می‌کنیم.

فرستادن مقادیر پیش فرض :

```
#include <stdio.h>

MyFunc(int a=1, int b=2, int c=3)
{
	printf("%x %x %x\n", a, b, c);
}

main()
{
	MyFunc();
}
```

کد disassemble شده را در زیر مشاهده می‌کنید.

کد disassemble شده فرستادن مقادیر پیش فرض

```
main		proc near		; CODE XREF: start+AF↓p
	push	ebp
	mov	ebp, esp
	push	3
	push	2
	push	1
```

همان‌طور که مشاهده می‌کنید تمام مقادیر پیش‌فرض توسط کامپایلر به تابع فرستاده شده است.

```
	call	MyFunc

	add	esp, 0Ch
	pop	ebp
	retn
main		endp
```

مقادیر بازگشتی توابع

همان‌طور که می‌دانید معمولاً مقدار بازگشتی تابع همان مقداری است که توسط عملگر return باز می‌گردد.

```
int xdiv(int a, int b, int *c=0)
{
	if (!b) return -1;
	if (c) c[0]=a % b;
	return a / b;
}
```

تابع xdiv مقدار تقسیم صحیح آرگومان a بر b را باز می‌گرداند. البته باقی مانده درc قرار می‌گیرد که با ارجاع به تابع فرستاده شده است. همان‌طور که می‌بینید تابع چند مقدار را بازگردانده است. در ادامه روش‌های گوناگونی که از آنها به‌منظور ارسال و مدیریت مقادیر بازگشتی استفاده می‌شود را مورد بررسی قرار خواهیم داد.

مکانیزم‌های مقادیر بازگشتی :

۱- بازگرداندن مقادیر با استفاده از عملگر return (از طریق ثبات یا پشته پردازنده کمکی)

۲- بازگرداندن مقادیر از طریق ارجاع آرگومان‌ها

۳- بازگردان مقادیر از طریق heap

٤- بازگرداندن مقادیربا استفاده از متغیرهای سراسری

٥- بازگردان مقادیر از طریق پرچم‌های CPU

بازگرداندن مقادیر با استفاده از عملگر return

براساس قرارداد مقدار بازگشتی توسط عملگر return در ثابت EAX قرار گرفته و در صورتی که نتیجه از ظرفیت ثبات تجاوز کند ۳۲ بیت بالای علموند در EDX بارگذاری می‌شود.

در اکثر مواقع نتایجی از نوع float از طریق پشته پردازنده کمکی باز می‌گردد. همچنین این مقادیر ممکن است از طریق ثبات‌های EDX : EAX بازگردانده شوند.

وقتی تابعی یک رکورد یا شئ که از صدها بایت تشکیل شده است را باز می‌گرداند، نه ثبات‌ها و نه پشته پردازنده کمکی برای بازگرداندن آن کافی نخواهند بود.

اگر فضایی برای بازگرداندن مقدار در ثبات‌ها وجود نداشته باشد کامپایلر بدون اطلاع برنامه‌نویس یک آرگومان مجازی به تابع می‌فرستد این آرگومان یک ارجاع به متغیر محلی است که نتیجه را ذخیره می‌کند. توابع struct mystruct myfunc (int a , int b) و تابع void myfunc (strtuct mystruct * my, int a , int b) پس از کامپایل شدن کدهای یکسانی تولید می‌کنند و غیر ممکن است که بتوانیم پیش تعریف دقیق آنها را از کد ماشین به دست آوریم.

مایکروسافت ویژوال C++ فقط یک سر نخ به ما می‌دهد و در این حالت یک اشاره‌گر به متغیری که بازگردانده می‌شود را باز می‌گرداند. با این وجود پیش‌تعریف بازسازی شده به این صورت می‌باشد.

```
Struct    mystruct * myfunc (Struct mystruct  * my , int a , int b)
```

وقتی یک تابع یک مقدار را در EAX یا EDX ذخیره می‌کند و اجرای آن متوقف می‌شود شناسایی نوع تابع از طریق جداولی که در ادامه آمده به سختی صورت می‌گیرد. اگر ثبات‌ها بدون استفاده رها شده‌اند بهترین نتیجه و پاسخ نوع void است یعنی هیچ مقداری باز نخواهد گشت.

تحلیل و بررسی تابع فراخواننده اطلاعات دقیق بیشتری را درباره چگونگی دسترسی تابع فراخوانی شده به ثبات EAX فراهم می‌آورد. برای مثال نوع char معمولاً نیمه پایین ثبات EAX را آدرس‌دهی می‌کند و با استفاده از عملگر AND منطقی بایت‌های نیمه بالای ثبات EAX را صفر می‌کند. به‌نظر می‌رسد در صورتی که تابع فراخواننده از مقدار قرار داده شده در ثبات‌های EAX و یا EDX توسط تابع فرخوانی شده استفاده نکند، مقدار برگشتی void بوده است. ولی این مطلب همیشه صحیح نیست و ممکن است برنامه‌نویس از مقدار برگشتی تابع استفاده نکرده باشد.

Type (Length)	Returned via
1 byte	`AL` *or* `AX`
2 bytes	`AX`
4 bytes	`DX:AX`
Real	`DX:BX:AX`
Float	`DX:AX` *or* Coprocessor stack
Double	Coprocessor stack
Near pointer	`AX`
Far pointer	`DX:AX`
More than 4 bytes	Implicit argument by reference

جدول (۷-۷) مکانیزم بازگردانی مقدار با استفاده از عملگر Return در کامپایلرهای ۱۶ بیتی

Type (Length)	Returned via
1 byte	`AL` *or* `AX` *or* `EAX`
2 bytes	`AX` *or* `EAX`
4 bytes	`EAX`
8 bytes	`EDX:EAX`
Float	Coprocessor stack *or* `EAX`
Double	Coprocessor stack *or* `EDX:EAX`
Near pointer	`EAX`
More than 8 bytes	Implicit argument by reference

جدول (۷-۸) مکانیزم بازگردانی مقدار با استفاده از عملگر Return در کامپایلرهای ۳۲ بیتی

مثال بعدی مکانیزم استفاده شده برای بازگردانی مقادیر انواع را نشان می‌دهد.

بازگردانی مقادیر انواع اصلی

```
#include <stdio.h>
#include <malloc.h>
```

نمایش بازگردانی یک مقدار از نوع char توسط عملگر return

```
char char_func (char a, char b)

{
	return a+b;
}
```

نمایش بازگردانی یک متغیر از نوع int توسط عملگر return

```
int int_func(int a, int b)
{
	return a+b;
}
```

نمایش بازگردانی یک متغیر از نوع int 64 توسط عملگر return

```
__int64 int64_func(__int64 a, __int64 b)
{
	return a+b;
}
```

نمایش بازگردانی یک متغیر از نوع اشاره گر به int توسط عملگر return

```
int* near_func(int* a, int* b)
{
	int *c;
	c=(int *)malloc(sizeof(int));
	c[0]=a[0]+b[0];
	return c;
}

main()
{
	int a;
	int b;
	a=0x666;
	b=0x777;
	printf("%x\n",
```

```
        char_func(0x1,0x2)+
        int_func(0x3,0x4)+
        int64_func(0x5,0x6)+
        near_func(&a,&b)[0]);
}
```

کد disassemble شده این مثال با استفاده از کامپایلر مایکروسافت ویژوال C++ با تنظیمات پیش فرض موجود ، به صورت زیر است.

کد disassemble شده بازگردانی مقادیر انواع اصلی کامپایل شده توسط ویژوال C++

```
char_func       proc near           ; CODE XREF: main+1A↓p

arg_0           = byte ptr 8
arg_4           = byte ptr 0Ch

        push    ebp
        mov     ebp, esp
```

قاب پشته باز می‌شود.

```
        movsx eax, [ebp+arg_0]
```

آرگومان arg_0 از نوع کاراکتر علامتدار در EAX بارگذاری شده و به اندازه int گسترده می‌شود.

```
        movsx ecx, [ebp+arg_4]
```

آرگومان arg_4 از نوع کاراکتر علامت‌دار در ECX بارگذاری شده و به‌اندازه int گسترده می‌شود.

```
        add     eax, ecx
```

آرگومان‌های arg_0 و arg_4 که به اندازه int گسترده شده‌اند با هم جمع شده و در EAX ذخیره می‌شوند. این کار برای فراهم آورن مقدار بازگشتی تابع انجام می‌شود. متاسفانه شناسایی دقیق نوع غیر ممکن بوده و می‌تواند int یا char باشد. از این دو گزینه int محتمل تر است. جمع دو آرگومان char به دلایل امنیتی باید در int قرار بگیرد. در غیر این صورت امکان سرریز وجود دارد.

```
        pop     ebp
        retn
char_func       endp

int_func        proc near           ; CODE XREF: main+29↓p

arg_0           = dword ptr 8
arg_4           = dword ptr 0Ch
```

```
        push    ebp
        mov     ebp, esp
```

قاب پشته باز می‌شود.

```
        mov     eax, [ebp+arg_0]
```

مقدار آرگومان arg_0 در EAX بارگذاری می‌شود.

```
        add     eax, [ebp+arg_4]
```

مقدار آرگومان‌های arg_0 و arg_4 با هم جمع شده و حاصل در ثبات EAX ذخیره می‌شود. این مقداری است که توسط تابع بازگردانده می‌شود و نوع احتمالی آن int می‌باشد.

```
        pop     ebp
        retn
int_func        endp

int64_func      proc near               ; CODE XREF: main+40↓p

arg_0           = dword ptr 8
arg_4           = dword ptr 0Ch
arg_8           = dword ptr 10h
arg_C           = dword ptr 14h

        push    ebp
        mov     ebp, esp
```

قاب پشته باز می‌شود.

```
        mov     eax, [ebp+arg_0]
```

مقدار آرگومان arg_0 در EAX بارگذاری می‌شود.

```
        add     eax, [ebp+arg_8]
```

آرگومان‌های arg_0 و arg_8 با هم جمع می‌شوند.

```
        mov     edx, [ebp+arg_4]
```

مقدار آرگومان arg_4 در EDX بارگذاری می‌شود.

```
        adc edx, [ebp+arg_C]
```

آرگومان‌های arg_4 و arg_c با هم جمع شده و نتیجه با رقم نقلی که از جمع arg_0 و arg_8 باقی مانده جمع می‌شود. Arg_0 و arg_4 مانند arg_8 و arg_c نیمه‌های دو آرگومان‌های از نوع int 64 هستندکه با هم جمع خواهند شد. پس نتیجه این محاسبات از طریق ثبات‌های EDX : EAX باز خواهد گشت.

```
        pop     ebp
        retn
int64_func      endp

near_func       proc near               ; CODE XREF: main+54↓p

var_4           = dword ptr -4
arg_0           = dword ptr 8
arg_4           = dword ptr 0Ch

        push    ebp
        mov     ebp, esp
```

قاب پشته باز می‌شود.

```
        push    ecx

        push    4
        call    _malloc
        add     esp, 4
```

مقدار ECX ذخیره شده و چهار بایت از heap اختصاص داده می‌شود.

```
        mov     [ebp+var_4], eax
```

اشاره گر اختصاص داده شده به حافظه در متغیر var_4 قرار داده می‌شود.

```
        mov     eax, [ebp+arg_0]
```

مقدار آرگومان arg_0 در EAX بارگذاری می‌شود.

```
        mov     ecx, [eax]
```

مقدار int ارجاع داده شده توسط ثبات EAX در ECX بارگذاری می‌شود. آرگومان arg_0 از نوع int * می‌باشد.

```
        mov     edx, [ebp+arg_4]
```

مقدار آرگومان arg_4 در EDX بارگذاری می‌شود.

```
        add     ecx, [edx]
```

مقدار int خانه حافظه‌ای که توسط ثبات EDX به آن اشاره می‌شد با arg_0 * جمع می‌شود. نوع آرگومان arg_40 ، * int می‌باشد.

```
        mov     eax, [ebp+var_4]
```

اشاره‌گر به بلوک حافظه اختصاص داده شده از heap در EAX بارگذاری می‌شود.

```
        mov     [eax], ecx
```

حاصل جمع arg_0 * و arg_4 * در heap کپی می‌شود.

```
        mov     eax, [ebp+var_4]
```

اشاره‌گر به بلوک حافظه اختصاص داده شده از heap در EAX بارگذاری می‌شود. این مقداری است که توسط تابع بازگردانده می‌شود. پیش‌تعریف تابع به‌نظر می‌رسد که به این صورت باشد.

```
Int*  myfunc (int *a , int *b)

        mov     esp, ebp
        pop     ebp
        retn
near_func       endp

main            proc near           ; CODE XREF: start+AF↓p

var_8           = dword ptr -8
var_4           = dword ptr -4

        push    ebp
        mov     ebp, esp
```

قاب پشته باز می‌شود.

```
        sub     esp, 8
```

فضا به متغیرهای محلی اختصاص داده می‌شود.

```
        push    esi
        push    edi
```

ثبات‌ها در پشته ذخیره می‌شوند.

```
mov     [ebp+var_4], 666h
```

مقدار 0x666 در متغیر var_4 که متغیری محلی و از نوع int می‌باشد ذخیره می‌شود.

```
mov     [ebp+var_8], 777h
```

مقدار 0x777 در متغیر محلی var_8 که از نوع int می‌باشد ذخیره می‌شود.

```
push    2
push    1
call    char_func
add     esp, 8
```

تابع char_func (1, 2) فراخوانی می‌شود.

```
movsx   esi, al
```

مقدار بازگشتی تابع (char) به اندازه int علامتدار گسترش داده می‌شود.

```
push    4
push    3
call    int_func
add     esp, 8
```

تابع (٣ و ٤) int_func فراخوانی می‌شود. این تابع مقدار int باز می‌گرداند.

```
add     eax, esi
```

محتوای ESI با مقداری که تابع باز می‌گرداند جمع می‌شود.

```
cdq
```

مقدار دو کلمه‌ای (double word) که در ثبات EAX قرار دارد، به مقدار ٤ کلمه ای تبدیل شده و در ثبات‌های EDX : EAX قرار داده می‌شود. این نشان می‌دهدکه مقدار بازگشتی تابع از int به int 64 تبدیل شده است. البته هدف این کار همچنان نامعلوم است.

```
mov     esi, eax
mov     edi, edx
```

مقدار ٤ کلمه‌ای گسترش یافته، در ثبات‌های EDI : ESI کپی می‌شود.

```
push    0
push    6
push    0
push    5
call    int64_func
add     esp, 10h
```

تابع int64_func (5 , 6) فراخوانی می‌شود. این تابع مقداری از نوع int 64 باز می‌گرداند. حال مفهوم و هدف گسترش‌های قبلی روشن شد.

```
add     esi, eax
adc edi, edx
```

مقدار بازگشت داده شده توسط int64_func با مقدار ٤ کلمه‌ای موجود در ثبات‌های EDI : ESI جمع می‌شود.

```
lea     eax, [ebp+var_8]
```

اشاره‌گر به متغیر var_8 در EAX بارگذاری می‌شود.

```
push    eax
```

اشاره‌گر var_8 به‌عنوان آرگومان به تابع near_func فرستاده می‌شود.

```
lea     ecx, [ebp+var_4]
```

اشاره‌گر به متغیر var_4 در ECX بارگذاری می‌شود.

```
push    ecx
```

اشاره‌گر var_4 به‌عنوان آرگومان به تابع near_func فرستاده می‌شود.

```
call    near_func
add     esp, 8
```

تابع near_func فراخوانی می‌شود.

```
mov     eax, [eax]
```

همان‌طور که قبلاً اشاره شد این تابع اشاره‌گری به متغیری از نوع int را در ثبات EAX باز می‌گرداند. حال مقدار این متغیر در EAX بارگذاری می‌شود.

```
        cdq
```

EAX به ٤ کلمه گسترش داده می‌شود.

```
        add     esi, eax
       adc edi, edx
```

دو، چهار کلمه ای با هم جمع می‌شوند.

```
        push    edi
        push    esi
```

حاصل این جمع به تابع printf فرستاده می‌شود.

```
         push    offset unk_406030
```

اشاره‌گر به رشته کنترلی فرستاده می‌شود.

```
        call    _printf
        add     esp, 0Ch

        pop     edi
        pop     esi
        mov     esp, ebp
        pop     ebp
        retn
main            endp
```

مقادیر بازگشتی از طریق آرگومان‌های فرستاده شده با ارجاع

مراحل شناسایی مقادیر بازگشتی از طریق آرگومان‌های فرستاده شده با ارجاع، بسیار شبیه مراحل شناسایی آرگومان‌های توابعی هستند که قبلاً مورد بررسی قرار گرفتند. در این قسمت باید اشاره‌گرها را از میان آرگومان‌های فرستاده شده به تابع شناسایی کرد و آنها را به عنوان مقادیر بازگشتی کاندید کرد. سپس باید بررسی شود که آیا اشاره‌گر به متغیرهای مقداردهی نشده در میان آنها وجود دارد یا نه. تحلیل و بررسی تابع فراخوانی شده شرایط را بهتر توضیح می‌دهد. در این مراحل تمامی عملیاتی که متغیرهای فرستاده شده با ارجاع را دستکاری می‌کنند موردنظر ما هستند.

بازگردانی مقدار از طریق متغیرهایی که با ارجاع فرستاده شدند

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

```
void Reverse{char *dst, const char *src)
{
	strcpy(dst,src);
	_strrev( dst);
}
```

رشته src معکوس شده و در رشته dst نوشته می‌شود.

```
void Reverse(char *s) {
	_strrev( s );
}
```

رشته s معکوس شده و نتیجه حاصل در خود s ریخته می‌شود.

```
int sum(int a,int b)
```

این تابع حاصل جمع دو آرگومان را باز می‌گرداند.

```
{
	a+=b; return a;
}
```

آرگومان‌های فرستاده شده با ارجاع می‌توانند دستکاری شوند و همانند متغیرهای محلی با آنها رفتار می‌شود.

```
main()
{
	char s0[]="Hello, Sailor!";
	char s1 [100];

	Reverse(&s1[0], &s0[0]);
	printf("%s\n", &s1[0]);
```

رشته so معکوس شده و در s1 ریخته می‌شود.

```
	Reverse(&s1[0]);
	printf("%s\n", &s1 [0]);
```

رشته s1 بازنویسی شده سپس معکوس می‌شود.

```
	printf("%x\n", sum(0x666, 0x777));
```

حاصل جمع دو عدد چاپ می‌شود.

کد disassemble شده باز گردانی مقادیر از طریق متغیرهای فرستاده شده با ارجاع :

```
main            proc near       ; CODE XREF: start+AF↓p

var_74          = byte ptr -74h
var_10          = dword ptr -10h
var_C           = dword ptr -0Ch
var_8           = dword ptr -8
var_4           = word ptr -4

        push    ebp
        mov     ebp, esp
```

قاب پشته باز می‌شود.

```
        sub     esp, 74h
```

حافظه به متغیرهای محلی اختصاص می‌یابد.

```
        mov     eax, dword ptr aHelloSailor ; "Hello, Sailor!"
```

٤ بایت اول رشته "Hello, Sailor!" در EAX قرار داده می‌شود. کامپایلر احتمالاً رشته را در متغیرهای محلی کپی می کند.

```
        mov     [ebp+var_10], eax
        mov     ecx, dword ptr aHelloSailor+4
        mov     [ebp+var_C], ecx
        mov     edx, dword ptr aHelloSailor+8
        mov     [ebp+var_8], edx
        mov     ax, word ptr aHelloSailor+0Ch
        mov     [ebp+var_4], ax
```

رشته "Hello , sailor!" در متغیر محلی var_10 کپی می‌شود انتظار می‌رود نوع این متغیر char s[0x10] باشد. مقدار 0x10 از شمارش تعداد بایت‌های کپی شده به دست آمده است. ٤ تکرار ٤ بایتی در مجموع ١٦ بایت می‌شود.

```
        lea     ecx, [ebp+var_10]
```

اشاره‌گر به متغیر var_10 که شامل رشته "Hello, sailor!" می‌باشد در ECX بارگذاری می‌شود.

```
        push    ecx ;
```

اشاره‌گر به رشته "Hello , sailor" به تابع Reverse_1 فرستاده می‌شود. IDA نوع متغیر را اشتباه به دست آورده است. باز خوانی این موضوع که چگونه رشته کپی شده، اشتباه IDA را روشن می‌کند.

```
lea     edx, [ebp+var_74]
```

اشاره‌گر به متغیر محلی مقدار دهی نشده var_74 در ECX بارگذاری می‌شود.

```
push    edx
```

اشاره‌گر به متغیر از نوع char با تعریف s1[100] به تابع Reverse_1 فرستاده می‌شود. مقدار ۱۰۰ از کم کردن offset متغیر var_74 از var_10 به دست آمد. متغیر var_10 که بعد از var_74 قرار گرفته شامل رشته "Hello , sailor!" می‌باشد.0x74 – 0x10=0x64 . این مقدار برابر ۱۰۰ در نمایش دهدهی می‌باشد. با مشاهده فرستادن یک اشاره‌گر به متغیر مقداردهی نشده، می‌توانیم حدس بزنیم که تابع می‌خواهد توسط آن مقداری را بازگرداند.

```
call    Reverse_1
add     esp, 8
```

تابع Reverse_1 فراخوانی می شود.

```
lea     eax, [ebp+var_74]
```

اشاره‌گر به متغیر Var_74 در EAX بارگذاری می‌شود.

```
push    eax
```

اشاره‌گر به متغیر Var_74 به تابع Printf فرستاده می‌شود. تابع فرخواننده این متغیر را مقداردهی نکرده است. در نتیجه می‌توانیم حدس بزنیم که تابع فراخوانی شده می‌خواهد از طریق این متغیر مقداری را بازگرداند، تابع Reverse_1 ممکن است متغیر Var_10 را دستکاری کرده و در آن تغییری ایجاد کند اما بدون مطالعه کد تابع نمی‌توان مطمئن بود.

```
push    offset unk_406040
call    _printf
add     esp, 8
```

تابع Printf به منظور نمایش رشته خروجی فراخوانی می‌شود.

```
lea     ecx, [ebp+var_74]
```

ثبات ECX با اشاره‌گر به متغیر Var_74 پرمی‌شود. این متغیر ظاهراً شامل مقدار بازگشتی از تابع Reverse_1 می‌باشد.

```
push    ecx
```

اشاره‌گر به متغیر Var_74 به تابع Reverse_2 فرستاده می‌شود. تابع Reverse_2 نیز ممکن است مقدار بازگشتی خود را در متغیر Var_74 قرار دهد یا مقدار این متغیر را تغییر دهد یا اصلاً هیچ مقداری را بازنگرداند.

```
call    Reverse_2
add     esp, 4
```

تابع Reverse_2 فراخوانی می‌شود.

```
lea     ecx, [ebp+var_74]
```

اشاره‌گر به متغیر Var_74 در EDX بارگذاری می‌شود.

```
push    edx
```

اشاره‌گر به متغیر Var_74 به تابع Printf فرستاده می‌شود. باتوجه به اینکه از مقدار بازگشتی تابع که توسط ثبات‌های EDX:EAX نگهداری می‌شود، استفاده نشده است، می‌توانیم حدس بزنیم که این تابع، مقدار بازگشتی خود را به جای ثبات‌ها در متغیر Var_74 قرار می دهد. البته این موضوع تنها یک فرض است.

```
push    offset unk_406044
call    _printf
add     esp, 8
```

تابع Printf فراخوانی می‌شود.

```
push    777h
```

مقدار 0x777 از نوع int به تابع Sum فرستاده می‌شود.

```
push    666h
```

مقدار 0x444 از نوع int به تابع Sum فرستاده می‌شود.

```
call    Sum
```

```
        add     esp, 8
```

تابع Sum فراخوانی می‌شود.

```
        push    eax
```

ثبات EAX که شامل مقدار بازگشتی از تابع Sum می‌باشد به عنوان آرگومان به تابع Printf فرستاده می‌شود.

```
        push    offset unk_406048
        call    _printf
        add     esp, 8
```

تابع Printf فراخوانی می‌شود.

```
        mov     esp, ebp
        pop     ebp
```

قاب پشته بسته می‌شود.

```
        retn
main            endp
```

پیش تعریف تابع به‌صورت int _cdecl Reverse_1 (chart*,int) اشتباه است. همان‌طور که قبلاً از تابع فراخواننده استنباط شد پیش تعریف تابع باید به این صورت باشد Reverse (char *dst , char *src) آرگومان چپ یک اشاره‌گر به بافر مقداردهی نشده است که به عنوان مقصد درنظر گرفته می‌شود و آرگومان سمت راست نیز به‌عنوان Source درنظر گرفته می‌شود.

```
Reverse_1       proc near               ; CODE XREF: main+32↑p

arg_0           = dword ptr 8
arg_4           = dword ptr 0Ch

        push    ebp
        mov     ebp, esp
```

قاب پشته باز می‌شود.

```
        mov     eax, [ebp+arg_4]
```

آرگومان arg_4 در EAX بارگذاری می‌شود.

```
push    eax
```

آرگومان arg_4 به تابع Strcpy فرستاده می‌شود.

```
mov     ecx, [ebp+arg_0]
```

مقدار آرگومان arg_0 در ECX بارگذاری می‌شود.

```
push    ecx
```

مقدار آرگومان arg_0 به Strcpy فرستاده می‌شود.

```
call    strcpy
add     esp, 8
```

محتوای رشته‌ای که توسط arg_4 به آن اشاره می‌شود در بافری که arg_0 به آن اشاره می‌کند کپی می‌شود.

```
mov     edx, [ebp+arg_0]
```

مقدار ثبات EDX توسط محتوای آرگومان arg_4 بارگذاری می‌شود. این آرگومان به بافری که شامل رشتهٔ کپی شده است اشاره می‌کند.

```
push    edx
```

آرگومان arg_0 به تابع Strrev_ فرستاده می‌شود.

```
call    __strrev
add     esp, 4
```

تابع Strrev رشته‌ای که توسط arg_0 به آن اشاره می‌شود را معکوس می‌کند. تابع Reverse_1 مقدار بازگشتی خود را از طریق arg_0 که با ارجاع فرستاده شده است بازمی‌گرداند و رشته‌ای که arg_4 به آن اشاره می‌کند بدون تغییر باقی می‌ماند. با این وجود پیش‌تعریف تابع به این صورت می‌باشد

void Reverse_1 (char *dst, const char *src) . توصیف‌کننده Const را نباید فراموش کرد زیرا نشان می‌دهد که اشاره‌گر Source به یک متغیر فقط خواندنی اشاره می‌کند.

```
        pop     ebp
```

قاب پشته بسته می‌شود.

```
        retn
Reverse_1       endp
```

پیش‌تعریف صحیح تابع به این صورت است:(* Int _cdecl Reverse_2 (Char

```
Reverse_2       proc near               ; CODE XREF: main+4F↑p

arg_0           = dword ptr 8

        push    ebp
        mov     ebp, esp
```

قاب پشته باز می‌شود.

```
        mov     eax, [ebp+arg_0]
```

محتوای آرگومان arg_0 در EAX بار گذاری می‌شود.

```
        push    eax ; char *
```

محتوای آرگومان arg_0 به تابع Strrev فرستاده می‌شود.

```
        call    __strrev
        add     esp, 4
```

رشته معکوس شده و نتیجه در همان جا قرار داده می‌شود. پس تابع Reverse_2 مقدار بازگشتی خود را از طریق آرگومان arg_0 باز می‌گرداند و اثبات می‌کند که فرضیه ما درست است.

```
        pop     ebp
```

قاب پشته بسته می‌شود.

```
        retn
```

براساس آخرین بررسی انجام شده، پیش تعریف Reverse_2 به این صورت خواهد بود. void Reverse_2 (char *s) .

```
Reverse_2       endp
```

```
Sum             proc near ; CODE XREF: main+72↑p

arg_0           = dword ptr 8
arg_4           = dword ptr 0Ch

        push    ebp
        mov     ebp, esp
```

قاب پشته باز می‌شود.

```
        mov     eax, [ebp+arg_0]
```

مقدار آرگومان arg_0 در EAX بارگذاری می‌شود.

```
        add     eax, [ebp+arg_4]
```

آرگومان‌های arg_0 و arg_4 با هم جمع شده و نتیجه در EAX قرار داده می‌شود.

```
        mov     [ebp+arg_0], eax
```

حاصل جمع arg_0 و arg_4 در arg_0 کپی می‌شود. ممکن است تصور کنید که این یک بازگردانی مقدار از طریق آرگومان می‌باشد. اما این فرض درست نیست زیرا آرگومان‌های فرستاده شده به تابع از پشته برداشته شده و از بین می‌روند. توجه داشته باشید که با آرگومان‌های فرستاده شده با مقدار، همانند متغیرهای محلی رفتار می‌شود.

```
        mov     eax, [ebp+arg_0]
```

حالا مقدار بازگشتی در ثبات EAX کپی می‌شود. پس پیش تعریف تابع به این صورت می‌باشد:

```
int Sum (int a, int b)

        pop     ebp
```

قاب پشته بسته می‌شود.

```
        retn
Sum             endp
```

بازگردانی مقدار از طریق متغیرهای سراسری

به‌طور معمول استفاده از متغیرهای سراسری برای بازگردانی مقدار روش مناسبی نیست. تمام متغیرهای سراسری می‌توانند متغیرهای مجازی یک تابع باشند و مقادیر را بازگردانند. هر تابع به

دلخواه خود می‌تواند آنها را دستکاری کرده و تغییراتی را در آن‌ها ایجاد نماید. با تجزیه و تحلیل تابع فراخواننده، فرستادن و بازگردانی مقدار توسط متغیرهای سراسری قابل بررسی نبوده و تنها تحقیق و بررسی دقیق در مورد کد تابع فراخوانده شده لازم است. مهم است که بدانیم آیا تابع فراخوانی شده متغیرهای سراسری را دستکاری می‌کند و اگر این کار را انجام می‌دهد این کار بر روی کدام یک از متغیرهای سراسری صورت می‌گیرد. با مرور بخش داده می‌توانیم کلیه متغیرهای سراسری را همراه با آفست‌های آنها بدست آورده و سپس ارجاع‌های انجام شده به آنها در قسمت کد را مورد بررسی قرار دهیم.

بازگردانی مقدار از طریق متغیرهای سراسری و ایستا.

```
#include <stdio.h>
char* MyFunc(int a)
{
       static char x[7][16]=("Monday", "Tuesday", "Wednesday", \
        "Thursday", "Friday", "Saturday", "Sunday"};
       return &x[a-1][0];
}

main()
{
       printf("%s\n", MyFunc(6));
}
```

کد disassemble شده این مثال با استفاده از کامپایلر ویژوال C++ با تنظیمات پیش‌فرض به‌صورت زیر است:

کد disassemble شده بازگردانی مقدار از طریق متغیرهای سراسری و ایستا:

```
MyFunc        proc near         ; CODE XREF: main+5↓p

arg_0         = dword ptr 8

       push   ebp
       mov    ebp, esp
```

قاب پشته باز می‌شود.

```
       mov    eax, [ebp+arg_0]
```

مقدار آرگومان arg_0 در EAX بارگذاری می‌شود.

```
       sub    eax, 1
```

از EAX یک واحد کم می‌شود. این مسئله می‌تواند تأییدی بر این موضوع باشد که arg_0 اشاره‌گر نیست. البته در زبان C عملیات ریاضی برروی اشاره‌گرها مجاز می‌باشد.

```
        shl eax, 4
```

در اینجا arg_0 در ١٦ ضرب می‌شود. چهار Shift به سمت راست معادل ٢ به توان ٤ (١٦) است.

```
        add     eax, offset aMonday; "Monday"
```

مقدار بدست آمده به اشاره‌گر پایه که به جدول رشته‌ها در بخش داده ارجاع می‌کند اضافه می‌شود. بخش داده شامل متغیرهای سراسری و متغیرهای ایستا می‌باشد.

```
pop     ebp
```

قاب پشته بسته می‌شود و اشاره‌گر به عنصری از آرایه که ثبات EAX به آن اشاره می‌کند بازگردانده می‌شود. همانطور که می‌بینید هیچ فرقی بین بازگردانی یک اشاره‌گر به حافظهٔ اختصاص یافته از heap و بازگردانی اشاره‌گر به متغیرهای ایستا در بخش داده نمی‌باشد.

```
        retn
MyFunc          endp

main            proc near               ; CODE XREF: start+AF↓p

        push    ebp
        mov     ebp, esp
```

قاب پشته باز می‌شود.

```
        push    6
```

مقداری از نوع int به تابع myfunc فرستاده می‌شود.

```
        call    MyFunc
        add     esp, 4
```

تابع myfunc فراخوانی می‌شود.

```
        push    eax
```

مقدار بازگردانده شده توسط تابع myfunc به تابع Printf فرستاده می‌شود. رشته مشخص کننده فرمت نشان می‌دهد که این یک اشاره‌گر به رشته می‌باشد.

```
        push    offset aS           ; "%s\n"
        call    _printf
        add     esp, 8

        pop     ebp
```

قاب پشته بسته می‌شود.

```
        retn
main            endp

aMonday         db 'Monday',0, 0, 0, 0, 0 ; DATA XREF: MyFunc+C↑o
```

نمایش این نوع ارجاع به یک داده رشته‌ای تقویت‌کنندهٔ این حدس است که این متغیر از نوع ایستا بوده است.

```
aTuesday db 'Tuesday',0,0,0,0,0,0,0,0,0
aWednesday db 'Wednesday',0,0,0,0,0,0,0,0,0,0,0
aThursday db 'Thursday',0,0,0,0,0,0,0,0,0
aFriday db 'Friday',0,0,0,0,0,0,0,0,0
aSaturday db 'Saturday',0,0,0,0,0,0,0,0,0
aSunday db 'Sunday',0,0,0,0,0
aS db '%s', 0Ah, 0              ; DATA XREF: main+E↑o
```

فصل هشتم

برنامه نویسی به زبان اسمبلی در ویندوز

فصل هشتم

مقدمه

چرا می‌خواهید از اسمبلی به جای C یا سایر زبان‌های برنامه‌نویسی استفاده کنید با وجود اینکه اسمبلی سخت‌تر از آنها می باشد؟

برنامه‌های نوشته شده با اسمبلی کم حجم و سریع هستند. در بسیاری از زبان‌های سطح بالا مانند زبان‌های هوش‌مصنوعی تهیه کد خروجی برای کامپایلر بسیارسخت است و کامپایلر باید سریع‌ترین و کوچکترین روش برای تهیه کد اسمبلی را کشف کند. اگر چه کامپایلرها پیشرفت می‌کنند و روز به روز بهتر کار می کنند ولی اگر خودتان کد اسمبلی را بنویسید مطمئناً برنامهٔ کوچکتر و سریع‌تری تهیه خواهد کرد. البته باید توجه داشته باشید که این کار بسیار سخت‌تر از برنامه‌نویسی با زبان‌های سطح بالا است.

تفاوت دیگری که با برخی زبان‌های سطح بالا وجود دارد اینست که آنها از Dll های زمان اجرا برای توابع خود استفاده می کنند. برای مثال Visual C++ از فایل Msvcrt.dll برای توابع خود استفاده می‌کند که حاوی توابع استاندارد C می‌باشد.

این روش در اکثر مواقع درست کار می‌کند ولی برخی مواقع مشکلاتی در زمینهٔ Versionهای این dllها به وجود می‌آید که کاربر را مجبور به تهیه و نصب آنها می‌کند. در مورد Visual C این مشکل جدی نیست زیرا فایل‌های مورد نیاز آن معمولاً با خود ویندوز نصب می‌شوند.

اسمبلی سریع‌ترین زبان برنامه‌نویسی است و به‌طور پیش‌فرض فقط از dllهای استاندارد سیستم مانند Kernel32 و User32 استفاده می‌کند.

معمولاً نوشتن پروژه‌های بزرگ با اسمبلی کاری سخت و طاقت‌فرسا است به همین علت بیشتر از آن برای نوشتن برنامه‌های کوچک و سریع و نیز نوشتن dllهایی برای استفاده در سایر زبان‌های برنامه‌نویسی برای قسمت‌هایی از برنامه که نیاز به سرعت بالا دارند استفاده می‌شوند. حال که با مزایا و معایب استفاده از اسمبلی آشنا شدید به بررسی خصوصیات برنامه‌های ۳۲ بیتی می‌پردازیم.

برنامه‌های ۳۲ بیتی

برنامه‌های ۳۲ بیتی در محیط محافظت شده‌ای اجرا می‌شوند که در زمان cpuهای ۸۰۲۸٦ ارائه شده است و به آن مد محافظت شدهٔ CPU (Protected CPU Mode) گفته می‌شود.

ویندوز هر برنامهٔ ۳۲ بیتی را در فضای حافظهٔ جداگانه‌ای که به‌صورت مجازی ایجاد کرده است اجرا می‌کند. هر برنامه توانایی آدرس‌دهی حداکثر ٤ گیگا بایت از حافظه را دارد.

هر برنامه ۳۲ بیتی تحت ویندوز باید به قوانینی که توسط ویندوز وضع شده است وفادار بماند. در غیر این صورت برنامه باعث ایجاد خطای مهلک حفاظت عمومی (General Protection Fault) می‌گردد.

هر برنامهٔ Win32 در فضای حافظهٔ خود تنها است (برخلاف Win16) . در Win16 همهٔ برنامه‌ها می‌توانند به فضای حافظهٔ یکدیگر دسترسی داشته باشند. ولی در Win32 اوضاع به این منوال نیست. این قابلیت شانس دسترسی یک برنامه به فضای data/code برنامهٔ دیگر را محدود می‌سازد.

مدل حافظه نیز به شدت با مدل‌های قدیمی ١٦ بیتی تفاوت دارد. تحت Win32، دیگر نیازی به نگرانی در مورد مدل حافظه یا سگمنت‌ها نداریم. تنها مدل حافظهٔ موجود مدل Flat است و دیگر خبری از محدودیت 64k برای هر سگمنت وجود ندارد. حافظه تشکیل شده است از 4GB فضای متوالی. همچنین این بدین معنی است که دیگر نیازی به سر و کله زدن با ثبات‌های سگمنت ندارید و می‌توانید از هر ثبات سگمنتی برای آدرس‌دهی هر نقطه‌ای از فضای حافظه استفاده کنید که این کمک بزرگی به برنامه‌نویسان می‌کند.

یکی از قوانینی که در هنگام برنامه‌نویسی تحت Win32 باید با آن آشنایی داشته باشید اینست که ویندوز از ثبات‌های esi، edi، ebp، ebx برای انجام کارهای داخلی خود استفاده می‌کند و انتظار هیچ‌گونه تغییری را در این ثبات‌ها ندارد. پس همیشه به‌خاطر داشته باشید که اگر از این ثبات‌ها در برنامه استفاده می کنید قبل از بازگردانی کنترل به ویندوز آنها را به‌صورت اولیهٔ خود بازگردانید. در مبحث بعد به ابزارهایی که برای برنامه‌نویسی با اسمبلی نیاز دارید اشاره خواهد شد.

حال که آشنایی نسبی با خصوصیات برنامه‌های ۳۲ بیتی پیدا کردید بهتر است برنامه‌نویسی را شروع کنید. برای این کار به یک سری ابزار نیاز دارید. معروف‌ترین بسته‌های نرم‌افزاری برای برنامه‌نویسی به زبان اسمبلی عبارتند از: Turbo Assembler ساخت شرکت Borland و Macro Assembler ساخت شرکت Microsoft . این دو از نظر روش برنامه‌نویسی و امکانات جانبی با هم

تفاوت‌هایی دارند. در این کتاب برای نوشتن کدها از Macro Assembler استفاده شده است که برای نوشتن برنامه‌های ۳۲ بیتی تحت ویندوز به نسخهٔ 6.11 یا بالاتر آن نیاز دارید.

نسخه 6.11 این کامپایلر در CD ضمیمه موجود می‌باشد

Tools\Masm6.11

Macro Assembler 6.11 از فایل ml.exe برای ساخت فایل Object برنامه (obj.) و از فایل Link.exe برای ساخت فایل اجرایی نهایی استفاده می‌کند که روش کار با آنها را در زیر مشاهده می‌کنید:

```
ml /c /coff  < نام فایل برنامه >
link /subsystem:windows   < برنامه  obj نام فایل  >
```

می توانید به جای انجام دو مرحلهٔ بالا از روش زیر برای ساخت فایل اجرایی برنامه استفاده کنید:

```
ml /coff   < نام فایل برنامه >
```

در صورت موفقیت‌آمیزبودن مراحل فوق فایل اجرایی برنامه که قابلیت اجرا تحت ویندوز را دارد ایجاد خواهد شد.

مزایای استفاده از Macro Assembler

۱- مقایسه و حلقه های تکرار

Macro Assembler یک سری دستورات ساده برای پیاده‌سازی ساختار حلقه‌ها و مقایسه‌ها دارد که عبارتند از:

.if , .elseif , .endif , .repeat , .until , .while , .endw , .break , .continue

.if

اگر تجربهٔ برنامه‌نویسی با زبان‌های سطح بالا مانند C، Pascal را داشته باشید حتماً ساختار مقایسه‌ای if / else را دیده‌اید. در زیر با یک مثال ساده روش پیاده‌سازی این ساختار را در ماکرو اسمبلر مشاهده می کنید:

```
.IF eax == 1
< If دستورات مورد نظر برای >
.ELSEIF eax == 3
< ELSEIF دستورات مورد نظر برای >
```

```
.ELSE
< دستورات مورد نظر براي ELSE >
.ENDIF
```

این ساختار بسیار مفید است زیرا دیگر شما مجبور نیستید که برنامه خود را باپرش‌های زیاد آشفته کنید. دستورات if تو در تو نیز مجاز می‌باشند برای مثال :

```
.IF eax == 1
       .IF ebx == 2
                < دستورات براي مجموع هر دو شرط >
          .ENDIF
.ENDIF
```

ولی به روش ساده‌تری هم می‌توان عمل بالا را انجام داد.

```
.IF (eax == 1  &&  ebx == 2)
            < دستورات براي مجموع هر دو شرط >
.ENDIF
```

لیست زیر عملگرهای قابل استفاده در ماکرو اسمبلر را نشان می‌دهد :

برابری	= =
نابرابری	! =
بزرگتربودن	>
کوچکتر بودن	<
بزرگتر یا مساوی	>=
کوچکتر یا مساوی	<=
نقیض	!

AND منطقی	&&
OR منطقی	||

.Repeat

این دستور بلوکی از دستورات را تا زمان برقرار بودن شرط معین اجرا می‌کند برای مثال :

```
.REPEAT
       < دستورات >
.UNTIL eax == 1
```

دستورات معین شده را تا زمان برقراری شرط eax == 1 انجام می دهد.

.While

این دستور دقیقاً همانند REPEAT عمل می‌کند با این تفاوت که شرط حلقه در ابتدا کنترل می‌شود، برای مثال:

```
.WHILE eax == 1
       < دستورات >
.ENDW
```

از این دستور برای خارج شدن از حلقه استفاده می‌شود. برای مثال :

```
.WHILE edx == 1
inc eax
.IF eax == 7
    .BREAK
.ENDIF
.ENDW
```

اگر شرط eax == 7 برقرار شود حلقهٔ WHILE خاتمه پیدا می‌کند.

.Continue

این دستور در حلقه سبب می‌شود که اجرای برنامه به ابتدای بلوک فرستاده شده و شرط حلقه در صورت وجود دوباره چک شود و دستورات بلوک از ابتدا اجرا شوند.

۲- دستور Invoke

می‌توان گفت که این بزرگترین مزیت ماکرو اسمبلر نسبت به دیگر اسمبلرها است که استفاده از توابع و فراخوانی آنها را بسیار ساده می‌کند. برای مثال:

```
روش عادی
push parameter3
push parameter2
push parameter1
call procedure

 دستورInvoke
invoke procedure, parameter1, parameter2, parameter3
```

کد ماشین ایجاد شده در هر دو روش دقیقاً یکسان است ولی Invoke ساده‌تر و قابل اعتمادتر می‌باشد.

برای استفاده از دستور Invoke ابتدا باید نمونهٔ اولیهٔ تابع را به برنامه معرفی کنید:

```
PROTO STDCALL testproc:DWORD, :DWORD, :DWORD
```

تعریف بالا پروسیجری به نام testproc را که ۳ متغیر Dword را به عنوان پارامترهای ورودی می‌گیرد، معرفی می کند. حال در صورت فراخوانی پروسیجر به صورت زیر

```
Invoke testproc, 1, 2, 3, 4
```

ماکرو اسمبلر پیغام خطا داده و به شما گوشزد می‌کند که پروسیجر testproc از ۳ پارامتر ورودی استفاده می‌کند نه ٤ تا !

همچنین ماکرو اسمبلر عمل کنترل نوع متغیرهای ورودی را به‌طور خودکار انجام می‌دهد و شما می‌توانید مطمئن باشید که پارامترهای دریافتی پروسیجر دقیقاً از همان نوع موردنظر هستند. در دستور Invoke می‌توانید از addr به جای offset استفاده کنید که روش مطمئن‌تری است.

پروسیجرها به‌صورت زیر تعریف می‌شوند:

```
testproc PROTO STDCALL :DWORD, :DWORD, :DWORD

code.
testproc proc param1:DWORD, param2:DWORD, param3:DWORD
ret
testproc endp
```

برای تعریف متغیرهای محلی درون یک پروسیجر می‌توانید به‌صورت زیر عمل کنید:

```
LOCAL< نوع متغیر> : <نام متغیر>
```

بدیهی است که از این متغیرها در خارج از پروسیجر مربوطه نمی‌توانید استفاده کنید.

اصول برنامه‌نویسی تحت Windows با ماکرو اسمبلر

در مباحث قبل با برخی از خصوصیات برنامه‌های ۳۲ بیتی و روش‌هایی که ویندوز برای مدیریت آنها به کار می‌گیرد آشنایی پیدا کردید. حال بهتر است به ساختار برنامه‌ها در ماکرو اسمبلر نگاهی بیاندازیم. در زیر اسکلت و ساختار یک برنامه را مشاهده می‌کنید که می‌خواهیم آن را بررسی کنیم.

```
.386
.MODEL Flat, STDCALL
.DATA
    <Your initialized data>
    ......
.DATA?
   <Your uninitialized data>
   ......
.CONST
   <Your constants>
   ......
.CODE
   <label>
    <Your code>
   .....
    end <label>
```

.386

این بخش یک directive (راهنمای کامپایلر) است و به اسمبلر می‌گوید که از مجموعه دستورالعمل‌های Cpuهای 80386 استفاده کند. می‌توانید از 0486 یا 0586 نیز استفاده کنید ولی مطمئن‌ترین راه استفاده از 386. است.

```
.MODEL FLAT, STDCALL
```

.Model

یک directive است که مدل حافظه مورد استفادهٔ برنامهٔ شما را مشخص می‌کند. همان‌طور که گفته شد تنها مدل موجود در Win32 مدل Flat است.

.Stdcall

برای ماکرو اسمبلر روش ارسال پارامترها را مشخص می‌کند. روش ارسال پارامترها ترتیب ارسال پارامترها به پروسیجرها را مشخص می‌کند که می‌تواند دو مقدار Stdcall یا Pascal را داشته باشد.

. DATA

. DATA?

. CONST

. CODE

هر چهار directive روی هم یک Section را می‌سازند. (به یاد دارید که در Win32 چیزی به نام Segment وجود ندارد) ولی شما می‌تواند تمام آدرس حافظهٔ برنامهٔ خود را به بخش‌های منطقی Logical تقسیم بندی کنید. شروع یک Section انتهای Section بعدی را علامت‌گذاری می کند.

Sectionها به دو دستهٔ data و code تقسیم‌بندی می شوند.

Data Sectionها به سه دسته تقسیم‌بندی می‌شوند.

.DATA

این بخش شامل داده‌های آغازین برنامه است. مقدار فضایی که داده‌های این بخش اشغال می‌کنند به حجم فایل اجرایی (exe.) افزوده می‌شود.

.DATA?

این بخش شامل داده‌ها و متغیرهایی است که باید از حافظه (memory) استفاده کنند. این بخش برای اسمبلر مشخص می‌کند که برنامهٔ شما برای شروع به چه مقدار حافظه نیاز دارد. فضایی که توسط متغیرهای این بخش اشغال می‌شود برروی حجم فایل اجرایی هیچ تأثیری ندارد.

.CONST

این بخش شامل مقادیر ثابت (constant)های برنامهٔ شما است که در موقع اجرا غیرقابل تغییر هستند.

فقط یک بخش برای کد برنامه وجود دارد (Code.). اینجا جایی است که برنامه شما نوشته می‌شود:

```
<label>
end<label>
```

این دو روی هم اندازهٔ کد برنامهٔ شما را برای اسمبلر مشخص می‌کنند. تمام کد برنامهٔ شما باید بین <label> و end <label> نوشته شوند.

ایجاد یک برنامه ساده

در این بخش می‌خواهیم یک برنامه ساده بنویسیم که به سادگی اسکلت و ساختار یک برنامهٔ Win32 را نشان می‌دهد.

کدها و فایل‌های مربوط به این قسمت در CD ضمیمه موجود می‌باشد

SourceCodes\Asm32\Chapter2

ویندوز منابع و وسایل زیادی را برای برنامه‌های خود فراهم می‌کند که مرکز همهٔ آنها API (Application Programming Interface) است. API شامل مجموعهٔ عظیمی از توابع است که درون ویندوز وجود دارد و تمام برنامه‌های Win32 به آن دسترسی دارند. این توابع در چندین فایل dll قرار دارند. که برخی از آنها عبارتند از : User32.dll ، Kernel32.dll ، Gdi32.dll

Kernel32 شامل توابعی است که با حافظه و Processها سروکار دارند. درحقیقت این توابع وظیفهٔ مدیریت حافظه و برنامه‌های ویندوز را بر عهده دارند.

User32 شامل توابعی است که وظیفهٔ مدیریت بخش رابط کاربر ویندوز را بر عهده دارند.

Gdi32 شامل توابعی است که عملکردهای گرافیکی ویندوز را کنترل می‌کنند.

علاوه بر این سه dll اصلی dll های دیگری نیز در ویندوز وجود دارند که برنامهٔ شما می تواند آنها را به کار گیرد. مشروط بر اینکه شما اطلاعات کافی در مورد توابع API مورد نیاز خود داشته باشید. این اطلاعات را می‌توانید از مرجع API (Win32 API Reference) دریافت کرده و یا می‌توانید از مجموعهٔ کامل CDهای راهنمای مایکروسافت برای برنامه‌نویسان (MSDN) که شامل کلیه اطلاعات مورد نیاز شما برای برنامه‌نویسی تحت ویندوز است، استفاده نمایید.

برنامه‌های تحت ویندوز به‌صورت دینامیک (پویا) به این dllهای ویندوز لینک می‌شوند. این بدان معنی است که توابع آنها به فایل اجرایی برنامهٔ شما لینک نمی‌شوند. ولی برای اینکه برنامهٔ شما بداند که درموقع اجرا از کجا باید تابع API موردنظر را پیدا کند باید اطلاعات موردنیاز را در فایل اجرایی خود قرار دهید. این اطلاعات به‌صورت آماده در کتابخانه‌هایی قرار دارند (Import Libraries) شما باید برنامهٔ خود را به کتابخانهٔ موردنظر لینک کنید. در غیر این‌صورت برنامه قادر به پیدا کردن توابع API موردنظر شما نمی باشد.

وقتی یک برنامهٔ ویندوز به حافظه بارگذاری می‌شود. ویندوز اطلاعات موجود در فایل اجرایی برنامه را می خواند. این اطلاعات شامل نام توابع مورد استفاده در برنامه و نام dll ای است که هر تابع در

آن قرار دارد. وقتی ویندوز این اطلاعات را پیدا کرد ابتدا dllها را بارگذاری کرده و سپس عمل تصحیح آدرس توابع را انجام می‌دهد. در نتیجه هرگونه باز خوانی توابع API کنترل را به تابع موردنظر منتقل می کند.

دو نوع از توابع API وجود دارند. یکی برای ANSI و دیگری برای Unicode. نام توابعی که مخصوص ANSI هستند با پسوند A می‌آید. برای مثال MessageBoxA. نوع دیگر که برای Unicode است با پسوند W (wide char) می‌آید. Win95 به‌صورت پیش‌فرض از ANSI و WinNT از Unicode پشتیبانی می کنند.

معمولاً با سیستم ANSI و رشته‌ها تحت این استاندارد آشنایی دارید. در این سیستم رشته متشکل از آرایه‌ای از کاراکترهای مختوم به صفر (Null) بوده و هر کاراکتر ANSI یک بایت فضا اشغال می‌کند. با اینکه استاندارد ANSI برای استانداردهای اروپایی مناسب است ولی نمی‌تواند برای سایر زبان‌های آسیایی که شامل چند صد نوع کاراکتر می‌باشد مناسب باشد. به همین علت سیستم Unicode ایجاد شد. هر کاراکتر Unicode ۲ بایت فضا اشغال کرده و می‌تواند تا ٦٥٥٣٦ نوع کاراکتر را پشتیبانی کند. ولی در اکثر مواقع شما از فایل ضمیمه‌ای استفاده می‌کنید که توانایی تشخیص تابع مناسب برای محیطی که قصد استفادهٔ آنرا دارید در خود دارد.

در زیر یک برنامهٔ بدون کد را مشاهده می کنید که قصد داریم کد را به آن اضافه کنیم:

```
.386
.model flat, stdcall
.data
.code
start:
end start
```

اجرای برنامه پس از اولین دستورالعملی که بعد از Label اصلی کد برنامه وجود دارد آغاز می‌شود که در مثال بالا Start است. دستورالعمل‌های کد برنامه یکی یکی تا آخر اجرا می‌شوند. با دستورالعمل‌هایی مانند Jne، Jmp، Je، Ret می‌توان روند اجرای دستورالعمل‌های برنامه را تغییر داد. در آخر وقتی برنامه قصد پایان دادن به کار خود و بازگشت به ویندوز را دارد باید از تابع ExitProcess استفاده کند.

```
ExitProcess proto uExitCode:DWORD
```

خط بالا نمونه اولیه (prototype) تابع نام دارد. نمونه اولیه تابع مشخصات یک تابع را برای اسمبلر مشخص می‌کند در نتیجه اسمبلر می‌تواند عملیات کنترل نوع (Type Checking) را برای ما انجام دهد. در زیر پیکربندی یک prototype را مشاهده می کنید:

```
FunctionName PROTO
...,[ParameterName]:DataType,[ParameterName]:DataType
```

در پیش تعریف قبل تابع ExitProcess با یک پارامتر ورودی از نوع DWORD معرفی می‌شود.

سعی کنید برای باز خوانی تابع از Invoke به جای دستور سادهٔ call استفاده کنید. در زیر نحوهٔ استفاده از دستور Invoke را مشاهده می‌کنید:

```
INVOKE  expression [,arguments]
```

بخش اول می‌تواند نام یک تابع و یا اشاره‌گر به یک تابع باشد. پارامترهای ارسالی به تابع با کاما از هم جدا می‌شوند.

اکثر پیش تعریف‌های توابع API در فایل‌های ضمیمهٔ اسمبلر وجود دارند که معمولاً در دایرکتوری include قرار دارند. پسوند این فایل‌ها (inc.) می‌باشد. پیش تعریف‌های توابع یک dll در فایل ضمیمه‌ای با همان نام ذخیره شده‌اند. برای مثال تابع ExitProcess در فایل kernel32.dll قرار دارد، پس فایل kernel32.inc حاوی پیش‌تعریف این تابع می‌باشد. حال به سراغ تابع ExitProcess برویم، در تعریف این تابع پارامتر uExitCode مقدار برگشتی برنامه را در موقع بازگشت به ویندوز مشخص می‌کند.

```
invoke ExitProcess, 0
```

خط بالا را بعد از start قرار دهید. حالا شما یک برنامه Win32 دارید که به سرعت خاتمه پیدا کرده و به ویندوز بازگشت پیدا می‌کند. با اینکه این برنامه کاری انجام نمی‌دهد ولی ساختار یک برنامه استاندارد Win32 را نشان می‌دهد.

```
.386
.model flat, stdcall
option casemap:none
```

```
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib
.data
.code
start:
        invoke ExitProcess,0
end start
```

ماکرو اسمبلر به‌طور پیش‌فرض به بزرگ یا کوچک بودن حروف حساس نیست. برای حساس کردن آن به بزرگی و کوچکی حروف (Case-Sensitive) از دستور "option casemap:none" استفاده می‌کنیم.

فایل ضمیمه Windows.inc حاوی ساختارها و ثابت‌های مورد استفاده در برنامه‌نویسی تحت ویندوز است و فایل Kernel32.inc حاوی پیش تعریف‌های توابع موجود در Kernel32.dll است که توسط کتابخانه ورودی Kernel32.lib ارائه می‌شوند. ما به این سه فایل برای برنامه خود نیاز داریم پس آنها را با استفاده از دایرکتوری‌های Include (برای فایل‌های ضمیمه) و Includelib (برای کتابخانه های ورودی) به برنامه ضمیمه می کنیم. بهتر است مسیر کامل فایلهایی که ضمیمه می شوند را بعد از دایرکتیو ذکر کنید. برای مثال: (c:\masm32\include\user32.inc)

حال برنامه قبل را تکمیل می کنیم به طوری که با اجرای آن پیغامی بر روی صفحه نشان داده نشود. ما این پیغام را توسط یک جعبه پیغام (MessageBox) نمایش می دهیم. حال به کد برنامه توجه کنید:

```
.386
.model flat,stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib
include \masm32\include\user32.inc
includelib \masm32\lib\user32.lib
.data
db "HELLO WORLD !!!",0  MsgBoxCaption
db "HELLO WORLD !!!",0    MsgBoxText
.code
start:
   invoke MessageBox, NULL, addr MsgBoxText, addr MsgBoxCaption,
MB_OK
   invoke ExitProcess, NULL
end start
```

برای استفاده از جعبه پیغام باید از تابع MessageBox که در فایل User32.dll قرار دارد استفاده کنیم. پس در ابتدای برنامه فایل‌های مورد نیاز را به لیست فایل‌های ضمیمه اضافه می‌کنیم.

در بخش Data دو رشته مختوم به صفر را معرفی می‌کنیم که اولی عنوان جعبه پیغام و دومی پیغام نمایش داده شده در جعبه پیغام می باشد.

ما در برنامه از دو ثابت MB-OK و NULL استفاده کرده‌ایم که در فایل ضمیمه Windows.inc معرفی شده‌اند.

```
Invoke MessageBox, NULL, addr MsgBoxText, addr MsgBoxCaption, MB_OK
```

پارامتر اول در تابع MessageBox شماره دسترسی به پنجره پدر را مشخص می‌کند. چون در اینجا ما هیچ پنجره‌ای نداریم، به این پارامتر مقدار NULL یا صفر را می‌دهیم. دو پارامتر بعدی آدرس رشته‌های عنوان پنجره و پیغام را تعیین می‌کند. پارامتر آخر تابع نوع جعبه پیغام را مشخص می‌کنند. ما در اینجا با استفاده از ثابت MB-OK تعیین می‌کنیم که یک جعبه پیغام ساده یا یک دکمه OK نیاز داریم.

ایجاد یک پنجره ساده

کدها و فایل‌های مربوط به این قسمت در CD ضمیمه موجود می‌باشد

SourceCodes\Asm32\Chapter3

برنامه های ویندوز به طور وسیعی به توابع API جهت طراحی رابط کاربر خود وابسته هستند. این روند هم به سود کاربران و هم به سود برنامه نویسان است. کاربران دیگر مجبور نیستند که برای کار با هر برنامه رابط کاربر جدیدی را کشف کنند. رابط کاربر برنامه های ویندوز برای تمام برنامه نویسان یکسان است و کدهای طراحی رابط کاربر به طور آماده و مطمئن در اختیار آنها قرار دارد.

در زیر به طور خلاصه مواردی را که برای ایجاد یک پنجره باید انجام شود می بینید:

۱- گرفتن شناسه یکتا برای برنامه.

۲- ثبت کلاس پنجره (که در صورت استفاده از پنجره‌های از پیش تهیه شده مانند MessageBox یا Dailog Box نیازی به انجام آن ندارید).

۳- ایجاد پنجره از روی کلاس تعریف شده در مرحله قبل.

٤- نمایش پنجره (در صورت نیاز)

٥- بازکشی منطقه کاری (Client Area) پنجره.

٦- وارد شدن به یک حلقه بینهایت برای چک کردن پیغام‌های رسیده به پنجره.

۷- در صورت دریافت پیغام آن پیغام توسط زیر برنامه مربوطه پردازش می‌شود. هر زیر برنامه مسئول پردازش پیغام خاصی از ویندوز می باشد.

۸- خارج شدن از برنامه در صورت بسته شدن آن توسط کاربر.

همان‌گونه که مشاهده کردید ساختار یک برنامه ویندوز بسیار پیچیده‌تر از یک برنامه تحت DOS است. دلیل این امر این است که ساختار سیستم عامل‌های DOS و ویندوز با هم تفاوت‌های بسیار زیادی دارند. برنامه های ویندوز برای اینکه قابلیت همزیستی مسالمت آمیز با هم را داشته باشند باید از قوانین سختی پیروی کنند. شما نیز به عنوان یک برنامه‌نویس باید حساسیت زیادی نسبت به سبک و عادات برنامه‌نویسی خود داشته باشید.

قبل از وارد شدن به جزئیات پیچیده برنامه‌نویسی با Win32 ASM به چند نکته مهم اشاره می‌کنیم که باعث سهولت در برنامه‌نویسی شما می شود.

تمام ساختارها و ثبات‌های ویندوز و پیش تعریف توابع را در یک فایل ضمیمه قرار داده و آن را در اول برنامه خود معرفی کنید. این کار به میزان زیادی از صرف وقت و خطاهای احتمالی شما می‌کاهد. درحال حاضر کامل‌ترین فایل ضمیمه برای ماکرو اسمبلر فایل Windows.inc نوشته Hutch می‌باشد که اکثر ثابت‌ها و ساختارهای ویندوز را در خود دارد. همچنین می‌توانید خودتان تعاریف ثابت‌ها و ساختارهای مورد نیاز را ایجاد کنید.

وقتی پیش تعریفها و ثابت‌های ویندوز را به برنامه خود معرفی می‌کنید سعی کنید از نام‌های اصلی آنها استفاده کنید. این امر باعث عدم سردرگمی شما در هنگام کار با مراجع API می‌گردد. حال به سراغ کدهای برنامه پنجره می‌رویم ...

```
.386
.model flat,stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\user32.inc
includelib \masm32\lib\user32.lib
include \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib
WinMain proto :DWORD,:DWORD,:DWORD,:DWORD
.DATA                    ; initialized data
ClassName db "SimpleWinClass",0
AppName db "Our First Window",0
.DATA?
hInstance HINSTANCE ?
CommandLine LPSTR ?
.CODE
start:
invoke GetModuleHandle, NULL

mov hInstance,eax
invoke GetCommandLine


mov CommandLine,eax
invoke WinMain, hInstance,NULL,CommandLine, SW_SHOWDEFAULT
invoke ExitProcess, eax
WinMain proc
hInst:HINSTANCE,hPrevInst:HINSTANCE,CmdLine:LPSTR,CmdShow:DWORD
    LOCAL wc:WNDCLASSEX
    LOCAL msg:MSG
    LOCAL hwnd:HWND
```

```
    mov   wc.cbSize,SIZEOF WNDCLASSEX
    mov   wc.style, CS_HREDRAW or CS_VREDRAW
    mov   wc.lpfnWndProc, OFFSET WndProc
    mov   wc.cbClsExtra,NULL
    mov   wc.cbWndExtra,NULL
    push  hInstance
    pop   wc.hInstance
    mov   wc.hbrBackground,COLOR_WINDOW+1
    mov   wc.lpszMenuName,NULL
    mov   wc.lpszClassName,OFFSET ClassName
    invoke LoadIcon,NULL,IDI_APPLICATION
    mov   wc.hIcon,eax
    mov   wc.hIconSm,eax
    invoke LoadCursor,NULL,IDC_ARROW
    mov   wc.hCursor,eax
    invoke RegisterClassEx, addr wc
    invoke CreateWindowEx,NULL,\
                          ADDR ClassName,\
                          ADDR AppName,\
                          WS_OVERLAPPEDWINDOW,\
                          CW_USEDEFAULT,\
                          CW_USEDEFAULT,\
                          CW_USEDEFAULT,\
                          CW_USEDEFAULT,\
                          NULL,\
                          NULL,\
                          hInst,\
                          NULL
    mov   hwnd,eax
    invoke ShowWindow, hwnd,CmdShow
    invoke UpdateWindow, hwnd
    .WHILE TRUE
                invoke GetMessage, ADDR msg,NULL,0,0
                .BREAK .IF (!eax)
                invoke TranslateMessage, ADDR msg
                invoke DispatchMessage, ADDR msg
   .ENDW
    mov     eax,msg.wParam
    ret
WinMain endp
WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
    .IF uMsg==WM_DESTROY
        invoke PostQuitMessage,NULL
    .ELSE
        invoke DefWindowProc,hWnd,uMsg,wParam,lParam
        ret
    .ENDIF
    xor eax,eax
    ret
WndProc endp
end start
```

ممکن است متعجب شده باشید که یک برنامه ساده ویندوز به این همه برنامه‌نویسی نیاز دارد. ولی اکثر این کدها به عنوان قالبی برای تمام برنامه‌های شما هستند و می‌توانید آنها را از یک برنامه به برنامه موردنظر کپی کنید و نیاز به بازنویسی آنها برای هر برنامه ندارید.

کدهای شما تماما درون Winmain قرار دارند. این دقیقاً همان کاری است که کامپایلرهای C انجام می‌دهند. آنها تنها به شما اجازه برنامه‌نویسی برای Winmain را می‌دهند و دیگر نیازی به دستکاری بقیه موارد ندارید. در این کامپایلرها شما حتما باید تابعی به نام Winmain داشته باشید در غیر این صورت این کا مپایلرها توانایی کامپایل کردن برنامه‌های شما را ندارند. ولی در اسمبلی این محدودیت وجود ندارد. و شما می‌توانید از هر نام دیگری به جای Winmain استفاده کنید. حتی می‌توانید در بر نامه هیچ تابعی نداشته باشید.

```
.386
.model flat,stdcall
option casemap:none
WinMain proto :DWORD,:DWORD,:DWORD,:DWORD
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
```

حال به بررسی کدهای برنامه می‌پردازیم. سه سطر اول از ملزومات برنامه هستند که قبلا ذکر شده اند. دستور بعدی پیش تعریف تابع Winmain است. قبل از آنکه بتوانیم این تابع را فراخوانی کنیم باید پیش تعریف آن را معرفی کنیم. سپس می‌توانیم با دستور Invoke آن را فراخوانی کنیم. برنامه ما از توابعی که در Kernel32.dll و User32.dll هستند، استفاده می‌کند. فایل‌های مورد نیاز برای استفاده از این توابع را به برنامه خود ضمیمه می‌کنیم.

```
.DATA
    ClassName db "SimpleWinClass",0
    AppName  db "Our First Window",0
.DATA?
hInstance HINSTANCE ?
CommandLine LPSTR ?
```

این بخش مربوط به داده‌های برنامه است.

در بخش DATA. ما دو رشته مختوم به صفر (ASCIIZ) را معرفی کرده‌ایم.

Classname : که نام کلاس پنجره ما است

AppName : که نام پنجره ما است. توجه داشته باشید که هر دو آنها مقدار دهی اولیه شده‌اند.

در بخش?DATA. دو متغیر معرفی شده‌اند.

hInstance : که شماره دسترسی برنامه است.

CommandLine : دستور خط فرمان برنامه ما است.

دو نوع داده ناشناخته LPSTR و HINSTANCE در حقیقت نام‌های دیگری برای نوع داده DWORD هستند. توجه داشته باشید که متغیرهای این بخش فاقد مقدار اولیه هستند. ولی بعداً مقدار دهی خواهند شد.

```
.CODE
 start:
     invoke GetModuleHandle, NULL
     mov    hInstance,eax
     invoke GetCommandLine
     mov    CommandLine,eax
     invoke WinMain, hInstance,NULL,CommandLine, SW_SHOWDEFAULT
     invoke ExitProcess,eax
     .....
end start
```

.CODE شامل تمام دستورالعمل‌های برنامه است. کدهای برنامه باید بین دو برچسب end <StartLabel> ، <Start Label> قرار گیرند. نام برچسب اهمیتی ندارد.

اولین دستورالعمل برنامه فراخوانی تابع GetModuleHandle است که شماره دسترسی به برنامه را بر می گرداند. در Win32 ، ModuleHandle و InstanceHandle یکی هستند.

Isntance Handle : به عنوان شناسه منحصر به فرد برنامه شما است. از این شناسه به عنوان پارامتر برای تعداد زیادی از توابع API استفاده می‌شود. توجه داشته باشید که در حقیقت این شناسه آدرس خطی برنامه در حافظه کامپیوتر است.

در موقع بازگشت از یک تابع API مقدار برگشتی تابع در صورت وجود در eax قرار می گیرد. انتظار نداشته باشید که مقادیر edx ، ecx ، eax در فراخوانی های توابع API بدون تغییر بمانند. این مطلب بیان می‌کند غیر از مقدار برگشتی تابع که در eax قرار دارد باید از قانون زیر پیروی کنید:

مقادیر موجود در ثبات‌های edx, edi, esi, ebp را ذخیره و در موقع بازگشت از توابع آنها را باز گردانی کنید. در غیر این صورت برنامه شما به زودی با مشکل مواجه خواهد شد.

با فراخوانی تابع GetCommandLine می‌توانید دستورات خط فرمان را گرفته و در صورت لزوم آنها را پردازش کنید.

دستور بعدی فراخوانی Winmain است و چهار پارامتر ورودی دریافت می کند که عبارتند از: شماره دسترسی برنامه، شماره دسترسی نمونه اجرا شده قبلی برنامه (در صورت وجود)، خط فرمان و وضعیت اولیه پنجره در شروع برنامه. در Win32 نمونه قبلی از برنامه وجود ندارد و هر برنامه در حافظه مخصوص به خود تنها است. پس مقدار hPrevInst همیشه تهی است. این پارامتر از دوران Win16 به جا مانده است که در آن همه نمونه‌های اجرا شده یک برنامه از فضای یکسانی استفاده می کردند و با این پارامتر مشخص می‌شد که آیا این برنامه برای اولین بار اجرا شده است یا خیر.

توجه داشته باشید که شما نیازی به تعریف تابعی به نام Winmain ندارید و می‌توانید تمام کدی را که درون این تابع نوشته‌اید، بعد از دستور GetCommandLine قرار دهید و هیچ تغییری در برنامه ایجاد نخواهد شد. پس از بازگشت از تابع Winmain از مقدار برگشتی این تابع که درون eax قرار دارد به‌عنوان پارامتر ورودی برای تابع ExitProcess استفاده می‌کنیم که به برنامه خاتمه می‌دهد.

```
WinMain proc
Inst:HINSTANCE,hPrevInst:HINSTANCE,CmdLine:LPSTR,CmdShow:DWORD
```

خط بالا معرفی تابع Winmain است. پارامترهای ورودی بعد ازproc تعریف می‌شوند که برای دسترسی به آنها در روند اجرایی تابع می‌توان از نام معین شده استفاده کرد.

```
    LOCAL wc:WNDCLASSEX
    LOCAL msg:MSG
    LOCAL hwnd:HWND
```

دایرکتیو LOCAL حافظه‌ای از Stack را برای متغیرهای محلی تابع اختصاص می‌دهد. تعاریف LOCAL باید دقیقا بعد از تعریف تابع بیایند. نحوه استفاده از این دایرکتیو را در زیر مشاهده می‌کنید.

```
LOCAL     <نام متغیر>:<نوع متغیر>
```

برای مثال LOCAL WC:WNDCLASSEX به ماکرو اسمبلر می‌گوید که حافظه‌ای از Stack به اندازه ساختار WNDCLASSEX را برای متغیر محلی WC رزرو کند.

متغیرهای محلی فقط در داخل تابعی که در آن تعریف می‌شوند قابل استفاده هستند و بعد از اتمام کار تابع و بازگشت به تابع فراخوان از بین می‌روند. نکته دیگر این است که شما نمی‌توانید برای آنها مقدار اولیه در نظر بگیرید زیرا آنها در حقیقت حافظه‌ای از Stack هستند که به‌صورت پویا در موقع ورود به تابع رزرو می‌شوند. مقادیر موردنظر را باید به‌صورت دستی بعد از تعریف به آنها اختصاص دهید.

```
mov   wc.cbSize,SIZEOF WNDCLASSEX
mov   wc.style, CS_HREDRAW or CS_VREDRAW
mov   wc.lpfnWndProc, OFFSET WndProc
mov   wc.cbClsExtra,NULL
mov   wc.cbWndExtra,NULL
push  hInstance
pop   wc.hInstance
mov   wc.hbrBackground,COLOR_WINDOW+1
mov   wc.lpszMenuName,NULL
mov   wc.lpszClassName,OFFSET ClassName
invoke LoadIcon,NULL,IDI_APPLICATION
mov   wc.hIcon,eax
mov   wc.hIconSm,eax
invoke LoadCursor,NULL,IDC_ARROW
mov   wc.hCursor,eax
invoke RegisterClassEx, addr wc
```

خطوطی که در بالا دیدیم از نظر مفهوم بسیار ساده هستند. هدف نهایی این کدها در حقیقت ایجاد کلاس پنجره (Window Class) است.

Window Class : چیزی نیست به جز خصوصیت پنجره و نحوه کار آن. که تعداد زیادی از خصوصیات زیاد پنجره مانند آیکون، کرسر، تابع مسئول پنجره رنگ‌های فرم و... را مشخص می کند. شما پنجره را از روی کلاس پنجره ایجاد می کنید. این یکی از مفاهیم برنامه‌نویسی شیئی گرا است.

ویندوز تعداد زیادی از کلاس‌های پنجره از پیش تعیین شده دارد. (مانند دکمه، جعبه متن، جعبه پیام و ...). شما برای استفاده از این کلاس‌ها نیازی به ثبت کلاس ندارید. تنها کار لازم این است که تابع CreateWindowEx را با نام کلاس از پیش تعیین شده فراخوانی کنید.

مهمترین عضو WNDCLASSEX, LPFNWNDPROC نام دارد. که اشاره‌گری به تابع مسئول پنجره است. هر کلاس پنجره به یک تابع وابسته به خود نیاز دارد که Window Procedure نام دارد.

Window Procedure : مسئول مدیریت پیغام‌های ارسالی به تمام پنجره‌هایی است که از کلاس پنجره وابسته به آن ایجاد می‌شود. ویندوز از پیغام ها برای آگاه ساختن پنجره از رویدادهای مهمی که مسئول پاسخگویی به آنها می‌باشد استفاده می‌کند. این پیغام‌ها می‌توانند رویدادهای : ماوس، کی‌بورد و ... باشند. وظیفه برنامه تنها پاسخگویی درست و دقیق به پیغام‌ها و رویدادهای رسیده است در حقیقت شما بیشتر وقت خود را صرف نوشتن کد برای مدیریت رویدادهای رسیده می‌کنید.

در زیر اعضای WNDCLASSEX را مشاهده می‌کنید.

```
WNDCLASSEX STRUCT DWORD
  cbSize            DWORD      ?
  style             DWORD      ?
  lpfnWndProc       DWORD      ?
  cbClsExtra        DWORD      ?
  cbWndExtra        DWORD      ?
  hInstance         DWORD      ?
  hIcon             DWORD      ?
  hCursor           DWORD      ?
  hbrBackground     DWORD      ?
  lpszMenuName      DWORD      ?
  lpszClassName     DWORD      ?
  hIconSm           DWORD      ?
WNDCLASSEX ENDS
```

CbSize: سایز کلاس WNDCLASSEX بر حسب بایت. می‌توانید از عملگر SIZEOF برای دریافت سایز آن استفاده کنید

Style: سبک پنجره ایجاد شده توسط این کلاس. می‌توانید با استفاده از عملگر or چند سبک را با هم ترکیب کرده و سبک جدیدی بسازید.

LpfnWndProc: آدرس تابع مسئول برای پنجره ایجاد شده توسط این کلاس.

CbClsExtra: تعداد بایت‌های اضافی را برای ساختار کلاس معین می‌کند. سیستم عامل این بایت ها را با صفر مقدار دهی اولیه می‌کند. شما می‌توانید داده‌های خاص خود را در آن قرار دهید.

CbWndExtra: تعداد بایت‌های اضافی را برای پنجره ایجاد شده توسط این کلاس مشخص می‌کند که سیستم عامل آنها را با صفر مقدار دهی اولیه می‌کند.

HInstance: شناسه برنامه را مشخص می‌کند.

HIcon: شماره دسترسی به آیکون که توسط تابع LoadIcon تعیین می‌شود.

HCursor: شماره دسترسی به کرسر که توسط تابع LoadCursor تعیین می‌شود.

HBrBackGround: رنگ زمینه پنجره ایجاد شده توسط کلاس.

LpszMenuName: شماره دسترسی به منوی پنجره.

LpszClassName: نام کلاس پنجره.

HIconSm: شماره دسترسی به آیکون کوچک میله عنوان پنجره.

```
invoke CreateWindowEx, NULL,\
                       ADDR ClassName,\
                       ADDR AppName,\
                       WS_OVERLAPPEDWINDOW,\
                       CW_USEDEFAULT,\
                       CW_USEDEFAULT,\
                       CW_USEDEFAULT,\
                       CW_USEDEFAULT,\
                       NULL,\
                       NULL,\
                       hInst,\
                       NULL
```

بعد از اینکه کلاس پنجره را ثبت کردیم می‌توانیم تابع CreateWindowEX را برای ساخت پنجره بر اساس کلاس پنجره ارائه شده فراخوانی کنیم. توجه داشته باشید که این تابع دارای ۱۲ پارامتر ورودی است.

```
CreateWindowExA proto dwExStyle:DWORD,\
                      lpClassName:DWORD,\
                      lpWindowName:DWORD,\
                      dwStyle:DWORD,\
                      X:DWORD,\
                      Y:DWORD,\
                      nWidth:DWORD,\
                      nHeight:DWORD,\
                      hWndParent:DWORD ,\
                      hMenu:DWORD,\
                      hInstance:DWORD,\
                      lpParam:DWORD
```

DwExStyle : سبک‌های اضافی پنجره مانند topmost window را تعریف می‌کند. در اینجا می‌توانید شماره سبک موردنظر را ذکر کنید. در غیر این صورت از Null استفاده کنید.

LpClassName : آدرس یک رشته مختوم به صفر که نام کلاس پنجره را در خود دارد. در حقیقت شما از کلاس پنجره به عنوان قالبی برای تولید پنجره استفاده می‌کنید. کلاس مذکور می‌تواند کلاس ثبت شده خودتان و یا سایر کلاس‌های از پیش تعیین شده ویندوز باشد.

LpWindowName : آدرس یک رشته مختوم به صفر که نام پنجره را معین می‌کند. که در Titlebar نشان داده می‌شود.

DwStyle : سبک و مدل پنجره را تعیین می‌کند. شما می‌توانید نمای ظاهری پنجره را در اینجا مشخص کنید استفاده از Null مشکلی به وجود نمی‌آورد ولی پنجره‌ای بدون منوی سیستم و دکمه‌های Minimize , Maximize , Close خواهید داشت که برای بستن آن باید از کلیدهای Alt+F4 استفاده کنید. متداول‌ترین مدل پنجره مدل WS_OVERLAPPEDWINDOW است. شما می‌توانید سبک‌ها و مدل‌های مختلف را با اپراتور" or" با هم ترکیب کنید.

X, Y : مختصات گوشه سمت بالا پنجره در صفحه است. به‌طور معمول از مقدار CW_UseDefault استفاده می‌شود که تصمیم‌گیری در مورد مختصات پنجره را به خود سیستم عامل واگذار می کند.

NWidth , nHeight : عرض و ارتفاع پنجره را مشخص می‌کند. می‌توانید از مقدار CW_UseDefault نیز استفاده کنید.

HWndParent : شماره دسترسی به پنجره پدر را در صورت وجود معین می‌کند. این پارامتر به ویندوز می‌گوید که آیا پنجره یک پنجره فرزند (Child Window) است یا خیر. و اگر هست کدام پنجره پدر آن است.

با بسته شدن پنجره پدر تمام پنجره‌های فرزند آن نیز بسته خواهند شد. دقت داشته باشید که رابطه پدر و فرزند در اینجا با رابطه پدر و فرزند در برنامه‌های واسط چند سندی(MDI) تفاوت دارد زیرا در اینجا پنجره‌های فرزند به منطقه کاری (client area) پدر خود محدود نیستند.

hMenu : شماره دسترسی به منوی پنجره را مشخص می‌کند. در صورت استفاده از Null از منوی ذکر شده در کلاس استفاده نمی‌شود. دوباره به تعریف کلاس WNDCLASSEX نگاه کنید. در آنجا عضو lpszMenuName منوی پیش‌فرض را برای کلاس پنجره مشخص می‌کند. پنجره‌های ایجاد شده توسط این کلاس به‌طور پیش‌فرض دارای منوهای یکسانی هستند. برای تغییر این منو برای هر پنجره می‌توانید از پارامتر hMenu برای جایگزینی منوی جدید به جای منوی پیش‌فرض استفاده کنید. hMenu در حقیقت یک پارامتر دو منظوره است. اگر پنجره ای که قصد ایجاد آن را دارید از پنجره‌های از پیش تعریف شده ویندوز (مانند دکمه و جعبه متن و...) باشد نمی‌تواند دارای منو باشد. در این موارد hMenu شماره شناسایی کنترل موردنظر (Control ID) است.

HIsntance : شماره دسترسی به برنامه‌ای که پنجره را ایجاد کرده است.

LpParam : یک اشاره‌گر اختیاری به داده‌هایی است که قصد فرستادن آنها را به پنجره داریم. معمولاً توسط برنامه‌های (MDI) از آنها برای انتقال داده CLIENTCREATESTRUCT استفاده می‌شود. معمولاً به این پارامتر مقدار Null می‌دهیم که نشانگر این است که هیچ داده‌ای به پنجره فرستاده نمی‌شود. پنجره برای دریافت داده‌های ارسالی می‌تواند از تابع GetWindowLong استفاده کند.

```
mov   hwnd,eax
invoke ShowWindow, hwnd,CmdShow
invoke UpdateWindow, hwnd
```

در صورت فراخوانی موفقیت آمیز تابع CreateWindowEX شماره دسترسی به پنجره مورد نظر در eax قرار می گیرد که ما برای استفاده‌های آتی آن را ذخیره می کنیم. پنجره ای که ما ساختیم به طور خودکار نمایش داده نمی شود. برای نمایش آن باید از تابع ShowWindow استفاده کنیم. این تابع برای نشان دادن پنجره ، شماره دسترسی و وضعیت مورد نظر برای پنجره را دریافت می کند. بعد از آن می توانید UpdateWindow را برای بازکشی منطقه کاری پنجره فراخوانی کنید.

```
.WHILE TRUE
            invoke GetMessage, ADDR msg,NULL,0,0
            .BREAK .IF (!eax)
            invoke TranslateMessage, ADDR msg
            invoke DispatchMessage, ADDR msg
.ENDW
```

حال پنجره ما روی صفحه قرار دارد. ولی نمی تواند از بیرون ورودی دریافت کند. پس باید آن را وابسته به رویدادها کنیم. این کار را با یک حلقه پیغام انجام می‌دهیم. برای هر Module فقط یک حلقه پیغام وجود دارد. این حلقه به طور پیاپی توسط تابع GetMessage پیغام‌های رسیده از طرف ویندوز را دریافت می کند. تابع GetMessage یک اشاره‌گر به ساختار MSG به ویندوز می فرستد. این ساختار با اطلاعاتی که ویندوز قصد فرستادن آنها به پنجره را دارد پر می شود.

تابع GetMessage تا زمانی که پیغامی برای پنجره وجود نداشته باشد بازگشت نمی‌کند. در طول این زمان ویندوز کنترل را به سایر برنامه‌ها می‌دهد. در صورت دریافت پیغام WM_QUIT توسط تابع GetMessage این تابع مقدار برگشتی FALSE را ایجاد می‌کند که در حلقه پیغام باعث خروج از حلقه و خاتمه به روند اجرایی برنامه می‌شود.

TranslateMessage : یک تابع کمکی است که ورودی خام کی‌بورد را گرفته و یک پیغام جدید WM_CHAR را ایجاد و به صف پیغام‌های ورودی اضافه می‌کند.

پیغام WM_CHAR حاوی کد اسکی کلید فشرده شده است که بررسی آن بسیار ساده‌تر از بررسی ورودی‌های خام کی‌بورد (Scan Codes) است. در صورتی که برنامه شما از ورودی‌های کی‌بورد استفاده نمی‌کند می‌توانید این فراخوانی را نادیده بگیرید.

DispatchMessage : داده‌های پیغام را به پروسیجر پنجره که مسئول بررسی پیغام‌های رسیده است منتقل می‌کند.

```
    mov     eax,msg.wParam
    ret
WinMain endp
```

در صورت خاتمه حلقه پیغام، مقدار برگشتی آن در بخش wParam از ساختار MSG قرار می‌گیرد که می‌توانید برای فرستادن آن به ویندوز مقدارش را در ثبات eax قرار دهید. در حال حاضر ویندوز از این مقدار برگشتی استفاده‌ای نمی‌کند ولی بهتر است با اطمینان کار کنید و به قوانین پایبند باشید.

```
WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
```

این پروسیجر پنجره است. پارامتر اول شماره دسترسی به پنجره است که پیغام‌ها برای آن فرستاده می‌شوند. توجه داشته باشید که UMSG از ساختار MSG نیست بلکه دقیقاً یک عدد است. ویندوز دارای هزاران پیغام تعریف شده است که اکثر آنها برای برنامه شما دارای اهمیت نیستند. ویندوز در صورتی پیغام اختصاصی به یک پنجره می‌فرستد که رویدادی مربوط به آن پنجره اتفاق افتاده باشد. پروسیجر پنجره پیغام را دریافت کرده و با دقت نسبت به آن واکنش نشان می‌دهد.

LParam و WParam پارامتر های اضافی هستند که برای برخی از پیغام‌ها استفاده می‌شوند.

```
    .IF uMsg==WM_DESTROY
        invoke PostQuitMessage,NULL
    .ELSE
        invoke DefWindowProc,hWnd,uMsg,wParam,lParam
        ret
    .ENDIF
    xor eax,eax
    ret
WndProc endp
```

در اینجا اصلی‌ترین بخش برنامه که همان پاسخ‌گویی به پیغام‌ها است قرار دارد. برنامه ابتدا پیغام‌های رسیده را چک کرده و در صورتی که پیغام برای برنامه اهمیت داشته باشد به آن پاسخ می‌دهد و در آخر هم با مقدار برگشتی صفر بازگشت می‌کند. در صورتی که پیغام برای برنامه اهمیتی نداشته باشد باید با فراخوانی تابع DefWindowProc و فرستادن کلیه اطلاعات پیغام به آن، پاسخگویی به پیغام را به ویندوز واگذار کنیم و بگذاریم روند معمولی پاسخگویی انجام شود. در این برنامه تنها پیغامی که شما باید به آن پاسخ دهید پیغام WM_DESTROY است. این پیغام در صورت بسته شدن پنجره فرستاده می‌شود. زمانی که این پیغام را دریافت می‌کنید پنجره شما دیگر روی صفحه وجود ندارد. در حقیقت این پیغام فقط برای اطلاع شما است که خود را برای بازگشت به ویندوز آماده کنید.

در صورتی که می‌خواهید از بسته شدن پنجره جلوگیری کنید باید از پیغام WM_CLOSE استفاده کنید. پس از دریافت WM_DESTROY و آمادگی برای خارج شدن از برنامه با فراخوانی تابع PostQuitMessage پیغام WM_QUIT به برنامه شما فرستاده می‌شود. این پیغام باعث می‌شود که تابع GetMessage با مقدار صفر بازگشت کند که باعث خاتمه حلقه پیغام و بازگشت به ویندوز می‌شود. شما می‌توانید با فراخوانی تابع DestroyWindow پیغام WM_DESTROY را به پروسیجر پنجره خود بفرستید.

نمایش متن

کدها و فایل‌های مربوط به این قسمت در CD ضمیمه موجود می‌باشد

SourceCodes\Asm32\Chapter4

متن در ویندوز یکی از اشیای گرافیکی است که جزئی از رابط کاربر ویندوز محسوب می‌شود. هر کاراکتر از تعدادی پیکسل تشکیل شده است که با الگویی مشخص کنار هم قرار گرفته‌اند به این علت است که معمولاً به جای نوشتن متن از واژه کشیدن متن استفاده می‌کنیم. و آن را در محل مورد نظر ترسیم می‌کنیم.

نمایش متن روی صفحه در Windows و Dos کاملا متفاوت است. در Dos شما به ابعاد ۸۰ * ۲۵ فکر می‌کردید. اما در ویندوز صفحه بین برنامه‌های مختلف تقسیم می‌شود. پس باید قوانینی اعمال شود که از نوشتن برنامه‌ها روی پنجره‌های یکدیگر جلوگیری کند. در ویندوز اندازه پنجره‌های برنامه‌های مختلف روی صفحه ثابت نمی‌باشد و کاربر یا خود برنامه می‌تواند آنها را تغییر دهد. پس شما باید ابعاد پنجره‌های خود را در صورت تغییر پذیر بودن آنها، به‌طور پویا تشخیص دهید. قبل از اینکه بخواهید چیزی را در منطقه کاری پنجره خود بکشید باید از ویندوز کسب اجازه کنید. به این دلیل است که شما مانند Dos کنترل مطلق روی صفحه نمایش ندارید.

برای کشیدن متن در بخش مربوط به برنامه خود روی صفحه ابتدا از ویندوز کسب اجازه می‌کنید. سپس ویندوز فونت، رنگ، اندازه و دیگر مشخصات رابط کاربر را تشخیص داده و یک شماره دسترسی به ابزار متن را به برنامه شما بر می‌گرداند. از این شماره که در حقیقت مجوز استفاده از ابزار متن است، برای چاپ متن در منطقه اختصاصی برنامه روی صفحه استفاده می‌شود.

ابزار متن در حقیقت یک ساختمان داده است که در داخل ویندوز نگهداری می‌شود. این ابزار به دستگاهی مانند پرینتر یا صفحه نمایش وابسته می‌شود. در مورد صفحه نمایش ابزار متن به پنجره در حال نمایش وابسته می‌شود. برخی از مشخصات ابزار متن مشخصات گرافیکی مثل رنگ و فونت می‌باشند که دارای مقادیر پیش‌فرض هستند ولی می‌توان این مقادیر را تغییر داد. این مقادیر پیش‌فرض باعث عدم لزوم مقداردهی این مشخصات برای هر فراخوانی تابع می‌شود.

روش‌های گوناگونی برای بدست آوردن شماره دسترسی به ابزار گرافیکی وجود دارد:

۱- فراخوانی تابع BeginPaint در پاسخ به پیغام WM_PAINT .

۲- فراخوانی تابع GetDC در مواقع موردنظر .

۳- فراخوانی تابع CreateDC برای بدست آوردن شماره دسترسی به ابزار جدید. در حقیقت از تابع CreateDC برای ایجاد یک DC (Device Content) جدید استفاده می‌شود.

ویندوز پیغام WM_PAINT را به یک پنجره می فرستد تا آن را از باز کشی خود با خبر کند. ویندوز محتوای گرافیکی منطقه کاری پنجره را ذخیره نمی‌کند به جای آن وقتی تغییر وضعیتی اتفاق می‌افتد بازسازی منطقه کاری پنجره‌ها را تضمین می‌کند. ویندوز پیغام WM_PAINT را در صف پیغام‌های رسیده به پنجره قرار می‌دهد و این مسئولیت پنجره است که در پاسخ به این پیغام منطقه کاری خود را باز کشی کند. شما باید تمام اطلاعات در مورد چگونگی باز کشی منطقه کاری را در زیر برنامه WM_PAINT جمع‌آوری کنید. بعدا" وقتی پنجره پیغام WM_PAINT را دریافت کرد می‌تواند منطقه کاری خود را بازکشی کند.

یکی از مواردی که باید با آن آشنایی داشته باشید مستطیل نامعتبر می‌باشد. مستطیل نامعتبر به عنوان کوچکترین مستطیل از منطقه کاری پنجره است که نیاز به بازکشی دارد. وقتی ویندوز از وجود یک مستطیل نامعتبر در منطقه کاری یک پنجره مطلع شود به سرعت پیغام WM_PAINT را به آن می‌فرستد. به همراه این پیغام ویندوز ساختار Paintstruct را که حاوی مختصات مستطیل نامعتبر می‌باشد را ارسال می‌کند. شما در پاسخ به پیغام WM_PAINT برای معتبر ساختن یک مستطیل نامعتبر تابع BeginPaint را فراخوانی می‌کنید.

اگر پیغام WM_PAINT را پردازش نمی‌کنید حداقل باید تابع DefWindowproc یا ValidateRect را برای معتبر ساختن مستطیل نامعتبر فراخوانی کنید. در غیر این‌صورت ویندوز مرتبا" پیغام WM_PAINT را برای پنجره می فرستد .

در زیر مراحلی که شما باید برای پاسخ به یک پیغام WM_PAINT اجرا کنید آمده است.

۱- گرفتن شماره دسترسی به ابزار گرافیکی با فراخوانی تابع BeginPaint.

۲- باز کشی منطقه کاری پنجره .

۳- رها کردن ابزار گرافیکی و بازگردانی کنترل آن به ویندوز .

شما مجبور نیستید که به‌طور کامل تمام مستطیل‌های بی‌اعتبار را معتبر سازید. این کار به‌طور اتوماتیک توسط تابع BeginPaint انجام می‌شود. ما بین فراخوانی این تابع و تابع EndPaint می‌توانید هر کدام از توابع گرافیکی ویندوز را برای کشیدن منطقه کاری پنجره صدا بزنید. اکثر این توابع از شماره دسترسی به ابزار به عنوان پارامتر ورودی استفاده می‌کنند.

حال برای مثال برنامه‌ای می‌نویسیم که یک رشته متن را در وسط منطقه کاری پنجره خود نمایش دهد.

```
.386
.model flat,stdcall
option casemap:none
WinMain proto :DWORD,:DWORD,:DWORD,:DWORD
include \masm32\include\windows.inc
include \masm32\include\user32.inc
includelib \masm32\lib\user32.lib
include \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib
.DATA
ClassName db "SimpleWinClass",0
AppName  db "Our First Window",0
OurText  db "Win32 assembly is great and easy!",0
.DATA?
hInstance HINSTANCE ?
CommandLine LPSTR ?
.CODE
start:
    invoke GetModuleHandle, NULL
    mov    hInstance,eax
    invoke GetCommandLine
    mov CommandLine,eax
    invoke WinMain, hInstance,NULL,CommandLine, SW_SHOWDEFAULT
    invoke ExitProcess,eax
WinMain proc hInst:HINSTANCE,\
hPrevInst:HINSTANCE,\
CmdLine:LPSTR,\
CmdShow:DWORD
    LOCAL wc:WNDCLASSEX
    LOCAL msg:MSG
    LOCAL hwnd:HWND
    mov   wc.cbSize,SIZEOF WNDCLASSEX
    mov   wc.style, CS_HREDRAW or CS_VREDRAW
    mov   wc.lpfnWndProc, OFFSET WndProc
    mov   wc.cbClsExtra,NULL
    mov   wc.cbWndExtra,NULL
    push  hInst
    pop   wc.hInstance
    mov   wc.hbrBackground,COLOR_WINDOW+1
    mov   wc.lpszMenuName,NULL
    mov   wc.lpszClassName,OFFSET ClassName
    invoke LoadIcon,NULL,IDI_APPLICATION
    mov   wc.hIcon,eax
    mov   wc.hIconSm,eax
    invoke LoadCursor,NULL,IDC_ARROW
    mov   wc.hCursor,eax
    invoke RegisterClassEx, addr wc
    invoke CreateWindowEx, NULL,\
                           ADDR ClassName,\
                           ADDR AppName,\
                           WS_OVERLAPPEDWINDOW,\
```

```
                            CW_USEDEFAULT,\
                            CW_USEDEFAULT,\
                            CW_USEDEFAULT,\
                            CW_USEDEFAULT,\
                            NULL,\
                            NULL,\
                            hInst,\
                            NULL
    mov   hwnd,eax
    invoke ShowWindow, hwnd,SW_SHOWNORMAL
    invoke UpdateWindow, hwnd
        .WHILE TRUE
                invoke GetMessage, ADDR msg,NULL,0,0
                .BREAK .IF (!eax)
                invoke TranslateMessage, ADDR msg
                invoke DispatchMessage, ADDR msg
        .ENDW
        mov     eax,msg.wParam
        ret
WinMain endp
WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
    LOCAL hdc:HDC
    LOCAL ps:PAINTSTRUCT
    LOCAL rect:RECT
    .IF uMsg==WM_DESTROY
        invoke PostQuitMessage,NULL
    .ELSEIF uMsg==WM_PAINT
        invoke BeginPaint, hWnd, ADDR ps
        mov    hdc,eax
        invoke GetClientRect, hWnd, ADDR rect
        invoke DrawText, hdc,\
                         ADDR OurText,\
                         -1,\
                         ADDR rect, \
                         DT_SINGLELINE or DT_CENTER or DT_VCENTER
        invoke EndPaint,hWnd, ADDR ps
    .ELSE
        invoke DefWindowProc, hWnd, uMsg, wParam, lParam
        ret
    .ENDIF
    xor   eax, eax
    ret
WndProc endp
end start
```

اکثر کدهای این برنامه همان کدهای مثال قبل هستند. ما در اینجا تغییرات مهم را توضیح می‌دهیم:

```
    LOCAL hdc:HDC
    LOCAL ps:PAINTSTRUCT
    LOCAL rect:RECT
```

اینها متغیرهای محلی هستند که توسط توابع گرافیکی API در بخش WM_PAINT استفاده می‌شوند. از hdc برای ذخیره شماره دسترسی به ابزار استفاده می‌شود که مقدار آن توسط تابع BeginPaint تعیین می‌شود. Ps یک ساختار PaintStruct می‌باشد که آیتم‌های آن توسط تابع BeginPaint مقدار دهی می‌شوند. در حقیقت این ساختار به تابع BeginPaint فرستاده می‌شود و ویندوز آنرا با مقادیر مناسب مقداردهی می‌کند. وقتی که کشیدن منطقه کاری پنجره به پایان رسید تابع EndPaint را فراخوانی کرده و Ps را به عنوان پارامتر ورودی به آن می‌فرستیم. Rect از نوع ساختار RECT می‌باشد که تعریف آنرا در زیر مشاهده می‌کنید.

```
RECT Struct
    left      LONG ?
    top       LONG ?
    right     LONG ?
    bottom    LONG ?
RECT ends
```

Left , Top : مختصات گوشه چپ بالای مستطیل می‌باشند.

Right , Bottom : مختصات گوشه راست پایین مستطیل می‌باشد.

```
invoke BeginPaint,hWnd, ADDR ps
mov    hdc,eax
invoke GetClientRect,hWnd, ADDR rect
invoke DrawText, hdc,\
                 ADDR OurText,\
                 -1,\
                 ADDR rect, \
                 DT_SINGLELINE or DT_CENTER or DT_VCENTER
invoke EndPaint,hWnd, ADDR ps
```

در پاسخ به پیغام WM_PAINT تابع BeginPaint را فراخوانی می‌کنید. بعد از فراخوانی موفقیت آمیز این تابع شماره دسترسی به ابزار در eax قرار می‌گیرد. سپس شما تابع GetClientRect را برای گرفتن ابعاد منطقه کاری پنجره فراخوانی می‌کنید. این ابعاد در یک ساختار RECT به شما بازگردانی می‌شود که از آن به عنوان پارامتر ورودی برای تابع DrawText استفاده می‌شود.

در زیر تعریف تابع DrawText را مشاهده می‌کنید.

```
DrawText proto hdc:HDC,\
lpString:DWORD,\
nCount:DWORD,\
lpRect:DWORD,\
```

```
uFormat:DWORD
```

DrawText یک تابع API سطح بالا برای چاپ متن روی صفحه است. این تابع برخی جزئیات چاپ متن را به‌صورت خودکار انجام می‌دهد. برای چاپ متن با جزئیات بیشتر می‌توانید از تابع سطح پایین TextOut استفاده کنید. تابع DrawText شکل متن موردنظر را برای قرار گرفتن در یک مستطیل مشخص که ابعاد آن در یک ساختار RECT به این تابع فرستاده می‌شود آماده می‌کند. حال پارامترهای ورودی این تابع را توضیح می‌دهیم.

Hdc : شماره دسترسی به ابزار.

LpString : یک اشاره‌گر به رشته‌ای که می‌خواهیم آنرا چاپ کنیم. رشته باید مختوم به صفر باشد در غیر این صورت باید تعداد کاراکتر را در متغیر بعدی یعنی nCount مشخص کنیم.

nCount : تعداد کاراکترهای خروجی را معین می‌کند. اگر رشته معین شده یک رشته مختوم به صفر باشد به این پارامتر مقدار ۱- می‌دهیم در غیر این صورت nCount باید تعداد کاراکترهای خروجی را مشخص کند.

lpRect : یک اشاره‌گر به مستطیلی که می‌خواهید متن را در آن چاپ کنید. این مستطیل یک منطقه مرزی را مشخص می‌کند که نمی‌توانید در خارج از آن متنی را چاپ کنید.

uFormat : مشخص می‌کند که متن چگونه در مستطیل نمایش داده شود. ما در اینجا از سه حالت که با عملگر or با هم ترکیب شده اند استفاده می‌کنیم.

- **Dt_singleline**: یک خط از متن را مشخص می‌کند.
- **Dt_center**: متن را از نظر افقی در مرکز قرار می‌دهد.
- **Dt_vcenter**: متن را از نظر عمودی در مرکز قرار می‌دهد.

پس از پایان یافتن کشیدن منطقه کاری باید تابع EndPaint را برای رها کردن ابزار گرافیکی فراخوانی کنید.

حال به طور خلاصه نکات برجسته این بخش را ذکر می‌کنیم.

- در پاسخ به پیغام WM_PAINT دو تابع EndPaint و BeginPaint را فراخوانی می‌کنید.
- هر کاری که می‌خواهید با منطقه کاری پنجره انجام دهید بین این دو فراخوانی انجام دهید.

اگر می‌خواهید منطقه کاری را در پاسخ پیغام دیگری بازکشی کنید دو راه وجود دارد:

۱- توابع GetDC , ReleaseDC را فراخوانی کرده و کارهای مورد نظر را بین این دو فراخوانی انجام دهید.

۲- توابع InvalidateRect و UpdateWindow را برای باطل کردن کل منطقه کاری صدا می‌زنیم. با این کار ویندوز را مجبور می‌کنیم که پیغام WM_PAINT را در صف پیغام‌های پنجره قرار دهد.

ورودی Keyboard

کدها و فایل‌های مربوط به این قسمت در CD ضمیمه موجود می‌باشد

SourceCodes\Asm32\Chapter5

معمولاً برای هرPC یک کی بورد وجود دارد که تمام برنامه‌های در حال اجرا در ویندوز باید آن را بین خود تقسیم کنند. ویندوز مسئولیت فرستادن ضربات کی‌بورد به پنجره فعال را برعهده دارد. اگر چه پنجره‌های زیادی ممکن است روی صفحه باشند ولی فقط یکی از آنها فوکوس ورودی را دارد که به آن پنجره فعال یا پنجره دارای فوکوس گفته می‌شود. شما می‌توانید پنجره فعال را با نگاه کردن به میله عنوان (Title Bar) تشخیص دهید.

در واقع بر حسب دید شما نسبت به کی بورد دو نوع پیغام وجود دارد. می‌توانید به کی بورد به دید مجموعه ای از کلیدها نگاه کنید. در این زمینه اگر شما یک کلید را فشار دهید ، ویندوز پیغام WM_KEYDOWN را به پنجره فعال می‌فرستد و مشخص می‌کند که کلید فشار داده شده است. وقتی شما کلید را رها کردید ویندوز پیغام WM_KEYUP را به پنجره می‌فرستد. با این دید شما با یک کلید مانند یک دکمه رفتار می‌کنید.

می‌توانید به کی‌بورد به عنوان مجموعه‌ای از کاراکترها نگاه کنید. برای مثال وقتی شما کلید a را فشار می‌دهید ویندوز پیغام WM_CHAR را به پنجره فعال می‌فرستد و برای برنامه معین می‌کند که کاربر کاراکتر a را به آن فرستاده است. در واقع ویندوز پیغام‌های , WM_KEYDOWN WM_KEYUP را به پنجره می‌فرستد که توسط تابع TranslateMessage به WM_CHAR ترجمه می‌شوند. در صورت نیاز می‌توانید پیغام‌های WM_KEYUP ,WM_KEYDOWNرا پردازش کنید یا اینکه آنها را به WM_CHAR ترجمه کنید در این مورد حق انتخاب با شما است.

حال که آشنایی نسبی با کی‌بورد و نحوه مدیریت آن پیدا کردید، برای روشن شدن بهتر مطلب به مثالی در این زمینه توجه کنید:

```
.386
.model flat,stdcall
option casemap:none
WinMain proto :DWORD,:DWORD,:DWORD,:DWORD
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
include \masm32\include\gdi32.inc
```

```
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\gdi32.lib
.data
ClassName db "SimpleWinClass",0
AppName  db "Our First Window",0
char WPARAM 20h
.data?
hInstance HINSTANCE ?
CommandLine LPSTR ?
.code
start:
    invoke GetModuleHandle, NULL
    mov    hInstance,eax
    invoke GetCommandLine
    mov CommandLine,eax
    invoke WinMain, hInstance,NULL,CommandLine, SW_SHOWDEFAULT
    invoke ExitProcess,eax
WinMain proc
hInst:HINSTANCE,hPrevInst:HINSTANCE,CmdLine:LPSTR,CmdShow:DWORD
    LOCAL wc:WNDCLASSEX
    LOCAL msg:MSG
    LOCAL hwnd:HWND
    mov   wc.cbSize,SIZEOF WNDCLASSEX
    mov   wc.style, CS_HREDRAW or CS_VREDRAW
    mov   wc.lpfnWndProc, OFFSET WndProc
    mov   wc.cbClsExtra,NULL
    mov   wc.cbWndExtra,NULL
    push  hInst
    pop   wc.hInstance
    mov   wc.hbrBackground,COLOR_WINDOW+1
    mov   wc.lpszMenuName,NULL
    mov   wc.lpszClassName,OFFSET ClassName
    invoke LoadIcon,NULL,IDI_APPLICATION
    mov   wc.hIcon,eax
    mov   wc.hIconSm,eax
    invoke LoadCursor,NULL,IDC_ARROW
    mov   wc.hCursor,eax
    invoke RegisterClassEx, addr wc
    invoke CreateWindowEx, NULL,\
                           ADDR ClassName,\
                           ADDR AppName,\
                           WS_OVERLAPPEDWINDOW,\
                           CW_USEDEFAULT,\
                           CW_USEDEFAULT,\
                           CW_USEDEFAULT,\
                           CW_USEDEFAULT,\
                           NULL,\
                           NULL,\
                           hInst,\
                           NULL
```

```
    mov   hwnd,eax
    invoke ShowWindow, hwnd,SW_SHOWNORMAL
    invoke UpdateWindow, hwnd
    .WHILE TRUE
                invoke GetMessage, ADDR msg,NULL,0,0
                .BREAK .IF (!eax)
                invoke TranslateMessage, ADDR msg
                invoke DispatchMessage, ADDR msg
        .ENDW
    mov     eax,msg.wParam
    ret
WinMain endp
WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
    LOCAL hdc:HDC
    LOCAL ps:PAINTSTRUCT
    .IF uMsg==WM_DESTROY
        invoke PostQuitMessage,NULL
    .ELSEIF uMsg==WM_CHAR
        push wParam
        pop  char
        invoke InvalidateRect, hWnd,NULL,TRUE
    .ELSEIF uMsg==WM_PAINT
        invoke BeginPaint,hWnd, ADDR ps
        mov    hdc,eax
        invoke TextOut,hdc,0,0,ADDR char,1
        invoke EndPaint,hWnd, ADDR ps
    .ELSE
        invoke DefWindowProc,hWnd,uMsg,wParam,lParam
        ret
    .ENDIF
    xor    eax,eax
    ret
WndProc endp
end start
```

حال به بررسی این برنامه می‌پردازیم.

```
char WPARAM 20h
```

در این بخش متغیری را برای ذخیره کاراکترهای دریافت شده تعریف می‌کنیم. این متغیر را با مقدار 20 که همان کاراکتر space است مقدار دهی اولیه می‌کنیم. زمانی که پنجره ما برای اولین بار منطقه کاری خود را بازکشی می‌کند، هیچ کاراکتر ورودی وجود ندارد به همین دلیل ما کاراکتر space که فضای خالی است را به جای آن نشان می‌دهیم.

```
.ELSEIF uMsg==WM_CHAR
        push wParam
        pop  char
        invoke InvalidateRect, hWnd,NULL,TRUE
```

این بخش برای مدیریت و واکنش نسبت به پیغام WM_CHAR به پروسیجر پنجره اضافه می شود. و در حقیقت کاراکتر ها را از ورودی گرفته و در متغیر char قرار می‌دهد. سپس تابع InvalidateRect را برای بازکشی منطقه کاری فراخوانی می‌کند. این تابع محدوده معینی از منطقه کاری پنجره را بی‌اعتبار ساخته و باعث ارسال پیغام WM_PAINT از طرف ویندوز به زیر برنامه پنجره می‌شود.

در زیر پیش تعریف تابع InvalidateRect را مشاهده می‌کنید.

```
InvalidateRect proto hWnd:HWND,\
                     lpRect:DWORD,\
                     bErase:DWORD
```

LpRect : یک اشاره‌گر به مستطیلی از منطقه کاری است که می‌خواهیم آن را بی‌اعتبار کنیم. در صورت استفاده از Null کل منطقه کاری پنجره بی‌اعتبار می‌شود.

bErase : یک flag است که به ویندوز می‌گوید آیا نیاز به پاک کردن پشت زمینه هست یا نه. اگر این متغیر True باشد ویندوز پشت زمینه مستطیل نا معتبر را در هنگام فراخوانی BeginPaint پاک می‌کند.

استراتژی که ما در اینجا استفاده می‌کنیم این است که تمام اطلاعات مورد نیاز برای کشیدن منطقه کاری را ذخیره کرده و سپس از پیغام WM_PAINT برای کشیدن آن استفاده می‌کنیم. می‌توانید منطقه کاری را در طول پردازش WM_CHAR با فراخوانی توابع ReleaseDC و GetDC بکشید و در حقیقت کدهای مربوط به کشیدن منطقه کاری را در بخش WM_CHAR قرار دهید. ولی مشکل از اینجا شروع می‌شود که زمانیکه پنجره ما نیاز به بازکشی دارد، کدهایی که مربوط به بازکشی منطقه کاربر هستند در قسمت WM_CHAR قرار گرفته‌اند و پروسیجر پنجره نمی‌تواند منطقه کاری را بازکشی کند. پس روش بهتر این است که اطلاعات و کدهای مربوط به بازکشی منطقه کاری پنجره را در بخش WM_CHAR قرار دهیم. در صورت نیاز می‌توانید با فراخوانی تابع InvalidateRect پیغام WM_PAINT را به پروسیجر پنجره خود بفرستید.

```
invoke TextOut,hdc,0,0,ADDR char,1
```

وقتی که تابع InvalidateRect صدا زده می‌شود پیغام WM_PAINT را به زیر برنامه پنجره می‌فرستد. در نتیجه دستورات قسمت WM_PAINT اجرا می‌شوند طبق معمول ابتدا تابع BeginPaint برای گرفتن شماره دسترسی به ابزار متن فرا خوانده شده و سپس تابع TextOut برای کشیدن متن در منطقه کاری در مختصات (0,0) صدا زده می‌شود.

حال وقتی شما برنامه را اجرا می کنید و کلیدی را فشار می‌دهید می‌توانید کاراکتر مورد نظر را در گوشه سمت چپ بالای پنجره کاربر ببینید و وقتی پنجره حداقل و حداکثر می‌شود، تغییری در شکل ظاهری و محتویات پنجره ظاهر نمی‌شود زیرا تمام اطلاعات مورد نیاز برای بازکشی منطقه کاری را در بخش WM_PAINT قرار داده‌ایم.

ورودی Mouse

کدها و فایل‌های مربوط به این قسمت در CD ضمیمه موجود می‌باشد

SourceCodes\Asm32\Chapter6

در این بخش یاد می‌گیریم که چگونه ورودی ماوس را دریافت و در زیر برنامه پنجره به آن پاسخ بدهیم. مثال این بخش برای دریافت کلیک چپ ماوس صبر می‌کند و در نقطه‌ای که کلیک انجام می‌شود یک رشته متن را چاپ می‌کند.

مانند ورودی کی بورد ویندوز اطلاعات و مشخصات فعالیت‌های ماوس را تشخیص داده و به پنجره‌های مربوطه می فرستد. این فعالیت‌ها شامل کلیک های چپ , راست ، حرکت ماوس روی پنجره و double-click می باشد. بر خلاف پیغام‌های کی بورد که مستقیماً به پنجره فعال فرستاده می شوند. پیغام های ماوس به پنجره ای که اشاره گر ماوس روی آن باشد فرستاده می شوند چه آن پنجره فعال باشد یا نباشد. اگر چه پیغام هایی برای مناطق غیر کاری از قبیل نوار عنوان ، دکمه های حداقل و حداکثر پنجره و ... نیز وجود دارند ولی در اکثر مواقع می توانید آنها را نادیده گرفته و روی موارد مربوط به منطقه کاری متمرکز شوید.

برای هر کلیک ماوس دو پیغام وجود دارد که عبارتند از:

```
WM_LBUTTONDOWN و WM_RBUTTONDOWN و WM_LBUTTONUP و WM_RBUTTONUP
```

برای ماوس‌های سه کلیدی پیغام‌های WM_MBUTTONDOWN و WM_MBUTTONUP نیز وجود دارند. وقتی ماوس روی منطقه کاری حرکت می‌کند ویندوز پیغام WM_MOUSEMOVE را به پروسیجر پنجره‌ای که زیر اشاره‌گر آن قرار دارد می‌فرستد. اگر کلاس پنجره فلگ CS_DBCLICK را داشته باشد می‌تواند پیغام‌های WM_LBUTTONDBCLK و WM_RBUTTONDBCLK را نیز دریافت کند. در غیر این صورت پنجره فقط می تواند پیغام های Mouse Up , Mouse Down را دریافت کند. در تمام این پیغام ها lParam حاوی موقعیت ماوس خواهد بود. Low word مختص X وHigh word مختص Y را مشخص می کند که مختصات گوشه سمت چپ بالای منطقه کاری پنجره هستند.

wParam حالت کلیدهای ماوس و کلیدهای Shift و Ctrl را معین می کند که از آن می توان برای کنترل کلید های ترکیبی استفاده کرد.

```
.386
.model flat,stdcall
option casemap:none
WinMain proto :DWORD,:DWORD,:DWORD,:DWORD
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
include \masm32\include\gdi32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\gdi32.lib
.data
ClassName db "SimpleWinClass",0
AppName  db "Our First Window",0
MouseClick db 0         ; 0=no click yet
.data?
hInstance HINSTANCE ?
CommandLine LPSTR ?
hitpoint POINT <>
.code
start:
    invoke GetModuleHandle, NULL
    mov    hInstance,eax
    invoke GetCommandLine
    mov CommandLine,eax
    invoke WinMain, hInstance,NULL,CommandLine, SW_SHOWDEFAULT
    invoke ExitProcess,eax
WinMain proc
hInst:HINSTANCE,hPrevInst:HINSTANCE,CmdLine:LPSTR,CmdShow:DWORD
    LOCAL wc:WNDCLASSEX
    LOCAL msg:MSG
    LOCAL hwnd:HWND
    mov   wc.cbSize,SIZEOF WNDCLASSEX
    mov   wc.style, CS_HREDRAW or CS_VREDRAW
    mov   wc.lpfnWndProc, OFFSET WndProc
    mov   wc.cbClsExtra,NULL
    mov   wc.cbWndExtra,NULL
    push  hInst
    pop   wc.hInstance
    mov   wc.hbrBackground,COLOR_WINDOW+1
    mov   wc.lpszMenuName,NULL
    mov   wc.lpszClassName,OFFSET ClassName
    invoke LoadIcon,NULL,IDI_APPLICATION
    mov   wc.hIcon,eax
    mov   wc.hIconSm,eax
    invoke LoadCursor,NULL,IDC_ARROW
    mov   wc.hCursor,eax
    invoke RegisterClassEx, addr wc
    invoke CreateWindowEx, NULL,\
                           ADDR ClassName,\
```

```
                            ADDR AppName,\
                            WS_OVERLAPPEDWINDOW,\
                            CW_USEDEFAULT,\
                            CW_USEDEFAULT,\
                            CW_USEDEFAULT,\
                            CW_USEDEFAULT,\
                            NULL,\
                            NULL,\
                            hInst,\
                            NULL
    mov   hwnd,eax
    invoke ShowWindow, hwnd,SW_SHOWNORMAL
    invoke UpdateWindow, hwnd
    .WHILE TRUE
                invoke GetMessage, ADDR msg,NULL,0,0
                .BREAK .IF (!eax)
                invoke DispatchMessage, ADDR msg
    .ENDW
    mov     eax,msg.wParam
    ret
WinMain endp
WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
    LOCAL hdc:HDC
    LOCAL ps:PAINTSTRUCT
    .IF uMsg==WM_DESTROY
        invoke PostQuitMessage,NULL
    .ELSEIF uMsg==WM_LBUTTONDOWN
        mov eax,lParam
        and eax,0FFFFh
        mov hitpoint.x,eax
        mov eax,lParam
        shr eax,16
        mov hitpoint.y,eax
        mov MouseClick,TRUE
        invoke InvalidateRect,hWnd,NULL,TRUE
    .ELSEIF uMsg==WM_PAINT
        invoke BeginPaint,hWnd, ADDR ps
        mov    hdc,eax
        .IF MouseClick
            invoke lstrlen,ADDR AppName
            invoke TextOut,hdc,\
                           hitpoint.x,\
                           hitpoint.y,\
                           ADDR AppName,\
                           eax
        .ENDIF
        invoke EndPaint,hWnd, ADDR ps
    .ELSE
        invoke DefWindowProc,hWnd,uMsg,wParam,lParam
        ret
    .ENDIF
    xor    eax,eax
```

```
    ret
WndProc endp
end start
    .ELSEIF uMsg==WM_LBUTTONDOWN
        mov eax,lParam
        and eax,0FFFFh
        mov hitpoint.x,eax
        mov eax,lParam
        shr eax,16
        mov hitpoint.y,eax
        mov MouseClick,TRUE
        invoke InvalidateRect,hWnd,NULL,TRUE
```

زیر برنامه پنجره منتظر کلیک چپ ماوس می‌ماند و وقتی پیغام WM_LBUTTONDOWN را دریافت کرد lParam حاوی مختصات اشاره گر ماوس در منطقه کاری خواهد بود وآن را در متغیری از نوع POINT ذخیره خواهیم کرد. که تعریف آن به صورت زیر است:

```
POINT STRUCT
    x   dd ?
    y   dd ?
POINT ENDS
```

پس از دریافت این پیغام زیر برنامه فلگ مربوط به کلیک ماوس را True خواهد کرد یعنی حداقل یک کلیک چپ ماوس در منطقه کاری اتفاق افتاده است.

```
mov eax,lParam
and eax,0FFFFh
mov hitpoint.x,eax
```

به خاطر این که مختصات X در Low word متغیر lParam قرار دارد و اعضای ساختار POINT , ۳۲ بیتی هستند باید قسمت High word از ثبات eax را صفر کنیم تا بتوانیم آن را در hitpoint.x ذخیره کنیم.

```
shr eax,16
mov hitpoint.y,eax
```

به علت این که مختص Y در Low word متغیر lParam قرار دارد، برای ذخیره کردن آن در hitpoint.y باید آن را در قسمت Low word ثبات eax قرار دهیم. این کار را با چرخش eax ۱٦ بیت به راست انجام می‌دهیم. به خاطر این که به بخش WM_PAINT نشان دهیم کلیکی در منطقه کاری روی داده است و می‌تواند رشته متن را در موقعیت ماوس بکشد، پس از ذخیره موقعیت

ماوس فلگ MouseClick را True می‌کنیم و بعد تابع InvalidateRect را صدا می‌زنیم که پنجره را مجبور به بازکشی منطقه کاری می‌کند.

```
.IF MouseClick
   invoke lstrlen,ADDR AppName
   invoke TextOut,hdc,hitpoint.x,hitpoint.y,ADDR AppName,eax
.ENDIF
```

کد قسمت WM_PAINT باید چک کند که آیا کلیکی اتفاق افتاده است یا نه. وقتی که پنجره ایجاد می شود یک پیغام WM_PAINT دریافت می‌کند در حالی که هیچ کلیکی اتفاق نیفتاده است پس نباید چیزی در منطقه کاری چاپ شود. ما مقدار اولیه MouseClick را False قرار می‌دهیم و هنگامی که کلیک واقعی ماوس اتفاق افتاد آن را به True تغییر می‌دهیم. اگر حداقل یک کلیک ماوس اتفاق افتد در موقعیت ماوس رشته متن چاپ خواهد شد. توجه داشته باشید که تابع lstrlen را برای گرفتن طول رشته مورد نمایش فراخوانی کرده و آنرا به عنوان آخرین پارامتر به تابع Textout می‌فرستیم.

منو

کدها و فایلهای مربوط به این قسمت در CD ضمیمه موجود می‌باشد

SourceCodes\Asm32\Chapter7

منو یکی از مهمترین اجزای پنجرها محسوب می‌شود زیرا درحقیقت لیستی از خدماتی را که برنامه شما انجام می‌دهد به کاربر نشان می‌دهد. کاربر مجبور نیست که راهنمای برنامه را بخواند تا بتواند از آن استفاده کند زیرا با بررسی کردن منوها می‌تواند به سرعت به قابلیت‌های برنامه پی برده و از آنها استفاده کند. به دلیل اینکه منو ابزاری برای کمک به کاربر جهت استفاده سریع‌تر از برنامه است، بهتر است از شکل استاندارد آن استفاده کنیم. به‌طور استاندارد دو آیتم اول منوها باید File , Edit باشد. و معمولاً در آخر آیتم help را قرار می‌دهند. سایر منوها بین Edit , Help قرار می‌گیرند. اگر منویی باعث باز شدن پنجره جدیدی می‌شود باید در جلوی عنوان آن (...) گذاشته شود.

منوها یکی از انواع منابع (Resourse) می باشند. انواع گوناگونی از منابع استاندارد وجود دارند که عبارتند از : bitmap , string , icon , menu , dialogbox و ... منابع در فایل‌های جداگانه‌ای تعریف می‌شوند. معمولاً پسوند این فایل‌ها rc. می باشد. در مرحله لینک می‌توانید منابع را با برنامه خود ترکیب کنید و در نهایت فایل اجرایی تولید شده شامل منابع مورد نیاز و دستورالعمل‌های برنامه می‌باشد.

می توانید فایل اسکریپت منابع را با هر ویرایشگری ایجاد کنید. این اسکریپت‌ها منابع مورد استفاده برنامه و مشخصات آنها را برای کامپایلر مشخص می‌کنند. اگر چه فایل اسکریپت را با هر ویرایشگری می‌توان نوشت اما این کار بسیار خسته کننده است. بهترین روش این است که از ویرایشگر منابع استفاده کنید که به شما اجازه می‌دهد به سادگی و به‌صورت بصری فایل‌های منبع را ایجاد کنید. ویرایشگرهای منابع معمولاً در بسته‌های کامپایلر مانند Visual C++ و Borland C++ وجود دارند.

اسکریپت منبع یک منو به صورت زیر است:

```
MyMenu   MENU
{
   [menu list here]
}
```

شاید برنامه نویسان C فکر کنند این تعریف یک رکورد (struct) است. MyMenu اسم منو ما را مشخص می‌کند. لیست آیتم های منو بین {} قرار می‌گیرد البته به جای این گیومه‌ها می‌توانستید از Begin و End نیز استفاده کنید که مورد علاقه برنامه نوسیان پاسکال است. MenuList می‌تواند MenuItem یا PopUp باشد. حالت MenuItem یک نوار منو تعریف می‌کند که وقتی انتخاب شد هیچ زیر منویی پایین آن ایجاد نمی‌شود. در زیر، تعریف MenuItem را مشاهده می‌کنید.

```
MENUITEM "&text", ID [,options]
```

این تعریف با کلمه کلیدی MenuItem شروع می‌شود سپس متنی را برای عنوان منو مشخص می‌کنید. علامت & قبل از رشته را فراموش نکنید چون باعث می‌شود که اولین حرف بعد از آن دارای زیر خط باشد. بعد از آن ID تعیین می‌شود. هنگامی که پیغامی به زیر پنجره ویندوز فرستاده می‌شود این ID باعث شناسایی منو آیتم می‌شود. مهمترین پیغام مربوط به منو آیتم ها پیغام انتخاب شدن آنها است.

بخش options در صورت نیاز خصوصیات منو آیتم را از حالت عادی خارج می‌کند. این خصوصیات عبارتند از:

- grayed: یعنی منو غیر فعال است و نمی‌تواند پیغام WM_COMMAND که پیغام انتخاب شدن آیتم است را ایجاد کند و نیز عنوان آیتم را خاکستری می‌کند.
- Inactive: منو غیرفعال است و نمی‌تواند پیغام WM_COMMAND را تولید کند ولی عنوان آن به‌صورت عادی می‌باشد.
- Menubreak: باعث می‌شود که این آیتم و آیتم‌های بعد از آن در سطر جدیدی از منوها ظاهر شوند.
- Help: این آیتم منو و آیتم‌های بعد از آن در طرف راست صفحه قرار می‌گیرند. به غیر از Grayed و Inactive می‌توانید سایر انتخاب های بالا را با عملگر or با یکدیگر ترکیب کنید.

تعریف منوی popup به صورت زیر است:

```
POPUP "&text" [,options]
{
  [menu list]
}
```

دستور popup یک نوار منو ایجاد می‌کند که وقتی انتخاب می‌شود، لیستی از آیتم ها در یک پنجره کوچک به سمت پایین باز می‌شود. menulist می‌تواند شامل دستورات MENUITEM یا POPUP باشد. نوع خاصی از دستور MENUITEM وجود دارد که MENUITEM SEPARATOR نامیده می‌شود و یک خط افقی در پنجره کوچک منو می‌کشد.

بعد از آن که فایل اسکریپت منبع منوی خود را ایجاد کردید باید آن را به برنامه خود ارجاع دهید. این کار را می توانید به دو صورت زیر انجام دهید.

۱- در lpszMenuName که از اعضای رکورد WNDCLASSEX می باشد تعیین کنید که منویی به نام First Menu دارید.

```
.DATA
MenuName  db "FirstMenu",0
..
..
.CODE
..
mov   wc.lpszMenuName, OFFSET MenuName
..
```

۲- از شماره دسترسی منو که از پارامتر های CreateWindowEx است استفاده کنید.

```
.DATA
MenuName  db "FirstMenu",0
hMenu HMENU ?
..
..
.CODE
..
invoke LoadMenu, hInst, OFFSET MenuName
mov   hMenu, eax
invoke CreateWindowEx,NULL,OFFSET ClsName,\
            OFFSET Caption, WS_OVERLAPPEDWINDOW,\
            CW_USEDEFAULT,CW_USEDEFAULT,\
            CW_USEDEFAULT,CW_USEDEFAULT,\
            NULL,\
            hMenu,\
            hInst,\
```

```
            NULL\
..

                                                    ..
```

ممکن است که بپرسید که تفاوت این دو روش چیست؟

وقتی منو را به رکورد WNDCLASSEX ارجاع می دهید این منو به صورت منوی پیش فرض این پنجره در می آید و تمام پنجره‌هایی که توسط این کلاس ایجاد شوند حاوی این منو خواهند بود. در صورتی که می خواهید هر پنجره‌ای که از این کلاس ایجاد می‌شود، منوی خاص خود را داشته باشد باید از روش دوم استفاده کنید. در این روش هر شماره دسترسی منویی که به تابع CreateWindowEx فرستاده شود روی حالت پیش فرض آن که در رکورد WNDCLASSEX قرار دارد بازنویسی می‌شود. حال می‌بینیم که چگونه یک منو زیر برنامه پنجره را متوجه می‌سازد که کار بر یک منو آیتم را انتخاب کرده است.

وقتی کاربر منو آیتمی را انتخاب می کند زیر برنامه پنجره یک پیغام WM_COMMAND دریافت می کند که Low word متغیر wParam حاوی ID آن آیتم می‌باشد. حال ما اطلاعات کافی برای ایجاد و استفاده از یک منو را داریم.

این مثال نشان می‌دهد که چگونه می‌توانید یک منو ایجاد کرده و با مشخص کردن اسم آن در کلاس پنجره، از آن استفاده کنید.

```
.386
.model flat,stdcall
option casemap:none
WinMain proto :DWORD,:DWORD,:DWORD,:DWORD
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
.data
ClassName db "SimpleWinClass",0
AppName  db "Our First Window",0
MenuName db "FirstMenu",0
Test_string db "You selected Test menu item",0
Hello_string db "Hello, my friend",0
Goodbye_string db "See you again, bye",0
```

```
.data?
hInstance HINSTANCE ?
CommandLine LPSTR ?
.const
IDM_TEST equ 1
IDM_HELLO equ 2
IDM_GOODBYE equ 3
IDM_EXIT equ 4
.code
start:
    invoke GetModuleHandle, NULL
    mov    hInstance,eax
    invoke GetCommandLine
    mov CommandLine,eax
    invoke WinMain, hInstance,NULL,CommandLine, SW_SHOWDEFAULT
    invoke ExitProcess,eax
WinMain proc
hInst:HINSTANCE,hPrevInst:HINSTANCE,CmdLine:LPSTR,CmdShow:DWORD
    LOCAL wc:WNDCLASSEX
    LOCAL msg:MSG
    LOCAL hwnd:HWND
    mov   wc.cbSize,SIZEOF WNDCLASSEX
    mov   wc.style, CS_HREDRAW or CS_VREDRAW
    mov   wc.lpfnWndProc, OFFSET WndProc
    mov   wc.cbClsExtra,NULL
    mov   wc.cbWndExtra,NULL
    push  hInst
    pop   wc.hInstance
    mov   wc.hbrBackground,COLOR_WINDOW+1
    mov   wc.lpszMenuName,OFFSET MenuName
    mov   wc.lpszClassName,OFFSET ClassName
    invoke LoadIcon,NULL,IDI_APPLICATION
    mov   wc.hIcon,eax
    mov   wc.hIconSm,eax
    invoke LoadCursor,NULL,IDC_ARROW
    mov   wc.hCursor,eax
    invoke RegisterClassEx, addr wc
    invoke CreateWindowEx, NULL,\
                           ADDR ClassName,\
                           ADDR AppName,\
                           WS_OVERLAPPEDWINDOW,\
                           CW_USEDEFAULT,\
                           CW_USEDEFAULT,\
                           CW_USEDEFAULT,\
                           CW_USEDEFAULT,\
                           NULL,\
                           NULL,\
                           hInst,\
                           NULL
    mov   hwnd,eax
    invoke ShowWindow, hwnd,SW_SHOWNORMAL
    invoke UpdateWindow, hwnd
```

```
    .WHILE TRUE
                invoke GetMessage, ADDR msg,NULL,0,0
                .BREAK .IF (!eax)
                invoke DispatchMessage, ADDR msg
    .ENDW
    mov     eax,msg.wParam
    ret
WinMain endp
WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
    .IF uMsg==WM_DESTROY
        invoke PostQuitMessage,NULL
    .ELSEIF uMsg==WM_COMMAND
        mov eax,wParam
        .IF ax==IDM_TEST
            invoke MessageBox,NULL,\
                              ADDR Test_string,\
                              OFFSET AppName,\
                              MB_OK
        .ELSEIF ax==IDM_HELLO
            invoke MessageBox, NULL,\
                               ADDR Hello_string,\
                               OFFSET AppName,\
                               MB_OK
        .ELSEIF ax==IDM_GOODBYE
            invoke MessageBox,NULL,\
                              ADDR Goodbye_string,\
                              OFFSET AppName,\
                              MB_OK
        .ELSE
            invoke DestroyWindow,hWnd
        .ENDIF
    .ELSE
        invoke DefWindowProc,hWnd,uMsg,wParam,lParam
        ret
    .ENDIF
    xor    eax,eax
    ret
WndProc endp
end start

;#########################################################################
;#########################################################################
Menu.rc
;#########################################################################
;#########################################################################
#define IDM_TEST 1
#define IDM_HELLO 2
#define IDM_GOODBYE 3
#define IDM_EXIT 4
FirstMenu MENU
{
 POPUP "&PopUp"
```

```
        {
         MENUITEM "&Say Hello",IDM_HELLO
         MENUITEM "Say &GoodBye", IDM_GOODBYE
         MENUITEM SEPARATOR
         MENUITEM "E&xit",IDM_EXIT
        }
 MENUITEM "&Test", IDM_TEST
}
```

ابتدا فایل منبع را تجزیه و تحلیل می‌کنیم.

```
#define IDM_TEST 1
#define IDM_HELLO 2
#define IDM_GOODBYE 3
#define IDM_EXIT 4
```

خطوط بالا IDهایی را برای منوهای تعریف شده در اسکریپت معین می‌کنند. هر مقداری را می‌توان برای این ID ها در نظر گرفت. تنها شرط لازم یکتا بودن آنها است.

```
FirstMenu MENU
```

منو با کلمه کلیدی MENU تعریف می‌شود.

```
 POPUP "&PopUp"
        {
         MENUITEM "&Say Hello",IDM_HELLO
         MENUITEM "Say &GoodBye", IDM_GOODBYE
         MENUITEM SEPARATOR
         MENUITEM "E&xit",IDM_EXIT
        }
.
```

کد بالا یک منو popup با ٤ آیتم تعریف می‌کند. توجه داشته باشید که سومین آیتم یک جدا کننده (separator) است.

```
 MENUITEM "&Test", IDM_TEST
```

تعریف یک منو بار در منو اصلی

حال کد اصلی برنامه را تجزیه و تحلیل می‌کنیم.

```
MenuName db "FirstMenu",0
Test_string db "You selected Test menu item",0
Hello_string db "Hello, my friend",0
Goodbye_string db "See you again, bye",0
```

MenuName همان اسم منوی برنامه در فایل منبع است. شما در فایل منبع می‌توانید چندین منو تعریف کنید فقط باید مشخص کنید که از کدامیک از آنها می‌خواهید استفاده کنید. بخش بعدی رشته‌هایی هستند که وقتی کاربر آیتم خاصی از منوها را انتخاب کرد در MessageBox نمایش داده خواهند شد.

```
IDM_TEST equ 1
IDM_HELLO equ 2
IDM_GOODBYE equ 3
IDM_EXIT equ 4
```

تعریف ID برای آیتم های منو به منظور استفاده در پروسیجر پنجره. این مقادیر باید دقیقاً همان مقادیر تعریف شده در فایل منبع باشند.

```
.ELSEIF uMsg==WM_COMMAND
    mov eax,wParam
    .IF ax==IDM_TEST
        invoke MessageBox,NULL,\
                          ADDR Test_string,\
                          OFFSET AppName,\
                          MB_OK
    .ELSEIF ax==IDM_HELLO
        invoke MessageBox, NULL,\
                           ADDR Hello_string,\
                           OFFSET AppName,\
                           MB_OK
    .ELSEIF ax==IDM_GOODBYE
        invoke MessageBox,NULL,\
                          ADDR Goodbye_string,\
                          OFFSET AppName,\
                          MB_OK
    .ELSE
        invoke DestroyWindow,hWnd
    .ENDIF
```

در پروسیجر پنجره به پردازش پیغام WM_COMMAND می پردازیم. وقتی کاربر یک آیتم منو را نتخاب کند Id آن همراه با پیغام WM_COMMAND در قسمت lowWord از متغیر wParam به زیر برنامه پنجره فرستاده شود. پس ما مقدار wParam را در eax ذخیره کرده و آنرا با IDهایی که از قبل در برنامه تعریف کرده ایم مقایسه می‌کنیم. در سه قسمت اول وقتی کاربر یکی از این آیتم‌ها را انتخاب می‌کند، متنی در MessageBox به نمایش در می‌آید. ولی در قسمت Exit تابع

DestroyWindow را صدا زده و شماره دسترسی پنجره خود را به‌عنوان پارامتر به آن می‌فرستیم که باعث بسته شدن پنجره ما می شود.

همان‌طور که می‌بینید اختصاص نام به منو در کلاس پنجره بسیار ساده است. با این حال می‌توانید راههای مختلفی را برای بارگذاری منو در پنجره خود انتخاب کنید.در اینجا قصد نداریم تمام کد را نشان دهیم زیرا فایل منبع در هر دو روش یکسان می‌باشد فقط باید تغییرات کوچکی در کد برنامه ایجاد کنیم که آنها را در زیر نشان می‌دهیم.

```
    data?
   hInstance HINSTANCE ?
CommandLine LPSTR ?
hMenu HMENU ?
```

تعریف یک متغییر از نوع HMENU برای ذخیره کردن شماره دسترسی به منو.

```
    invoke LoadMenu, hInst, OFFSET MenuName
    mov    hMenu,eax
    invoke CreateWindowEx, NULL,\
                           ADDR ClassName,\
                           ADDR AppName,\
                           WS_OVERLAPPEDWINDOW,\
                           CW_USEDEFAULT,\
                           CW_USEDEFAULT,\
                           CW_USEDEFAULT,\
                           CW_USEDEFAULT,\
                           NULL,\
                           NULL,\
                           hInst,\
                           NULL
```

قبل از فراخوانی تابع CreateWindowEx ، تابع LoadMenu را با یک ورودی به عنوان شماره دسترسی به منو، فراخوانی می‌کنیم. تابع LoadMenu یک اشاره گر به منوی ما در فایل منبع را بر می‌گرداند که آن را به عنوان پارامتر ورودی به تابع CreateWindowEx می‌فرستیم.

کنترل های فرزند

کدها و فایلهای مربوط به این قسمت در CD ضمیمه موجود می‌باشد
SourceCodes\Asm32\Chapter8

در این بخش نحوه مدیریت و استفاده از کنترل‌های فرزند که از مهمترین ابزارهای ورودی و خروجی برنامه‌های ما هستند، را مورد بررسی قرار خواهیم داد.

ویندوز تعدادی کلاس پنجره از پیش تعریف شده را برای برنامه‌نویسان در نظر گرفته است که اکثر مواقع از آنها به عنوان اجزا DialogBox خود استفاده می‌کنیم. به همین علت آنها را کنترل‌های فرزند می‌نامند. این کنترل‌ها خودشان پیغام‌های کی‌بورد و ماوس را پردازش می‌کنند و وقتی حالت آنها تغییر کند پنجره پدر را مطلع می‌سازند. این کنترل‌ها به میزان زیادی بار مسئولیت را از دوش برنامه‌نویسان برداشته‌اند و تا آنجا که امکان دارد باید از امکانات آنها استفاده کرد. در این قسمت از این کنترل‌ها فرزند در یک پنجره معمولی استفاده می‌کنیم ولی در واقع باید آنها را در DialogBox قرار داد.

نمونه‌هایی از کلاس‌های پنجره از پیش تعریف شده عبارتند از : edit , radio-button , checkbox , listbox , button و ... . برای استفاده از این کنترل‌ها باید آنها را توسط توابع CreateWindow یا CreateWindowEx ایجاد کنید. متذکر می‌شویم که لازم نیست کلاس‌های پنجره این کنترل‌ها را register کنید زیرا آنها توسط ویندوز برای شما register شده‌اند. پارامتر ClassName باید همان اسم از پیش تعیین شده ویندوز باشد. برای مثال اگر می‌خواهید یک button ایجاد کنید پارامتر ClassName در تابع CreateWindow را "button" قرار دهید. پارامتر های دیگری که حتما باید مشخص شوند عبارتند از شماره دسترسی به پنجره پدر و ID آن کنترل . از این ID برای شناسایی کنترل ها در پروسیجر پنجره استفاده می شود. پس از ایجاد کنترل ، هنگامی که وضعیت آن تغییر کند یا رویداد خاصی برای آن اتفاق بیفتد با پیغامی پنجره پدر را مطلع می سازد. معمولاً کنترلهای فرزند را پس از دریافت پیغام WM_CREATE از پنجره پدر می سازیم. این کنترل‌ها پیغام WM_COMMAND را همراه ID خود درLow word از متغییر wParam به پنجره پدر می‌فرستند. کد اطلاع (NotificationCode) آنها در High word از متغییر wParam وشماره دسترسی آنها در lParam قرار دارد.

پنجره پدر نیز می‌تواند توسط تابع SendMessage دستوراتی را به فرزندان خود بفرستد. تابع SendMessage می‌تواند هرگونه پیغامی را به پنجره دلخواه بفرستد. به این صورت که مقادیر

wParam ، lParam و شماره دسترسی پنجره را به عنوان پارامترهای ورودی گرفته و پیغام را به پنجره مورد نظر منتقل می‌کند.

در مثال این بخش می‌خواهیم یک پنجره بسازیم که دارای یک کنترل edit و یک دکمه است و وقتی دکمه را کلیک کنید متن درون edit توسط یک MessageBox نمایش داده می‌شود. همچنین برنامه دارای منویی با ٤ گزینه زیر است.

۱: Say Hello : یک رشته متن را درون Edit Box قرار می‌دهد.

۲: Clear Edit Box : متن درون Edit Box را پاک می‌کند.

۳: Get Text : متن درون Edit Box را توسط یک Message Box به نمایش در می‌آورد.

٤: Exit : برنامه را خاتمه می‌دهد.

```
.386
.model flat,stdcall
option casemap:none
WinMain proto :DWORD,:DWORD,:DWORD,:DWORD
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
.data
ClassName db "SimpleWinClass",0
AppName  db "Our First Window",0
MenuName db "FirstMenu",0
ButtonClassName db "button",0
ButtonText db "My First Button",0
EditClassName db "edit",0
TestString db "Wow! I'm in an edit box now",0
.data?
hInstance HINSTANCE ?
CommandLine LPSTR ?
hwndButton HWND ?
hwndEdit HWND ?
buffer db 512 dup(?)
.const
ButtonID equ 1
EditID equ 2
IDM_HELLO equ 1
IDM_CLEAR equ 2
IDM_GETTEXT equ 3
IDM_EXIT equ 4
.code
start:
```

```
    invoke GetModuleHandle, NULL
    mov    hInstance,eax
    invoke GetCommandLine
    mov CommandLine,eax
    invoke WinMain, hInstance,NULL,CommandLine, SW_SHOWDEFAULT
    invoke ExitProcess,eax
WinMain proc
hInst:HINSTANCE,hPrevInst:HINSTANCE,CmdLine:LPSTR,CmdShow:DWORD
    LOCAL wc:WNDCLASSEX
    LOCAL msg:MSG
    LOCAL hwnd:HWND
    mov   wc.cbSize,SIZEOF WNDCLASSEX
    mov   wc.style, CS_HREDRAW or CS_VREDRAW
    mov   wc.lpfnWndProc, OFFSET WndProc
    mov   wc.cbClsExtra,NULL
    mov   wc.cbWndExtra,NULL
    push  hInst
    pop   wc.hInstance
    mov   wc.hbrBackground,COLOR_BTNFACE+1
    mov   wc.lpszMenuName,OFFSET MenuName
    mov   wc.lpszClassName,OFFSET ClassName
    invoke LoadIcon,NULL,IDI_APPLICATION
    mov   wc.hIcon,eax
    mov   wc.hIconSm,eax
    invoke LoadCursor,NULL,IDC_ARROW
    mov   wc.hCursor,eax
    invoke RegisterClassEx, addr wc
    invoke CreateWindowEx, WS_EX_CLIENTEDGE,\
                           ADDR ClassName, \
                           ADDR AppName,\
                           WS_OVERLAPPEDWINDOW,\
                           CW_USEDEFAULT,\
                           CW_USEDEFAULT,\
                           300,\
                           200,\
                           NULL,\
                           NULL,\
                           hInst,\
                           NULL
    mov   hwnd,eax
    invoke ShowWindow, hwnd,SW_SHOWNORMAL
    invoke UpdateWindow, hwnd
    .WHILE TRUE
        invoke GetMessage, ADDR msg,NULL,0,0
        .BREAK .IF (!eax)
        invoke TranslateMessage, ADDR msg
        invoke DispatchMessage, ADDR msg
    .ENDW
    mov     eax,msg.wParam
    ret
WinMain endp
```

```
WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
    .IF uMsg==WM_DESTROY
        invoke PostQuitMessage,NULL
    .ELSEIF uMsg==WM_CREATE
        invoke CreateWindowEx, WS_EX_CLIENTEDGE,\
                               ADDR EditClassName,\
                               NULL,\
                               WS_CHILD or WS_VISIBLE or WS_BORDER
                               or ES_LEFT or ES_AUTOHSCROLL,\
                               50,\
                               35,\
                               200,\
                               25,\
                               hWnd,\
                               8,\
                               hInstance,\
                               NULL
        mov  hwndEdit,eax
        invoke SetFocus, hwndEdit
        invoke CreateWindowEx, NULL,\
                               ADDR ButtonClassName,\
                               ADDR ButtonText,\
                               WS_CHILD or WS_VISIBLE or
                               BS_DEFPUSHBUTTON,\
                               75,\
                               70,\
                               140,\
                               25,\
                               hWnd,\
                               ButtonID,\
                               hInstance,\
                               NULL
       mov  hwndButton,eax
    .ELSEIF uMsg==WM_COMMAND
        mov eax,wParam
        .IF lParam==0
            .IF ax==IDM_HELLO
                invoke SetWindowText,hwndEdit,ADDR TestString
            .ELSEIF ax==IDM_CLEAR
                invoke SetWindowText,hwndEdit,NULL
            .ELSEIF  ax==IDM_GETTEXT
                invoke GetWindowText,hwndEdit,ADDR buffer,512
                invoke MessageBox,NULL,\
                                  ADDR buffer,\
                                  ADDR AppName,\
                                  MB_OK
            .ELSE
                invoke DestroyWindow,hWnd
            .ENDIF
        .ELSE
            .IF ax==ButtonID
                shr eax,16
```

```
                .IF ax==BN_CLICKED
                    invoke SendMessage,hWnd,WM_COMMAND,IDM_GETTEXT,0

                .ENDIF
            .ENDIF
        .ENDIF
    .ELSE
        invoke DefWindowProc,hWnd,uMsg,wParam,lParam
        ret
    .ENDIF
     xor    eax,eax
    ret
WndProc endp
end start

    .ELSEIF uMsg==WM_CREATE
    invoke CreateWindowEx, WS_EX_CLIENTEDGE,\
                           ADDR ClassName, \
                           ADDR AppName,\
                           WS_OVERLAPPEDWINDOW,\
                           CW_USEDEFAULT,\
                           CW_USEDEFAULT,\
                           300,\
                           200,\
                           NULL,\
                           NULL,\
                           hInst,\
                           NULL
        mov  hwndEdit,eax
        invoke SetFocus, hwndEdit
        invoke CreateWindowEx, NULL,\
                               ADDR ButtonClassName,\
                               ADDR ButtonText,\
                               WS_CHILD or WS_VISIBLE or
                               BS_DEFPUSHBUTTON,\
                               75,\
                               70,\
                               140,\
                               25,\
                               hWnd,\
                               ButtonID,\
                               hInstance,\
                               NULL
        mov  hwndButton,eax
```

پس از دریافت پیغام WM_CREATE توسط تابع CreateWindowEx، کنترل‌های مورد نظر خود را ایجاد می کنیم. برای ایجاد کنترل‌ها می‌توان از سبک‌های مختلفی استفاده کرد. به عنوان مثال سبک WS_EX_CLIENTDGE باعث می‌شود که سطح کنترل یا پنجره فرورفته به نظر بیاید. هر

کنترل علاوه بر سبک‌های معمول دارای سبک‌های خاص اضافی نیز می‌باشد. به عنوان مثال سبک‌های کنترل button دارای پیشوند _BS (button style) و سبک های کنترل edit دارای پیشوند _ES (edit style) می باشند. برای پیدا کردن کلیه سبک‌ها می‌توانید به مراجع API از قبیل Win32 API Reference مراجعه کنید.

متذکر می شویم که به جای شماره دسترسی منو ، ID کنترل را قرار دهید. تا وقتی که پنجره فرزند حاوی منو نباشد این کار هیچ مشکلی به وجود نمی‌آورد. بعد از ایجاد هر کنترل ، شماره دسترسی آن را برای استفاده‌های بعدی ذخیره می‌کنیم.

برای اینکه فوکوس ورودی را به editbox بدهیم تابع SetFocus را صدا می‌زنیم. هر کنترل فرزند توسط پیغام WM_COMMAND با پنجره پدر خود ارتباط برقرار می کند.

```
.ELSEIF uMsg==WM_COMMAND
    mov eax,wParam
    .IF lParam==0
```

می‌دانید که منوها نیز با فرستادن پیغام WM_COMMAND به پنجره پدر ، آنرا از وضعیت خود با خبر می کنند. پس چگونه می توان بین پیغام های WM_COMMAND ای که از منو و کنترل فرستاده می شوند تفاوتی قائل شد؟

پاسخ را در جدول زیر مشاهده می‌کنید.

	Low word of wParam	**High word of wParam**	**lParam**
Menu	Menu ID	0	0
Control	Control ID	Notification Code	Child Window Handle

همان‌طور که می‌بینید، همیشه باید مقدار lParam را چک کنید. اگر صفر بود مشخص می‌کند که پیغام از طرف یک منو است، در غیر این صورت پیغام متعلق به یک کنترل است. برای این منظور نمی‌توانید از wParam استفاده کنید چون Notification Code نیز می‌تواند مقدار صفر داشته باشد.

```
.IF ax==IDM_HELLO
     invoke SetWindowText,hwndEdit,ADDR TestString
.ELSEIF ax==IDM_CLEAR
     invoke SetWindowText,hwndEdit,NULL
.ELSEIF  ax==IDM_GETTEXT
     invoke GetWindowText,hwndEdit,ADDR buffer,512
     invoke MessageBox,NULL,ADDR buffer,ADDR AppName,MB_OK
```

با فراخوانی تابع SetWindowText می‌توانید یک رشته متن را در EditBox قرار دهید یا اینکه با فراخوانی تابع با مقدار Null محتویات EditBox را پاک کنید. توسط این تابع می‌توانید عنوان یک Label یا یک دکمه را نیز عوض کنید. برای دسترسی به متن موجود در EditBox نیز می‌توانید از تابع GetWindowText استفاده کنید.

```
.IF ax==ButtonID
     shr eax,16
    .IF ax==BN_CLICKED
        invoke SendMessage,hWnd,WM_COMMAND,IDM_GETTEXT,0
    .ENDIF
.ENDIF
```

ابتدا چک می کنیم که آیا قسمت Low word از متغیر wParam با ID کنترل برابر است یا نه. اگر درست بود قسمت High word از متغیر wParam را چک می‌کنیم. در صورتی که کد اطلاع BN_CLICK بود، باید متن داخل EditBox را در یک MessageBox نمایش دهیم. برای این کار می‌توانیم کد را دوباره از قسمت IDM_GETEXT بازنویسی کنیم. اما این کار اصلا معقول نیست. اگر بتوانیم یک پیغام WM_COMMAND را که قسمت Low word از wParam آن شامل IDM_GETTEXT باشد به زیر برنامه پنجره بفرستیم توانسته‌ایم از بازنویسی کد جلوگیری کرده و برنامه خود را ساده‌تر سازیم. پاسخ این نیاز ، تابع SendMessage است. پس به جای بازنویسی کد، تابع SendMessage را با شماره دسترسی پنجره پدر و پیغام WM_COMMAND در Low wordاز wParam و IDM_GETTEXT در lParam فراخوانی می‌کنیم. این کار دقیقاً همان اثر انتخاب آیتم GetText از منو را داشته و زیر برنامه پنجره هیچ تفاوتی بین این دو قائل نیست.

تا حد ممکن باید از این تکنیک استفاده کرد. زیرا باعث سازمان دهی بهتر کدهای برنامه می شود. در نهایت تابع TranslateMessage را در MessageLoop فراموش نکنید. از زمانی که شما شروع به تایپ متن درEditBox می‌کنید برنامه شما باید ورودی خام کی‌بورد را به یک ورودی قابل خواندن ترجمه کند. اگر این تابع را فراخوانی نکنید امکان تایپ در EditBox وجود نخواهد داشت.

استفاده از DialogBox به عنوان پنجره اصلی برنامه

کدها و فایلهای مربوط به این قسمت در CD ضمیمه موجود می‌باشد

SourceCodes\Asm32\Chapter9

اگر با مثال فصل قبل به اندازه کافی کار کرده باشید در می یابید که توسط کلید Tab نمی توانید فوکوس را تغییر دهید و تنها راه تغییر فوکوس کلیک کردن روی کنترل مورد نظر است. این وضعیت بسیار خسته کننده است. مسئله دیگری که شاهد آن بوده اید تغییر رنگ پشت زمینه پنجره پدر از سفید به خاکستری است. علت این امر این است که رنگ پنجره فرزند به صورت هماهنگ با منطقه کاری پنجره پدر ترکیب می شود. و نیز این نکته قابل توجه است که تمام پنجره‌ها (کنترلها)ی فرزند باید یک زیر کلاس از پنجره پدر باشند.

دلیل به وجود آمدن چنین ناسازگاری‌هایی این است که کنترل‌های فرزند در اصل برای کار با DialogBox ها طراحی شده اند نه پنجره‌های معمولی. قبل از اینکه وارد جزئیات شویم لازم است که بدانیم DialogBox چیست. DialogBox چیزی بیش از یک پنجره معمولی نیست که برای کار با کنترل‌های فرزند طراحی شده است.

ویندوز بخشی را به منظور مدیریت DialogBox ها تدارک دیده است که یکی از وظایف آن هماهنگی اعمال منطقی کی بورد با پنجره DialogBox است. مانند جابجا کردن فوکوس توسط کاربر با زدن کلیک Tab و یا زدن Enter به عنوان کلید پیش‌فرض برای تایید و ... . دیالوگ باکسها اساسا به عنوان ابزار ورودی و خروجی به کار می‌روند. ما نیازی به این نداریم که بدانیم یک دیالوگ باکس خود چگونه عمل می‌کند. تنها چیزی که باید بدانیم اینست که چگونه با آن کار کنیم. این یکی از اصول برنامه نویسی شی گراست که به آن پنهان سازی اطلاعات گفته می شود. اگر یک ابزار به خوبی طراحی شده باشد کاربر از آن به سادگی استفاده می کند بدون این که بداند این شی در درون خود چگونه عمل می کند. پس باید ابزار به گونه ای بسیار دقیق طراحی شده باشد و این کار در واقع بسیار سخت است. در حقیقت API Win32 نیز به صورت یک جعبه ابزار عمل می کند که اعمال درونی خود را از دید کاربران مخفی نگه می دارد. حال دوباره به بحث اصلی باز می‌گردیم.

دیالوگ باکسها برای کاهش حجم کار برنامه نویسان هستند. به صورت معمول اگر یک کنترل فرزند را در یک پنجره معمولی قرار دهید خودتان باید مدیریت رویدادهای آن را بر عهده گرفته و عملیات منطقی کی بورد را انجام دهید. ولی اگر آن ها را در یک دیالوگ باکس قرار دهید بطور اتوماتیک این

عملیات انجام شوند تنها چیزی که باید بدانید این است که چگونه اطلاعات کاربر را از دیالوگ باکس بگیرید و چگونه به آن دستور بدهید.

دیالوگ باکس نیز مانند منو یکی از انواع منابع (resource) است. شما خصوصیات یک دیالوگ باکس را به همراه کنترل‌های آن می‌نویسید سپس توسط ویرایشگر منبع آنرا کامپایل می کنید. متذکر می‌شویم که کلیه منابع از قبیل منو ، کرسر ، دیالوگ باکس و غیره همه در کنار هم در یک فایل منبع قرار می گیرند. می توانید از هر ویرایشگر متنی برای ایجاد منابع دیالوگ باکس استفاده کنید. ولی ما این کار را توصیه نمی کنیم. بهتر است از یک ویرایشگر منبع برای این کار استفاده کنید زیرا قرار دادن کنترل های فرزند در یک دیالوگ باکس به‌صورت دستی بسیار سخت است.

دو نوع اصلی از دیالوگ باکس ها وجود دارند Modal و Modeless . نوع Modless به شما اجازه می‌دهد که فوکوس ورودی را به پنجره های دیگر (مانند پنجره Find در مایکروسافت Word) منتقل کنید. نوع بعدی Modal است که خود شامل دو بخش می باشد:

١- Application Modal: به شما اجازه نمی دهد فوکوس را به دیگر پنجره های همان برنامه منتقل کنید ولی اجازه انتقال فوکوس را به سایر برنامه‌های در حال اجرا در ویندوز می‌دهد.

٢- System Modal: قبل از اینکه به این پنجره پاسخ دهید اجازه تغییر فوکوس را به هیچ پنجره ای نمی دهد.

نوع Modless با فراخونی تابع CreateDialogParam و نوع Modal با فراخوانی تابع DialogBoxParam ایجاد می شوند. تنها تفاوت میان نوع Application Modal و System Modal در فرم DS_SYSMODAL است. اگر در الگوی دیالوگ باکس از این فرم استفاده کنیم به نوع SystemModal تبدیل می شود. با استفاده از تابع SendDlgItemMessage می توانید با هر یک از کنترل های فرزند دیالوگ باکس ارتباط برقرار کنید. ساختار این تابع بصورت زیر است.

```
SendDlgItemMessage proto hwndDlg:DWORD,\
                         idControl:DWORD,\
                         uMsg:DWORD,\
                         wParam:DWORD,\
                         lParam:DWORD
```

این تابع برای ارتباط با کنترل های فرزند بسیار مفید است برای مثال در صورتی که بخواهید یک متن را از EditBox بگیرید می توانید از دستور زیر استفاده کنید:

```
call SendDlgItemMessage, hDlg,\
                         ID_EDITBOX,\
                         WM_GETTEXT,\
                         256,\
                         ADDR text_buffer
```

برای اطلاعات بیشتر در مورد نحوه ارسال پیغامهای مختلف می توانید به Win32 API Reference مراجعه کنید. ویندوز برای عملیات دریافت و ارسال اطلاعات چندین تابع ارائه کرده است از قبیل GetDlgItemText , CheckDlgButton و ... . این توابع برنامه را بسیار ساده می‌کنند و باعث می‌شوند که برنامه نویسان برای هر پیغام درگیر مقادیر wParam , lParam نشوند.

Windows DialogBox Manager پیغام ها را به پروسیجرهای دیالوگ باکس می‌فرستد. به عنوان مثال ساختار یکی از این پروسیجرها به‌صورت زیر است.

```
DlgProc  proto hDlg:DWORD ,\
               iMsg:DWORD ,\
               wParam:DWORD ,\
               lParam:DWORD
```

پروسیجر دیالوگ باکس بسیار شبیه به پروسیجر پنجره است با این تفاوت که به جای LRESULT مقدار خروجی آن True/False است.DialogBox Manager داخلی خود ویندوز نیز در حقیقت یک پروسیجر پنجره برای دیالوگ باکس است. این پروسیجر ، پروسیجر دیالوگ ما را با پیغام هایی که دریافت می کند فراخوانی می کند. قاعده کلی در مورد پردازش پیغام ها این است که اگر یک زیر برنامه دیالوگ باکس پیغامی را پردازش کرد باید مقدار True را در eax بازگرداند و در غیر این صورت از مقدار False استفاده کند. پروسیجر دیالوگ باکس پیغام های پردازش نشده را دیگر به تابع DefWindowProc نمی فرستد چون این پروسیجر یک پروسیجر پنجره واقعی نیست. از دیالوگ باکس‌ها می‌توان به دو روش استفاده کرد. ۱- می توانید از آن به عنوان پنجره اصلی برنامه خود استفاده کنید. ۲- می‌توانید از آن به عنوان یک ابزار ورودی استفاده کنید.

در این بخش به بررسی روش اول پردازیم.

می توانید از ساختار دیالوگ باکس به عنوان یک الگوی کلاس استفاده کرده و آنها توسط تابع RegisterClassEx ثبت کنید. در این حالت دیالوگ باکس همانند یک پنجره معمولی رفتار می کند و پیغام ها را از طریق پروسیجر پنجره دریافت می کند. مزایای این روش این است که دیگر مجبور

نیستید خودتان کنترل های فرزند را بوجود بیاورید زیرا خود ویندوز آنها را در هنگام ایجاد دیالوگ باکس ، برای شما ایجاد می کند. ویندوز عملیات منطقی کی بورد مانند Tab Order را نیز خود بر عهده می گیرد. علاوه بر این می توانید کرسر و آیکون پنجره خود را در ساختار کلاس پنجره تعیین کنید. در این روش برنامه شما یک دیالوگ باکسی ایجاد می کند که پنجره پدری ندارد.

حال به سراغ مثال این بخش می رویم.

```
.386
.model flat,stdcall
option casemap:none
WinMain proto :DWORD,:DWORD,:DWORD,:DWORD
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
.data
ClassName db "DLGCLASS",0
MenuName db "MyMenu",0
DlgName db "MyDialog",0
AppName db "Our First Dialog Box",0
TestString db "Wow! I'm in an edit box now",0
.data?
hInstance HINSTANCE ?
CommandLine LPSTR ?
buffer db 512 dup(?)
.const
IDC_EDIT                        equ               3000
IDC_BUTTON                      equ               3001
IDC_EXIT                        equ               3002
IDM_GETTEXT                     equ              32000
IDM_CLEAR                       equ              32001
                                          IDM_EXIT        equ 32002
.code
start:
    invoke GetModuleHandle, NULL
    mov    hInstance,eax
    invoke GetCommandLine
    mov CommandLine,eax
    invoke WinMain, hInstance,NULL,CommandLine, SW_SHOWDEFAULT
    invoke ExitProcess,eax
WinMain proc
hInst:HINSTANCE,hPrevInst:HINSTANCE,CmdLine:LPSTR,CmdShow:DWORD
    LOCAL wc:WNDCLASSEX
    LOCAL msg:MSG
    LOCAL hDlg:HWND
    mov   wc.cbSize,SIZEOF WNDCLASSEX
    mov   wc.style, CS_HREDRAW or CS_VREDRAW
```

```
    mov   wc.lpfnWndProc, OFFSET WndProc
    mov   wc.cbClsExtra,NULL
    mov   wc.cbWndExtra,DLGWINDOWEXTRA
    push  hInst
    pop   wc.hInstance
    mov   wc.hbrBackground,COLOR_BTNFACE+1
    mov   wc.lpszMenuName,OFFSET MenuName
    mov   wc.lpszClassName,OFFSET ClassName
    invoke LoadIcon,NULL,IDI_APPLICATION
    mov   wc.hIcon,eax
    mov   wc.hIconSm,eax
    invoke LoadCursor,NULL,IDC_ARROW
    mov   wc.hCursor,eax
    invoke RegisterClassEx, addr wc
    invoke CreateDialogParam,hInstance,ADDR DlgName,NULL,NULL,NULL
    mov   hDlg,eax
    invoke ShowWindow, hDlg,SW_SHOWNORMAL
    invoke UpdateWindow, hDlg
    invoke GetDlgItem,hDlg,IDC_EDIT
    invoke SetFocus,eax
    .WHILE TRUE
        invoke GetMessage, ADDR msg,NULL,0,0
        .BREAK .IF (!eax)
       invoke IsDialogMessage, hDlg, ADDR msg
        .IF eax ==FALSE
            invoke TranslateMessage, ADDR msg
            invoke DispatchMessage, ADDR msg
        .ENDIF
    .ENDW
    mov     eax,msg.wParam
    ret
WinMain endp
WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
    .IF uMsg==WM_DESTROY
        invoke PostQuitMessage,NULL
    .ELSEIF uMsg==WM_COMMAND
        mov eax,wParam
        .IF lParam==0
            .IF ax==IDM_GETTEXT
                invoke GetDlgItemText,hWnd,IDC_EDIT,ADDR buffer,512
                invoke MessageBox,NULL,\
                                  ADDR buffer,\
                                  ADDR AppName,\
                                  MB_OK
            .ELSEIF ax==IDM_CLEAR
                invoke SetDlgItemText,hWnd,IDC_EDIT,NULL
            .ELSE
                invoke DestroyWindow,hWnd
            .ENDIF
        .ELSE
            mov edx,wParam
            shr edx,16
```

```
            .IF dx==BN_CLICKED
                .IF ax==IDC_BUTTON
                    invoke SetDlgItemText,hWnd,\
                                          IDC_EDIT,\
                                          ADDR TestString
                .ELSEIF ax==IDC_EXIT
                    invoke SendMessage,hWnd,WM_COMMAND,IDM_EXIT,0
                .ENDIF
            .ENDIF
        .ENDIF
    .ELSE
        invoke DefWindowProc,hWnd,uMsg,wParam,lParam
        ret
    .ENDIF
    xor    eax,eax
    ret
WndProc endp
end start
```

Dialog.rc

```
#include "resource.h"
#define IDC_EDIT                                    3000
#define IDC_BUTTON                                  3001
#define IDC_EXIT                                    3002
#define IDM_GETTEXT                                 32000
#define IDM_CLEAR                                   32001
#define IDM_EXIT                                    32003

MyDialog DIALOG 10, 10, 205, 60
STYLE 0x0004 | DS_CENTER | WS_CAPTION | WS_MINIMIZEBOX |
WS_SYSMENU | WS_VISIBLE | WS_OVERLAPPED | DS_MODALFRAME | DS_3DLOOK
CAPTION "Our First Dialog Box"
CLASS "DLGCLASS"
BEGIN
    EDITTEXT         IDC_EDIT,   15,17,111,13, ES_AUTOHSCROLL |
ES_LEFT
    DEFPUSHBUTTON   "Say Hello", IDC_BUTTON,    141,10,52,13
    PUSHBUTTON      "E&xit", IDC_EXIT,  141,26,52,13, WS_GROUP
END

MyMenu  MENU
BEGIN
    POPUP "Test Controls"
```

```
    BEGIN
        MENUITEM "Get Text", IDM_GETTEXT
        MENUITEM "Clear Text", IDM_CLEAR
        MENUITEM "", , 0x0800 /*MFT_SEPARATOR*/
        MENUITEM "E&xit", IDM_EXIT
    END
END
```

این مثال نشان می دهد که چگونه ساختار دیالوگ باکس را به عنوان یک کلاس پنجره رجیستر کرده و سپس با استفاده از آن ، پنجره مورد نظر خود را ایجاد کنید. همان طور که گفته شد ، دیگر لازم نیست کنترل های فرزند را بطور دستی ایجاد کنید زیرا آنها به طور خودکار برای دیالوگ باکس ایجاد می شوند. حال ساختار دیالوگ باکس را بررسی می کنیم.

```
MyDialog DIALOG 10, 10, 205, 60
```

در ابتدا نام دیالوگ مورد نظر را اعلام می کنیم و چهار عدد بعد از آن به ترتیب y ، x ، عرض و ارتفاع دیالوگ باکس هستند.

```
STYLE 0x0004 | DS_CENTER | WS_CAPTION | WS_MINIMIZEBOX |
WS_SYSMENU | WS_VISIBLE | WS_OVERLAPPED | DS_MODALFRAME | DS_3DLOOK
```

شکل ظاهری دیالوگ باکس را تعیین می کنیم.

```
CAPTION "Our First Dialog Box"
```

این متن در نوار عنوان دیالوگ باکس نشان داده می شود.

```
CLASS "DLGCLASS"
```

این خط بسیار مهم است. این کلمه کلیدی CLASS است که به ما اجازه می دهد از الگوی دیالوگ باکس به عنوان یک کلاس پنجره استفاده می کنیم. بعد از این کلمه کلیدی نام کلاس قرار دارد.

```
BEGIN
    EDITTEXT      IDC_EDIT,   15,17,111,13, ES_AUTOHSCROLL | ES_LEFT

    DEFPUSHBUTTON   "Say Hello", IDC_BUTTON,    141,10,52,13
    PUSHBUTTON      "E&xit", IDC_EXIT,  141,26,52,13
END
```

بلوک بالا کنترل های فرزند را تعریف می کند. آنها بین End , Begin قرار می گیرند. معمولاً روش تعریف کنترل های فرزند بصورت زیر است.

```
control-type  "text"   ,controlID, x, y, width, height [,styles]
```

Control-Type از ثابت های منبع است. برای دیدن مقادیر مجاز آن می توانید به مستندات منابع در MSDN مراجعه کنید.

```
mov   wc.cbWndExtra,DLGWINDOWEXTRA
mov   wc.lpszClassName,OFFSET ClassName
```

معمولاً به این عضو مقدار Null می دهیم ولی هنگامی که می خواهیم یک ساختار دیالوگ باکس را به عنوان کلاس پنجره ثبت کنیم ، باید به این عضو مقدار DLGWINDOWEXTRA را نسبت دهیم. متذکر می شویم که نام کلاس باید عیناً همان نامی باشد که بعد از کلمه کلیدی CLASS نوشته شده است. سایر اعضا نیز همانند قبل معرفی می شوند. بعد از اینکه اعضا رکورد پنجره معرفی شدند ، آن کلاس را توسط تابع RegisterClassEx ثبت می کنیم. اینها دقیقاً همان اعمالی هستند که برای ثبت کردن یک کلاس پنجره معمولی انجام می دادیم.

```
invoke CreateDialogParam,hInstance,ADDR DlgName,NULL,NULL,NULL
```

پس از ثبت کلاس پنجره ، دیالوگ باکس خود را ایجاد می کنیم. در این مثال دیالوگ باکس خود را از نوع Modless انتخاب کرده و با استفاده از تابع CreateDialogParam آنرا ایجاد می کنیم. این تابع ٥ پارامتر ورودی می گیرد که ما فقط دو تای اول را مقدار دهی می کنیم که عبارتند از: شماره دسترسی برنامه و اشاره گر به نام ساختار دیالوگ باکس. متذکر می شویم که پارامتر دوم اشاره گر به نام کلاس نیست.

در این مرحله دیالوگ باکس به همراه کنترل های فرزند خود توسط ویندوز اجرا شده است و پروسیجر پنجره طبق معمول پیغام WM_CREATE را دریافت خواهد کرد.

```
invoke GetDlgItem,hDlg,IDC_EDIT
invoke SetFocus,eax
```

پس از اینکه دیالوگ باکس ایجاد شد می خواهیم فوکوس ورودی را به کنترل Edit انتقال می دهیم. اگر این کد را در بخش WM_CREATE قرار دهیم فراخوانی تابع GetDlgItem با شکست مواجه می شود زیرا در آن زمان کنترل های فرزند هنوز ایجاد نشده اند. پس باید این بخش از کد را بعد از اینکه دیالوگ باکس و کنترل های فرزند آن بوجود آمدند بنویسیم. ما این دو خط را پس از فراخوانی تابع UpdateWindow قرار می دهیم. تابع GetDlgItem شماره شناسه کنترل را گرفته و شماره دسترسی به پنجره آنرا باز می گرداند.

```
invoke IsDialogMessage, hDlg, ADDR msg
 .IF eax ==FALSE
     invoke TranslateMessage, ADDR msg
     invoke DispatchMessage, ADDR msg
 .ENDIF
```

برنامه وارد MessageLoop می شود و ما قبل از این که پیغام را ترجمه کنیم تابع IsDialogMessage را برای اینکه DialogBox Manager عملیات منطقی کی بورد را انجام دهد فراخوانی می کنیم. در صورتی که این تابع مقدار True را برگرداند به این معنا است که پیغام مفهوم بوده و DialogBox Manager آنرا پردازش کرده است. یکی دیگر از تفاوتها با فصل قبل در این است که وقتی زیر برنامه پنجره می خواهد متن را از کنترل Edit بگیرد ، به جای تابع GetWindowText از تابع GetDlgItemText استفاده می کند. این تابع شماره شناسه را به جای شماره دسترسی به پنجره می پذیرد که بسیار ساده تر است.

در مثال بعد قصد داریم به روش دیگر از دیالوگ باکس به عنوان پنجره اصلی استفاده کنیم. در این مثال یک دیالوگ باکس از نوع Application Modal ایجاد خواهیم کرد که فاقد MessageLoop و پروسیجر پنجره است.

```
.386
.model flat,stdcall
option casemap:none
DlgProc proto :DWORD,:DWORD,:DWORD,:DWORD
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
.data
DlgName db "MyDialog",0
AppName db "Our Second Dialog Box",0
TestString db "Wow! I'm in an edit box now",0
.data?
hInstance HINSTANCE ?
CommandLine LPSTR ?
buffer db 512 dup(?)
.const
IDC_EDIT            equ 3000
IDC_BUTTON     equ 3001
IDC_EXIT            equ 3002
IDM_GETTEXT  equ 32000
IDM_CLEAR       equ 32001
IDM_EXIT           equ 32002

.code
start:
    invoke GetModuleHandle, NULL
    mov    hInstance,eax
    invoke DialogBoxParam, hInstance, ADDR DlgName,NULL, addr
DlgProc, NULL
    invoke ExitProcess,eax
DlgProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
    .IF uMsg==WM_INITDIALOG
        invoke GetDlgItem, hWnd,IDC_EDIT
        invoke SetFocus,eax
    .ELSEIF uMsg==WM_CLOSE
        invoke SendMessage,hWnd,WM_COMMAND,IDM_EXIT,0
    .ELSEIF uMsg==WM_COMMAND
        mov eax,wParam
        .IF lParam==0
            .IF ax==IDM_GETTEXT
                invoke GetDlgItemText,hWnd,IDC_EDIT,ADDR buffer,512
                invoke MessageBox,NULL,\
                                  ADDR buffer,\
                                  ADDR AppName,\
                                  MB_OK
            .ELSEIF ax==IDM_CLEAR
                invoke SetDlgItemText,hWnd,IDC_EDIT,NULL
```

```
            .ELSEIF ax==IDM_EXIT
                invoke EndDialog, hWnd,NULL
            .ENDIF
        .ELSE
            mov edx,wParam
            shr edx,16
            .if dx==BN_CLICKED
                .IF ax==IDC_BUTTON
                    invoke SetDlgItemText,hWnd,\
                                          IDC_EDIT,\
                                          ADDR TestString
                .ELSEIF ax==IDC_EXIT
                    invoke SendMessage,hWnd,WM_COMMAND,IDM_EXIT,0
                .ENDIF
            .ENDIF
        .ENDIF
    .ELSE
        mov eax,FALSE
        ret
    .ENDIF
    mov eax,TRUE
    ret
DlgProc endp
end start
```

dialog.rc (part 2)

```
#include "resource.h"
#define IDC_EDIT                                3000
#define IDC_BUTTON                              3001
#define IDC_EXIT                                3002
#define IDR_MENU1                               3003
#define IDM_GETTEXT                             32000
#define IDM_CLEAR                               32001
#define IDM_EXIT                                32003

MyDialog DIALOG 10, 10, 205, 60
STYLE 0x0004 | DS_CENTER | WS_CAPTION | WS_MINIMIZEBOX |
WS_SYSMENU | WS_VISIBLE | WS_OVERLAPPED | DS_MODALFRAME | DS_3DLOOK
CAPTION "Our Second Dialog Box"
MENU IDR_MENU1
BEGIN
    EDITTEXT         IDC_EDIT,   15,17,111,13, ES_AUTOHSCROLL |
ES_LEFT
    DEFPUSHBUTTON   "Say Hello", IDC_BUTTON,    141,10,52,13
    PUSHBUTTON      "E&xit", IDC_EXIT,  141,26,52,13
END

IDR_MENU1  MENU
BEGIN
```

```
    POPUP "Test Controls"
    BEGIN
        MENUITEM "Get Text", IDM_GETTEXT
        MENUITEM "Clear Text", IDM_CLEAR
        MENUITEM "", , 0x0800 /*MFT_SEPARATOR*/
        MENUITEM "E&xit", IDM_EXIT
    END
END
```

```
    DlgProc proto :DWORD,:DWORD,:DWORD,:DWORD
```

ابتدا پیش تعریف تابع DlgProc را مشاهده می کنید. که می توانید با استفاده از عملگر addr آنرا به تابع DialogBoxParam ارجاع دهید.

```
  invoke DialogBoxParam, hInstance,\
                         ADDR DlgName,\
                         NULL,\
                         addr DlgProc,\
                         NULL
```

خط بالا تابع DialogBoxParam را با ٥ پارامتر فراخوانی می کند که عبارتند از: شماره دسترسی برنامه ، نام ساختار و دیالوگ باکس ، شماره دسترسی پنجره پدر ، آدرس پروسیجر دیالوگ باکس و اطلاعات خاص در مورد دیالوگ.

تابع DialogBoxParam یک دیالوگ باکس از نوع modal ایجاد می کند و تا زمانی که دیالوگ باکس از بین نرود بازگشت نمی کند

```
    .IF uMsg==WM_INITDIALOG
        invoke GetDlgItem, hWnd,IDC_EDIT
        invoke SetFocus,eax
    .ELSEIF uMsg==WM_CLOSE
        invoke SendMessage,hWnd,WM_COMMAND,IDM_EXIT,0
```

پروسیجر دیالوگ باکس همانند پروسیجر پنجره عمل می‌کند با این تفاوت که پیغام WM_CREATE دریافت نمی‌کند. اولین پیغامی که این پروسیجر دریافت می کند پیغام WM_INITDIALOG است. می توانید کد مربوط به عملیات ابتدایی برنامه را در این قسمت قرار دهید.

DialogBox Manager برای بستن پنجره از پیغام WM_CLOSE به جای WM_DESTROYاستفاده می کند. پس اگر می خواهید هنگامی که کاربر دکمه close را کلیک می کند عکس العملی را نشان دهید باید پیغام WM_CLOSE را پردازش کنید. در مثال بالا ما در پاسخ به پیغام IDM_EXIT تابع EndDialog را فراخوانی می کنیم.

وقتی که می خواهید دیالوگ باکسی را از بین ببرید تنها راه اینست که از تابع EndDialog استفاده کنید. در این مورد تابع DestroyWindow نمی تواند به شما هیچ کمکی بکند. حال بیایید فایل منبع را آزمایش کنیم تغییر قابل ذکر این است که به جای استفاده از رشته متنی برای اسم منو از مقدار IDR_Menu1 استفاده کردیم زیرا برای به کار گیری منو در دیالوگ باکسی که با دستور DialogBoxWindow بوجود آمده است ، استفاده از این روش ضروری می باشد. متذکر می‌شویم که در ساختار دیالوگ باکس باید کلمه کلیدی Menu را که به دنبال آن شماره شناسایی منبع می‌آید اضافه کنیم.

استفاده از DialogBox به عنوان ابزار ورودی / خروجی

کدها و فایلهای مربوط به این قسمت در CD ضمیمه موجود می‌باشد

SourceCodes\Asm32\Chapter10

در این بخش نکات بیشتری در مورد دیالوگ باکس خواهیم آموخت. به عنوان مثال می آموزیم که چگونه از دیالوگ باکس های استاندارد ویندوز به عنوان ابزارهای ورودی و خروجی استفاده کنیم.

با کمی توضیح نحوه استفاده از دیالوگ باکسها را به عنوان ابزارهای ورودی و خروجی برنامه خواهید آموخت. برنامه شما طبق معمول پنجره اصلی را می سازد. و برای نمایش دیالوگ باکسها از توابع DialogBoxParam و CreateDialogParam استفاده می کند. در صورت استفاده از تابع DialogBoxParam لازم نیست کار دیگری غیر از پردازش پیغام های رسیده به پروسیجر دیالوگ انجام دهید. ولی در صورت فراخوانی و استفاده از تابع CreateDialogParam باید تابع IsDialogMessage را در حلقه پیغامها قرار دهید تا هدایت عملیات منطقی کی بورد را به DialogBox Manager ویندوز واگذار کند.

ویندوز دیالوگ باکسهای از پیش تعریف شده ای را برای برنامه نویسان ارائه کرده است که هدف اصلی آنها ایجاد یک رابط کاربر استاندارد برای کاربران است. برخی از این دیالوگ باکسها عبارتند از: چاپ ، رنگ ، فونت ، فایل و ... . این دیالوگ باکسها در فایل Comdlg.dll قرار دارند و برای استفاده از آنها باید با فایل Comdlg.lib ارتباط برقرار کنید. با فراخوانی توابع مناسب در این فایل می توانید از قابلیتهای این دیالوگ ها در برنامه خود استفاده کنید. به عنوان مثال برای استفاده از دیالوگ OpenFile از تابع GetOpenFileName و برای استفاده از دیالوگ Save File از تابع GetSaveFileName کمک می گیریم. هر یک از این توابع یک اشاره گر به رکورد را به عنوان پارامتر ورودی می گیرند. برای اطلاعات بیشتر در مورد این رکورد ها و پارامترهای آنها می توانید به Win32 API Reference مراجعه کنید. در این بخش چگونگی ایجاد و استفاده از دیالوگ OpenFile را بیان می کنیم.

```
GetOpenFileName proto lpofn:DWORD
```

همانطور که مشاهده می کنید این تابع تنها یک پارامتر ورودی دریافت می کند که اشاره گری به رکورد OPENFILENAME است. اگر مقدار برگشتی این تابع True باشد به این معنی است که

کاربر فایلی را برای باز کردن انتخاب کرده است. در غیر این صورت مقدار برگشتی از تابع False خواهد بود.

حال به بررسی رکورد OPENFILENAME می پردازیم.

```
STRUCT OPENFILENAME
      lStructSize          DWORD        ?
      hwndOwner            HWND         ?
      hInstance            HINSTANCE    ?
      lpstrFilter          LPCSTR       ?
      lpstrCustomFilter    LPSTR        ?
      nMaxCustFilter       DWORD        ?
      nFilterIndex         DWORD        ?
      lpstrFile            LPSTR        ?
      nMaxFile             DWORD        ?
      lpstrFileTitle       LPSTR        ?
      nMaxFileTitle        DWORD        ?
      lpstrInitialDir      LPCSTR       ?
      lpstrTitle           LPCSTR       ?
      Flags                DWORD        ?
      nFileOffset          WORD         ?
      nFileExtension       WORD         ?
      lpstrDefExt          LPCSTR       ?
      lCustData            LPARAM       ?
      lpfnHook             DWORD        ?
      lpTemplateName       LPCSTR       ?
OPENFILENAME  ENDS
```

در این بخش برخی از اعضا این رکورد را که مورد استفاده عمومی دارند شرح می دهیم. برای توضیحات بیشتر در مورد اعضا این رکورد می توانید از مراجع API کمک بگیرید.

lStructSize : طول رکورد OPENFILENAME بر حسب بایت.

hwndOwner : شماره دسترسی پنجره دیالوگ.

hInstance : شماره دسترسی برنامه ای که دیالوگ را ایجاد کرده است.

lpstrFilter : شامل یک یا چند جفت رشته مختوم به صفر است که در هر جفت ، رشته اول توضیح آن نوع فایل و رشته دوم الگوی فیلتر می باشد. به عنوان مثال:

```
FilterString   db "All Files (*.*)",0, "*.*",0
               db "Text Files (*.txt)",0,"*.txt",0,0
```

متذکر می شویم که رشته دوم در هر جفت توسط ویندوز برای فیلتر کردن فایلها استفاده می شود. همچنین باید دقت داشته باشید که یک صفر اضافه در انتهای رشته فیلتر قرار دهید تا رشته مختوم به صفر شود.

nFilterIndex : مشخص می کند که برای بار اول از کدام فیلتر استفاده شود. مقدار 1 برای فیلتر اول ، 2 برای فیلتر دوم و به همین ترتیب ادامه می دهیم. در مثال این بخش از فیلتر دوم یعنی" *.txt" به عنوان فیلتر پیش فرض استفاده می کنیم.

lpstrFile : اشاره گری به بافری است که نام و مسیر کامل فایل انتخاب شده را در خود دارد. حداقل طول این بافر ۲۶۰ بایت است. هنگامی که کاربر فایلی را برای باز شدن انتخاب کند نام و مسیر کامل آن در این بافر ذخیره می شود که می توانید در صورت نیاز از آن استفاده کنید.

nMaxFile : طول بافر lpstrFile.

lpstrTitle : اشاره گر به رشته عنوان دیالوگ باکس.

Flags : خصوصیات و نوع دیالوگ باکس را تعیین می کند.

nFileOffset : هنگامی که کاربر فایلی را انتخاب می کند این مقدار شماره اولین کاراکتر نام اصلی فایل را مشخص می کند. به عنوان مثال اگر نام و مسیر یک فایل به صورت c:\windows\system\x.dll باشد مقدار این متغیر ۱۸ یعنی اندیس حرف x در رشته مسیر فایل خواهد بود.

nFileExtension : در صورت انتخاب فایل توسط کاربر این عضو شماره اندیس اولین کاراکتر پسوند آنرا در رشته مسیر فایل در خود دارد.

حال به سراغ مثال این بخش می رویم.

با انتخاب گزینه Open از منوی File ، برنامه یک دیالوگ باکس از نوع OpenFile را نمایش می‌دهد و اگر کاربر فایلی را انتخاب کند نام و مسیر کامل آنرا در یک MessageBox به نمایش می‌گذارد.

```
.386
.model flat,stdcall
option casemap:none
WinMain proto :DWORD,:DWORD,:DWORD,:DWORD
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
include \masm32\include\comdlg32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\comdlg32.lib
.const
IDM_OPEN equ 1
IDM_EXIT equ 2
MAXSIZE equ 260
OUTPUTSIZE equ 512
.data
ClassName db "SimpleWinClass",0
AppName  db "Our Main Window",0
MenuName db "FirstMenu",0
ofn   OPENFILENAME <>
FilterString db "All Files",0,"*.*",0
             db "Text Files",0,"*.txt",0,0
buffer db MAXSIZE dup(0)
OurTitle db "-=Our First Open File Dialog Box=-: Choose the file to
open",0
FullPathName db "The Full Filename with Path is: ",0
FullName  db "The Filename is: ",0
ExtensionName db "The Extension is: ",0
OutputString db OUTPUTSIZE dup(0)
CrLf db 0Dh,0Ah,0
.data?
hInstance HINSTANCE ?
CommandLine LPSTR ?
.code
start:
    invoke GetModuleHandle, NULL
    mov    hInstance,eax
    invoke GetCommandLine
    mov CommandLine,eax
    invoke WinMain, hInstance,NULL,CommandLine, SW_SHOWDEFAULT
    invoke ExitProcess,eax
WinMain proc
hInst:HINSTANCE,hPrevInst:HINSTANCE,CmdLine:LPSTR,CmdShow:DWORD
    LOCAL wc:WNDCLASSEX
    LOCAL msg:MSG
    LOCAL hwnd:HWND
    mov   wc.cbSize,SIZEOF WNDCLASSEX
    mov   wc.style, CS_HREDRAW or CS_VREDRAW
    mov   wc.lpfnWndProc, OFFSET WndProc
    mov   wc.cbClsExtra,NULL
```

```
    mov   wc.cbWndExtra,NULL
    push  hInst
    pop   wc.hInstance
    mov   wc.hbrBackground,COLOR_WINDOW+1
    mov   wc.lpszMenuName,OFFSET MenuName
    mov   wc.lpszClassName,OFFSET ClassName
    invoke LoadIcon,NULL,IDI_APPLICATION
    mov   wc.hIcon,eax
    mov   wc.hIconSm,eax
    invoke LoadCursor,NULL,IDC_ARROW
    mov   wc.hCursor,eax
    invoke RegisterClassEx, addr wc
    invoke CreateWindowEx, WS_EX_CLIENTEDGE,\
                           ADDR ClassName,\
                           ADDR AppName,\
                           WS_OVERLAPPEDWINDOW,\
                           CW_USEDEFAULT,\
                           CW_USEDEFAULT,\
                           300,\
                           200,\
                           NULL,\
                           NULL,\
                           hInst,\
                           NULL
    mov   hwnd,eax
    invoke ShowWindow, hwnd,SW_SHOWNORMAL
    invoke UpdateWindow, hwnd
    .WHILE TRUE
        invoke GetMessage, ADDR msg,NULL,0,0
        .BREAK .IF (!eax)
        invoke TranslateMessage, ADDR msg
        invoke DispatchMessage, ADDR msg
    .ENDW
    mov     eax,msg.wParam
    ret
WinMain endp
WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
    .IF uMsg==WM_DESTROY
        invoke PostQuitMessage,NULL
    .ELSEIF uMsg==WM_COMMAND
        mov eax,wParam
        .if ax==IDM_OPEN
            mov ofn.lStructSize,SIZEOF ofn
            push hWnd
            pop  ofn.hwndOwner
            push hInstance
            pop  ofn.hInstance
            mov  ofn.lpstrFilter, OFFSET FilterString
            mov  ofn.lpstrFile, OFFSET buffer
            mov  ofn.nMaxFile,MAXSIZE
            mov  ofn.Flags, OFN_FILEMUSTEXIST or \
                            OFN_PATHMUSTEXIST or \
```

```
                            OFN_LONGNAMES or \
                            OFN_EXPLORER or \
                            OFN_HIDEREADONLY
            mov  ofn.lpstrTitle, OFFSET OurTitle
            invoke GetOpenFileName, ADDR ofn
            .if eax==TRUE
                invoke lstrcat,offset OutputString,\
                               OFFSET FullPathName
                invoke lstrcat,offset OutputString,ofn.lpstrFile
                invoke lstrcat,offset OutputString,offset CrLf
                invoke lstrcat,offset OutputString,offset FullName
                mov  eax,ofn.lpstrFile
                push ebx
                xor  ebx,ebx
                mov  bx,ofn.nFileOffset
                add  eax,ebx
                pop  ebx
                invoke lstrcat,offset OutputString,eax
                invoke lstrcat,offset OutputString,offset CrLf
                invoke lstrcat,offset OutputString,\
                               offset ExtensionName
                mov  eax,ofn.lpstrFile
                push ebx
                xor ebx,ebx
                mov  bx,ofn.nFileExtension
                add eax,ebx
                pop ebx
                invoke lstrcat,offset OutputString,eax
                invoke MessageBox,hWnd,\
                                  OFFSET OutputString,\
                                  ADDR AppName,\
                                  MB_OK
                invoke RtlZeroMemory,offset OutputString,OUTPUTSIZE
            .endif
        .else
            invoke DestroyWindow, hWnd
        .endif
    .ELSE
        invoke DefWindowProc,hWnd,uMsg,wParam,lParam
        ret
    .ENDIF
    xor    eax,eax
    ret
WndProc endp
 end start
```

حال به بررسی کد برنامه می پردازیم.

```
mov ofn.lStructSize,SIZEOF ofn
push hWnd
pop  ofn.hwndOwner
push hInstance
pop  ofn.hInstance
```

اعضا معمول رکورد ofn را مقدار دهی می کنیم.

```
mov  ofn.lpstrFilter, OFFSET FilterString
```

lpstrFilter را مقدار دهی می کنیم.

```
FilterString db "All Files",0,"*.*",0
             db "Text Files",0,"*.txt",0,0
```

دقت داشته باشید که هر چهار رشته ، مختوم به صفر هستند. رشته اول توضیح رشته دوم و رشته سوم توضیح رشته چهارم است. برای مشخص کردن الگوی فیلترها از عبارات مختلفی می توان استفاده کرد که در این مثال txt.* و*.* هستند. در آخر رشته فیلتر هم باید صفر را وارد کنید. این نکته را فراموش نکنید زیرا در غیر این صورت ممکن است از دیالوگ باکس شما رفتار غیر عادی سر بزند.

```
mov  ofn.lpstrFile, OFFSET buffer
mov  ofn.nMaxFile,MAXSIZE
```

بافری را برای ذخیره نام و مسیر فایل انتخاب شده تعیین کرده و در سطر بعد حداکثر طول آنرا مشخص می کنیم.

```
mov  ofn.Flags, OFN_FILEMUSTEXIST or \
                OFN_PATHMUSTEXIST or \
                OFN_LONGNAMES or \
                OFN_EXPLORER or \
                OFN_HIDEREADONLY
```

فلگها خصوصیات و نحوه عملکرد دیالوگ باکسها را تعیین می کنند. OFN_PATHMUSTEXIST و OFN_FILEMUSTEXIST اعلام می کنند که نام فایل و مسیری که کاربر را تایپ می کند حتما باید وجود داشته باشند. OFN_LONGNAMES معین می‌کند که دیالوگ باکس نام فایلها را بصورت کامل نمایش دهد.

OFN_EXPLORER مشخص می کند که شکل ظاهری دیالوگ باکس مانند explorer باشد.

تعداد این فلگ ها بسیار زیاد است. برای اطلاعات بیشتر در مورد آنها می توانید به مراجع API رجوع کنید.

```
mov  ofn.lpstrTitle, OFFSET OurTitle
```

عنوان دیالوگ باکس را مشخص می کنیم.

```
invoke GetOpenFileName, ADDR ofn
```

تابع GetOpenFileName را فراخوانی کرده و اشاره گر به رکورد ofn را به عنوان پارامتر ورودی به آن می فرستیم. این تابع مقداری را باز نمی گرداند مگر اینکه کاربر فایلی را انتخاب کند یا دکمه Cancel را فشار دهد و یا پنجره را ببندد. اگر فایلی توسط کاربر انتخاب شود مقدار True در eax قرار خواهد گرفت در غیر اینصورت مقدار EAX ، False خواهد بود.

```
.if eax==TRUE
    invoke lstrcat,offset OutputString,\
                   OFFSET FullPathName
    invoke lstrcat,offset OutputString,ofn.lpstrFile
    invoke lstrcat,offset OutputString,offset CrLf
    invoke lstrcat,offset OutputString,offset FullName
```

هنگامی که کاربر فایلی را انتخاب می کند ، ما رشته خروجی را برای نمایش در MessageBox آماده می کنیم. به این منظور بلوکی از حافظه را به متغیر OutputString اختصاص می دهیم و از یکی از توابع API به نام lstrcat برای به هم چسباندن رشته های خروجی استفاده می کنیم. برای قرار دادن رشته ها در سطر های مختلف نیز می توانید از کاراکترهای جدا کننده خط یا Crlf استفاده کنید که سطر فعلی را پایان داده و به ابتدای سطر بعد می رود. همانطور که در ابتدای برنامه نیز مشاهده می کنید این عبارت حاصل در کنار هم قرار گرفتن کاراکترهای 13 , 10 اسکی است.

```
mov  eax,ofn.lpstrFile
push ebx
xor  ebx,ebx
mov  bx,ofn.nFileOffset
add  eax,ebx
```

```
pop  ebx
invoke lstrcat,offset OutputString,eax
```

خطوط بالا نیاز به توضیح دارند. nFileOffset حاوی اندیسی از رشته ofn.lpstrFile است ولی نمی توانیم آن را با ofn.lpstrFile جمع کنیم زیرا nFileOffset از نوع WORD است ولی lpstrFile از نوع DWORD می باشد. برای رفع این مشکل ابتدا مقدار nFileOffset را در Low word از ebx قرار داده و سپس آنرا با lpstrFile جمع می کنیم.

```
invoke MessageBox,hWnd,\
                  OFFSET OutputString,\
                  ADDR AppName,\
                  MB_OK
```

حال متن ایجاد شده در مرحله قبل را در یک MessageBox قرار داده و آنرا نمایش می دهیم.

```
invoke RtlZerolMemory,offset OutputString,OUTPUTSIZE
```

برای اینکه بتوانیم رشته دیگری را در متغیر OutputString قرار دهیم باید آنرا پاک کنیم. برای این کار از تابع RtlZeroMemory استفاده می کنیم.

مدیریت حافظه و فایل

کدها و فایل‌های مربوط به این قسمت در CD ضمیمه موجود می‌باشد

SourceCodes\Asm32\Chapter11

مدیریت حافظه در Win32 از دیدگاه برنامه ها بسیار ساده است. هر Process یک فضای حافظه ٤ گیگابایتی در اختیار دارد که از مدل Flat استفاده می کند. در این مدل کلیه ثباتهای سگمنت به آدرس شروع حافظه برنامه اشاره می کنند. هر برنامه می تواند بدون تغییر مقدار ثباتها به هر نقطه از فضای حافظه خود دسترسی داشته باشد. این قابلیت مدیریت حافظه را آسانتر می سازد. اشاره گرهای far و near نیز دیگر وجود ندارند. در Win16 دو نوع اصلی از توابع حافظه وجود داشتند که عبارتند از Global و Local. توابع Global برای برقراری ارتباط با Segment های دیگر به کار می روند به همین دلیل آنها را جزء توابع far به حساب می آورند. توابع Local برای برقراری ارتباط با heap محلی Process استفاده می شوند به همین دلیل آنها جز توابع near محسوب می‌شوند. ولی در Win32 عملکرد این دو نوع یکسان است و اگر LocalAlloc یا GlobalAlloc را صدا بزنید نتیجه یکسانی خواهید گرفت.

مراحل اختصاص حافظه و استفاده از آن به صورت زیر است.

۱- یک بلوک از حافظه را با فراخوانی تابع GlobalAlloc رزرو می کنید. این تابع شماره دسترسی به بلوک حافظه مورد نظر را بر می گرداند.

۲- توسط تابع GlobalLock بلوک مورد نظر را Lock می کنیم. این تابع شماره دسترسی حافظه را به عنوان پارامتر ورودی می گیرد و یک اشاره گر به آن قسمت از حافظه را به شما باز می گرداند.

۳- می توانید از یک اشاره گر برای خواندن و نوشتن در حافظه استفاده کنید.

٤- با استفاده از تابع GlobalUnlock بلوک حافظه مورد نظر را Unlock می کنیم. این تابع اشاره گر مربوط به این قسمت از حافظه را باطل می کند.

٥- توسط تابع GlobalFree بلوک حافظه را برای استفاده های بعدی آزاد می کنیم. این تابع شماره دسترسی به بلوک حافظه را به عنوان پارامتر ورودی دریافت می کند.

می توانید روش بالا را با استفاده از فلگ GMEM_FIXED در تابع GlobalAlloc باز هم ساده تر کنید. با استفاده از این فلگ مقدار برگشتی از این تابع اشاره گری به بلوک حافظه مورد نظر خواهد

بود و دیگر نیازی به فراخوانی تابع GlobalLock نیست. می توانید با فرستادن این اشاره گر به تابع GlobalFree بدون فراخوانی تابع GlobalUnlock حافظه را آزاد کنید. ما در این بخش از روش سنتی برای کار با حافظه استفاده کرده ایم زیرا ممکن است در مواقع خواندن کدهای برنامه های دیگر با آن مواجه شوید.

عملیات ورودی و خروجی فایل در Win32 ظاهر بهتری نسبت به Dos دارد ولی مراحل کار یکسان است فقط باید وقفه ها را به تابع API تبدیل کنید. در زیر مراحل این کار را در Win32 مشاهده می کنید.

۱- باز کردن یا ایجاد فایل توسط تابع CreateFile. این تابع بسیار انعطاف پذیر است. شما علاوه بر فایل می توانید از آن برای کار با پورتها ، کنسول Dos یا pipe نیز استفاده کنید. در صورت موفقیت ، این تابع شماره دسترسی به فایل یا ابزار مورد نظر را به شما بر می گرداند. با استفاده از این شماره می توانید آنها را کنترل کرده و دستورات مورد نظر خود را اجرا کنید. همچنین می توانید با استفاده از تابع SetFilePointer اشاره گر فایل را به نقطه دلخواهی از آن منتقل کنید.

۲- اعمال خواندن و نوشتن را با فراخوانی توابع WriteFile و ReadFile انجام می دهیم. این توابع توانایی انتقال اطلاعات از فایل به حافظه یا برعکس را دارند.

۳- بستن فایل با استفاده از تابع CloseFile انجام می شود. این تابع شماره دسترسی به فایل را به عنوان پارامتر ورودی دریافت می کند.

حال به سراغ مثال این بخش می رویم.

این برنامه یک دیالوگ OpenFile را نمایش داده و به کاربر اجازه می دهد که یک فایل متنی را برای نمایش در کنترل Edit انتخاب کند. کاربر می تواند محتویات فایل مورد نظر را تغییر داده و با انتخاب گزینه Save آنرا در فایل جدیدی ذخیره کند.

```
.386
.model flat,stdcall
option casemap:none
WinMain proto :DWORD,:DWORD,:DWORD,:DWORD
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
include \masm32\include\comdlg32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\comdlg32.lib
.const
IDM_OPEN equ 1
IDM_SAVE equ 2
IDM_EXIT equ 3
MAXSIZE equ 260
MEMSIZE equ 65535
; ID of the edit        EditID equ 1
control
.data
ClassName db "Win32ASMEditClass",0
AppName  db "Win32 ASM Edit",0
EditClass db "edit",0
MenuName db "FirstMenu",0
ofn   OPENFILENAME <>
FilterString db "All Files",0,"*.*",0
             db "Text Files",0,"*.txt",0,0
buffer db MAXSIZE dup(0)
.data?
hInstance HINSTANCE ?
CommandLine LPSTR ?
; Handle to the edit      hwndEdit HWND ?
control
; File handle     hFile HANDLE ?
;handle to the    hMemory HANDLE ?
allocated memory block
;pointer to the     pMemory DWORD ?
allocated memory block
; number of bytes        SizeReadWrite DWORD ?
actually read or write
.code
start:
    invoke GetModuleHandle, NULL
    mov    hInstance,eax
    invoke GetCommandLine
    mov CommandLine,eax
    invoke WinMain, hInstance,NULL,CommandLine, SW_SHOWDEFAULT
    invoke ExitProcess,eax
WinMain proc
hInst:HINSTANCE,hPrevInst:HINSTANCE,CmdLine:LPSTR,CmdShow:SDWORD
    LOCAL wc:WNDCLASSEX
```

```
    LOCAL msg:MSG
    LOCAL hwnd:HWND
    mov   wc.cbSize,SIZEOF WNDCLASSEX
    mov   wc.style, CS_HREDRAW or CS_VREDRAW
    mov   wc.lpfnWndProc, OFFSET WndProc
    mov   wc.cbClsExtra,NULL
    mov   wc.cbWndExtra,NULL
    push  hInst
    pop   wc.hInstance
    mov   wc.hbrBackground,COLOR_WINDOW+1
    mov   wc.lpszMenuName,OFFSET MenuName
    mov   wc.lpszClassName,OFFSET ClassName
    invoke LoadIcon,NULL,IDI_APPLICATION
    mov   wc.hIcon,eax
    mov   wc.hIconSm,eax
    invoke LoadCursor,NULL,IDC_ARROW
    mov   wc.hCursor,eax
    invoke RegisterClassEx, addr wc
    invoke CreateWindowEx, WS_EX_CLIENTEDGE,\
                           ADDR ClassName,\
                           ADDR AppName,\
                           WS_OVERLAPPEDWINDOW,\
                           CW_USEDEFAULT,\
                           CW_USEDEFAULT,\
                           300,\
                           200,\
                           NULL,\
                           NULL,\
                           hInst,\
                           NULL
    mov   hwnd,eax
    invoke ShowWindow, hwnd,SW_SHOWNORMAL
    invoke UpdateWindow, hwnd
    .WHILE TRUE
        invoke GetMessage, ADDR msg,NULL,0,0
        .BREAK .IF (!eax)
        invoke TranslateMessage, ADDR msg
        invoke DispatchMessage, ADDR msg
    .ENDW
    mov     eax,msg.wParam
    ret
WinMain endp
WndProc proc uses ebx hWnd:HWND, uMsg:UINT, wParam:WPARAM,
lParam:LPARAM
    .IF uMsg==WM_CREATE
        invoke CreateWindowEx,NULL,\
                              ADDR EditClass,\
                              NULL,\
                              WS_VISIBLE or WS_CHILD or ES_LEFT or
                              ES_MULTILINE or\
                              ES_AUTOHSCROLL or ES_AUTOVSCROLL,0,\
                              0,\
```

```
                                0,\
                                0,\
                                hWnd,\
                                EditID,\
                                hInstance,\
                                NULL
        mov hwndEdit,eax
        invoke SetFocus,hwndEdit
;=========================================================
;         Initialize the members of OPENFILENAME structure
;=========================================================
        mov ofn.lStructSize,SIZEOF ofn
        push hWnd
        pop  ofn.hWndOwner
        push hInstance
        pop  ofn.hInstance
        mov  ofn.lpstrFilter, OFFSET FilterString
        mov  ofn.lpstrFile, OFFSET buffer
        mov  ofn.nMaxFile,MAXSIZE
    .ELSEIF uMsg==WM_SIZE
        mov eax,lParam
        mov edx,eax
        shr edx,16
        and eax,0ffffh
        invoke MoveWindow,hwndEdit,0,0,eax,edx,TRUE
    .ELSEIF uMsg==WM_DESTROY
        invoke PostQuitMessage,NULL
    .ELSEIF uMsg==WM_COMMAND
        mov eax,wParam
        .if lParam==0
            .if ax==IDM_OPEN
                mov  ofn.Flags,  OFN_FILEMUSTEXIST or \
                                 OFN_PATHMUSTEXIST or \
                                 OFN_LONGNAMES or\
                                 OFN_EXPLORER or \
                                 OFN_HIDEREADONLY
                invoke GetOpenFileName, ADDR ofn
                .if eax==TRUE
                    invoke CreateFile, ADDR buffer,\
                                       GENERIC_READ or GENERIC_WRITE
                                       ,\
                                       FILE_SHARE_READ or
                                       FILE_SHARE_WRITE,\
                                       NULL,\
                                       OPEN_EXISTING,\
                                       FILE_ATTRIBUTE_ARCHIVE,\
                                       NULL
                    mov hFile,eax
                    invoke GlobalAlloc,GMEM_MOVEABLE or \
                                         GMEM_ZEROINIT, MEMSIZE
                    mov  hMemory,eax
                    invoke GlobalLock,hMemory
```

```
            mov  pMemory,eax
            invoke ReadFile, hFile,\
                             pMemory,\
                             MEMSIZE-1,\
                             ADDR SizeReadWrite,\
                             NULL
            invoke SendMessage, hwndEdit,\
                                WM_SETTEXT,\
                                NULL,\
                                pMemory
            invoke CloseHandle,hFile
            invoke GlobalUnlock,pMemory
            invoke GlobalFree,hMemory
        .endif
        invoke SetFocus,hwndEdit
    .elseif ax==IDM_SAVE
        mov ofn.Flags,OFN_LONGNAMES or\
                      OFN_EXPLORER or OFN_HIDEREADONLY
        invoke GetSaveFileName, ADDR ofn
            .if eax==TRUE
                invoke CreateFile,\
                                  ADDR buffer,\
                                  GENERIC_READ or
                                  GENERIC_WRITE ,\
                                  FILE_SHARE_READ or
                                  FILE_SHARE_WRITE,\
                                  NULL,\
                                  CREATE_NEW,\
                                  FILE_ATTRIBUTE_ARCHIVE,\
                                  NULL
                mov hFile,eax
                invoke GlobalAlloc,\
                                  GMEM_MOVEABLE or \
                                  GMEM_ZEROINIT,\
                                  MEMSIZE
                mov  hMemory,eax
                invoke GlobalLock,hMemory
                mov  pMemory,eax
                invoke SendMessage,hwndEdit,\
                                   WM_GETTEXT,\
                                   MEMSIZE-1,\
                                   pMemory
                invoke WriteFile, hFile,\
                                  pMemory,\
                                  eax,\
                                  ADDR SizeReadWrite,\
                                  NULL
                invoke CloseHandle,hFile
                invoke GlobalUnlock,pMemory
                invoke GlobalFree,hMemory
            .endif
            invoke SetFocus,hwndEdit
```

```
                .else
                    invoke DestroyWindow, hWnd
                .endif
            .endif
        .ELSE
            invoke DefWindowProc,hWnd,uMsg,wParam,lParam
            ret
.ENDIF
xor    eax,eax
ret
WndProc endp
end start
```

حال به بررسی کدهای برنامه می پردازیم.

```
    invoke CreateWindowEx, WS_EX_CLIENTEDGE,\
                           ADDR ClassName,\
                           ADDR AppName,\
                           WS_OVERLAPPEDWINDOW,\
                           CW_USEDEFAULT,\
                           CW_USEDEFAULT,\
                           300,\
                           200,\
                           NULL,\
                           NULL,\
                           hInst,\
                           NULL
     mov hwndEdit,eax
```

در بخش WM_CREATE یک کنترل Edit ایجاد می کنیم. متذکر می شویم که کلیه پارامتر های مربوط به مختصات و ابعاد کنترل را صفر قرار می دهیم و در ادامه آنها را به منظور هماهنگی با ابعاد منطقه کاری پنجره ، مقدار دهی می کنیم. در این حالت دیگر نیازی به فراخوانی تابع ShowWindow برای نمایش کنترل نیست. زیرا در تابع CreateWindowEx از مدل WS_VISIBLE استفاده کرده ایم. می توانید از این روش برای ایجاد و نمایش پنجره های اصلی برنامه نیز استفاده کنید.

```
;=========================================================
;        Initialize the members of OPENFILENAME structure
;=========================================================
        mov ofn.lStructSize,SIZEOF ofn
        push hWnd
        pop  ofn.hWndOwner
        push hInstance
        pop  ofn.hInstance
```

```
mov  ofn.lpstrFilter, OFFSET FilterString
mov  ofn.lpstrFile, OFFSET buffer
mov  ofn.nMaxFile,MAXSIZE
```

بعد از ایجاد کنترل Edit نوبت به مقدار دهی اعضا ofn می رسد. چون می خواهیم از آن در دیالوگ Save نیز استفاده کنیم. در ابتدا فقط اعضا مشترک GetSaveFileName و GetOpenFileName را مقدار دهی می کنیم. WM_CREATE محل مناسبی برای انجام کارهایی مثل این مورد است که در سرتاسر برنامه ، تنها یک بار باید انجام شوند.

```
.ELSEIF uMsg==WM_SIZE
    mov eax,lParam
    mov edx,eax
    shr edx,16
    and eax,0ffffh
    invoke MoveWindow,hwndEdit,0,0,eax,edx,TRUE
```

وقتی اندازه پنجره اصلی تغییر کند ، ما پیغام WM_SIZE را دریافت خواهیم کرد. این پیغام را هنگامی که پنجره برای اولین بار ایجاد می شود نیز دریافت می کنیم. برای دریافت این پیغام ساختار کلاس پنجره باید از مدلهای CS_VREDRAW و CS_HREDRAW استفاده کرده باشد. ما از این فرصت برای هماهنگی ابعاد کنترل Edit با منطقه کاری پنجره پدر استفاده می کنیم. ابتدا باید پهنا و ارتفاع منطقه کاری پنجره پدر را بدانیم. این اطلاعات را از lParam دریافت می کنیم. بخش Low word آن شامل عرض منطقه کاری می باشد. سپس از این اطلاعات برای تغییر اندازه کنترل Edit توسط تابع MoveWindow استفاده می کنیم. این تابع علاوه بر تغییر دادن ابعاد پنجره، می‌تواند مختصات آن را نیز در صفحه تعیین کند.

```
.if ax==IDM_OPEN
    mov  ofn.Flags, OFN_FILEMUSTEXIST or \
                    OFN_PATHMUSTEXIST or \
                     OFN_LONGNAMES or\
                    OFN_EXPLORER or OFN_HIDEREADONLY
    invoke GetOpenFileName, ADDR ofn
```

وقتی کاربر گزینه Open از منوی File را انتخاب کرد ما بخش flag از ساختار ofn را مقدار دهی کرده و سپس تابع GetOpenFileName را برای نمایش دیالوگ OpenFile فراخوانی می کنیم.

```
.if eax==TRUE
    invoke CreateFile, ADDR buffer,\
                GENERIC_READ or \
          GENERIC_WRITE ,\
          FILE_SHARE_READ or  FILE_SHARE_WRITE,\
          NULL,OPEN_EXISTING,FILE_ATTRIBUTE_ARCHIVE,\

          NULL
    mov hFile,eax
```

پس از انتخاب فایل توسط کاربر ، تابع CreateFile را برای باز کردن آن فراخوانی کرده و مشخص می کنیم که این تابع باید فایل مورد نظر را به منظور خواندن و نوشتن باز کند . پس از اینکه فایل باز شد ، این تابع یک شماره دسترسی به فایل باز شده را بر می گرداند. ما آنرا در یک متغییر عمومی برای استفاده های بعدی ذخیره می کنیم . پیش تعریف این تابع بصورت زیر است :

```
CreateFile proto lpFileName:DWORD,\
                        dwDesiredAccess:DWORD,\
                        dwShareMode:DWORD,\
                        lpSecurityAttributes:DWORD,\
                        dwCreationDistribution:DWORD\,
                        dwFlagsAndAttributes:DWORD\,
                        hTemplateFile:DWORD
```

dwDesiredAccess : مشخص می کند که چه عملیاتی قرار است بر روی فایل انجام شود .

- **0** : فایل را بر اساس مشخصات خودش باز می کند (Attributes) و اجازه Write Read, را از مشخصات خود فایل دریافت می کند .
- **GENERIC_READ** : فایل را برای خواندن باز می کند .
- **GENERIC_WRITE** : فایل را برای نوشتن باز می کند .

dwShareMode : مشخص می کند که چه اعمالی می تواند توسط دیگر Process ها بر روی این فایل باز شده انجام شود .

- **0** : Process های دیگر به این فایل هیچگونه دسترسی ندارند .
- **FILE_SHARE_READ** : اجازه خواندن به دیگر Process ها داده می شود .

- **FILE_SHARE_WRITE** : اجازه نوشتن روی فایل به دیگر Process ها داده می شود .
- **lpSecurityAttributes** : مشخصات امنیتی فایل را تعیین میکند که در Win 9X اهمیتی ندارد .
- **dwCreationDistribution**: نحوه عملکرد تابع CreateFile را هنگامی که فایل مورد نظر قبلا ایجاد شده باشد یا نباشد مشخص می کند .
- **CREATE_NEW** : یک فایل جدید باز می کند و اگر فایل مورد نظر قبلا ایجاد شده باشد تابع با شکست مواجه می شود .
- **CREATE_ALWAYS** : یک فایل جدید باز می کند و اگر فایل قبلا ایجاد شده باشد روی آن بازنویسی می کند .
- **OPEN_EXISTING** : فایل معینی را باز می کند و اگر فایل مورد نظر قبلا ایجاد نشده باشد با شکست مواجه خواهد شد .
- **OPEN_ALWAYS** : فایل را باز می کند و اگر فایل مورد نظر قبلا ایجاد شده باشد آنرا باز می کند و در غیر اینصورت همانند CREATE_NEW عمل کرده و یک فایل جدید ایجاد می کند .
- **TRUNCATE_EXISTING** : فایل را باز کرده و محتویات آنرا از بین می برد و سایز آنرا صفر می کند . اگر فایل مورد نظر قبلا ایجاد نشده باشد تابع با شکست مواجه خواهد شد .
- **dwFlagsAndAttributes** : مشخصات فایل مانند ReadOnly , Hidden و ... را مشخص می کند .
- **FILE_ATTRIBUTE_ARCHIVE** : فایل به صورت Archive در خواهد آمد یعنی برای تهیه نسخه پشتیبان و پاک شدن مارک می شود .
- **FILE_ATTRIBUTE_COMPRESSED** : فایل یا شاخه بصورت فشرده است. در مورد فایل این به معنی فشرده بودن تمام اطلاعات موجود در آن است و برای شاخه یعنی فشرده سازی حالت پیش فرض برای ایجاد فایلها و زیر شاخه های جدید می باشد .
- **FILE_ATTRIBUTE_NORMAL** : فایل هیچ حالت خاصی ندارد .
- **FILE_ATTRIBUTE_HIDDEN** : فایل پنهان است و در لیستهای معمول فایلها و شاخه ها نمایش داده نمی شود .

- **FILE_ATTRIBUTE_READONLY** : فایل به صورت فقط خواندنی است و نمی توان روی آن چیزی نوشت یا آنرا پاک کرد .
- **FILE_ATTRIBVTE_SYSTEM** : این فایل توسط سیستم عامل مورد استفاده قرار می‌گیرد .

```
invoke GlobalAlloc,GMEM_MOVEABLE or GMEM_ZEROINIT,MEMSIZE

mov  hMemory,eax
invoke GlobalLock,hMemory
mov  pMemory,eax
```

هنگامی که فایل باز شد یک بلوک از حافظه را برای استفاده توسط توابع ReadFile و WriteFile در نظر می گیریم : فلگ GMEM_MOVABLE به ویندوز اجازه جابجایی بلوک حافظه را برای ایجاد یکپارچگی در حافظه می دهد . GMEM_ZEROINIT مشخص میکند که تابع باید بلوک حافظه را با صفر مقدار دهی اولیه کند . وقتی GlobalAlloc با موفقیت وظیفه خود را انجام دهد eax حاوی شماره دسترسی به بلوک حافظه اختصاص داده شده خواهد بود . در مرحله بعد این شماره را به تابع Globallock می فرستیم که اشاره گری به بلوک حافظه را بر می گرداند .

```
invoke ReadFile, hFile,\
                 pMemory,\
                 MEMSIZE-1,\
                 ADDR SizeReadWrite,\
                 NULL
invoke SendMessage,\
                    hwndEdit,\
                    WM_SETTEXT,\
                    NULL,\
                    pMemory
```

وقتی بلوک حافظه آماده استفاده شد از تابع ReedFile برای خواندن اطلاعات استفاده می کنیم . وقتی فایلی باز یا ایجاد می شود ، اشاره گر آن در Offset صفر است . اولین پارامتر ورودی تابع ReadFile شماره دسترسی به فایل مورد نظر است . دومین پارامتر اشاره گری به بلوک حافظه ای است که برای نگهداری اطلاعات در نظر گرفته ایم . پارامتر بعدی تعداد بایتهایی است که باید از فایل خوانده شوند و پارامتر چهارم هم آدرس متغییری است که تعداد واقعی بایتهای خوانده شده از فایل در آن ذخیره خواهد شد .

پس از خواندن فایل و پر کردن بلوک حافظه اطلاعات خوانده شده را توسط پیغام WM_SETTEXTبه کنترل Edit ارسال می کنیم . برای این کار آدرس حافظه رزرو شده را در lParam قرار می دهیم . پس از ارسال این پیغام کنترل Edit محتویات فایل مورد نظر را نمایش خواهد داد .

```
    invoke CloseHandle,hFile
    invoke GlobalUnlock,pMemory
    invoke GlobalFree,hMemory
.endif
```

پس از خواندن اطلاعات از فایل ، دیگر نیازی نیست که فایل را باز نگه داریم زیرا هدف ما ذخیره سازی اطلاعات در یک فایل جدید است نه در فایل قبلی . پس از فراخوانی تابع CloseHandle و فرستادن شماره دسترسی فایل ، آنرا می بندیم . در مرحله بعد بلوک حافظه را آزاد می کنیم . در واقع نیازی به آزاد سازی حافظه ندارید و می توانید از آن برای عملیات Save نیز استفاده کنید . ولی در اینجا ما حافظه را آزاد می کنیم .

```
invoke SetFocus,hwndEdit
```

وقتی دیالوگ Open File روی صفحه نمایش داده می شود ، فوکوس را در اختیار می گیرد ولی هنگامی که بسته می شود ، فوکوس باید به کنترل Edit باز گردد . در این مرحله عملیات خواندن از فایل به اتمام رسیده است و کاربر می تواند محتویات فایل را تغییر داده و در صورت نیاز با انتخاب گزینه Save از منوی File ، آنها را در فایل دیگری ذخیره کند. ایجاد دیالوگ Save تفاوت چندانی با دیالوگ Open File ندارد. در واقع تفاوت اصلی در نام توابع مورد نیاز است. می توانید به غیر از Flag از تمام اعضا ofn در تابع GetSaveFileName نیز استفاده کنید.

```
mov ofn.Flags, OFN_LONGNAMES or\
               OFN_EXPLORER or \
               OFN_HIDEREADONLY
```

در این قسمت می خواهیم یک فایل جدید ایجاد کنیم. پس OFN_PATHMUSTEXIST و OFN_FILEMUSTEXIST باید از فلگها کنار گذاشته شوند در غیر این صورت این دیالوگ باکس به ما اجازه ایجاد فایل جدید را نخواهد داد. پارامتر dwCreationDistribution را نیز باید به CREATE_NEW تغییر دهیم. بقیه کد برنامه به جز دو سطر پایین با بخش Open File مشترک است.

```
invoke SendMessage,hwndEdit,\
                   WM_GETTEXT,\
                   MEMSIZE-1,\
                   pMemory
invoke WriteFile, hFile,\
                  pMemory,\
                  eax,\
                  ADDR SizeReadWrite,\
                  NULL
```

برای دریافت محتویات کنترل Edit ، پیغام WM_GETTEXT را به آن می فرستیم که محتویات آن را در بلوک حافظه مشخص شده قرار می دهد. مقدار برگشتی در eax طول داده های ذخیره شده در بافر را مشخص می کند. پس از دریافت محتویات کنترل Edit ، آنها را در فایل جدید می نویسیم.

فایلهای نگاشت شده به حافظه

کدها و فایل‌های مربوط به این قسمت در CD ضمیمه موجود می‌باشد

SourceCodes\Asm32\Chapter12

اگر مثال بخش قبل را به خوبی بررسی کرده باشید ، خواهید دید که یک نقطه ضعف جدی دارد و آن اینست که اگر فایل مورد نظر بزرگتر از بلوک حافظه رزرو شده باشد چه اتفاقی خواهد افتاد؟ پاسخ سوال اینست که اطلاعات باید به دفعات متوالی از فایل خوانده شود تا به انتهای فایل برسیم. خیلی خوب بود اگر می توانستیم کل فایل را در حافظه ذخیره کنیم ولی این کار باعث اتلاف منابع سیستم می شود. برای جلوگیری از این امر ، از نگاشت فایل استفاده می کنیم. با استفاده از این تکنیک می‌توانید تصور کنید که تمام فایل قبلا به حافظه بارگذاری شده است و از یک اشاره گر به حافظه برای خواندن و نوشتن اطلاعات در فایل استفاده کنید و دیگر نیازی به استفاده از توابع API مربوط به حافظه و ورودی خروجی فایل ندارید. همچنین از نگاشت فایل به عنوان ابزاری جهت تبادل داده ها بین Process های مختلف استفاده می شود. در این مورد نگاشت فایل شبیه به اختصاص بلوکی از حافظه است که تمام Process ها می توانند به آن دسترسی داشته باشند. عمل تبادل داده ها بین Process های مختلف ، کاری بسیار حساس و ظریف است و به سادگی نمی توان با آن برخورد کرد.

در عمل در طراحی و پیاده سازی موتورهای جستجو از نگاشت فایل استفاده های زیادی می شود. در واقع بارگذار برنامه های ویندوز (PE Loader) نیز از نگاشت فایل برای بارگزاری فایلهای اجرایی در حافظه استفاده می کند. البته محدودیتهایی هم برای نگاشت فایل وجود دارد. اول اینکه وقتی فایلی را در حافظه نگاشت می کنید نمی توانید اندازه آن را تغییر دهید به همین دلیل معمولاً از نگاشت فایل برای کار با فایلهای فقط خواندنی و کارهایی که تاثیری در اندازه فایل ندارند استفاده می شود. این بدان معنا نیست که نمی توانید از نگاشت برای کارهایی استفاده کنید که اندازه فایل را تغییر می دهند. برای انجام اینگونه اعمال باید ابتدا اندازه جدید فایل را تخمین بزنید و فایلی با اندازه جدید در حافظه ایجاد کنید.

در زیر مراحل استفاده و نگاشت فایل را مشاهده می کنید.

۱- تابع CreateFile را برای باز کردن فایل مورد نظر فراخوانی می کنیم.

۲- تابع CreateFileMapping را به همراه شماره دسترسی به فایل فراخوانی می کنیم.

۳- تابع MapViewOfFile را برای نگاشت قسمت انتخابی یا کل فایل ، در حافظه فراخوانی می کنیم. این تابع اشاره گری به اولین بایت فایل نگاشت شده را بر می گرداند.

٤- از این اشاره گر برای خواندن و نوشتن در فایل استفاده می کنیم.

٥- تابع UnMapViewOfFile را برای پاک کردن و از بین بردن نگاشت فایل در حافظه فراخوانی می کنیم.

٦- تابع CloseHandle را برای بستن فایل نگاشته شده فراخوانی کرده و اشاره گر به نگاشت را به عنوان پارامتر ورودی به آن می فرستیم.

۷- برای بستن فایل اصلی از تابع CloseHandle به همراه اشاره گر فایل استفاده می کنیم.

حال به سراغ مثال این بخش می رویم.

این برنامه به شما اجازه می دهد که فایلی را از طریق دیالوگ OpenFile انتخاب کرده و آن را با استفاده از نگاشت فایل باز کنید. در صورت موفقیت آمیز بودن عمل فوق عنوان پنجره به نام فایل تغییر پیدا خواهد کرد. اکنون می توانید با انتخاب آیتم Save as از منوی File ، فایل مورد نظر را با نامی دیگر ذخیره کنید. برنامه کلیه محتویات فایل انتخاب شده را در فایل جدید ذخیره می کند. متذکر می شویم که در این برنامه نباید از توابعی مانند GlobalAlloc برای اختصاص حافظه استفاده کنید.

```
.386
.model flat,stdcall
WinMain proto :DWORD,:DWORD,:DWORD,:DWORD
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
include \masm32\include\comdlg32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\comdlg32.lib
.const
IDM_OPEN equ 1
IDM_SAVE equ 2
IDM_EXIT equ 3
MAXSIZE equ 260
.data
ClassName db "Win32ASMFileMappingClass",0
AppName  db "Win32 ASM File Mapping Example",0
MenuName db "FirstMenu",0
ofn   OPENFILENAME <>
FilterString db "All Files",0,"*.*",0
             db "Text Files",0,"*.txt",0,0
buffer db MAXSIZE dup(0)
hMapFile HANDLE 0


.data?
hInstance HINSTANCE ?
CommandLine LPSTR ?
hFileRead HANDLE ?
hFileWrite HANDLE ?
hMenu HANDLE ?
pMemory DWORD ?
SizeWritten DWORD ?
.code
start:
        invoke GetModuleHandle, NULL
        mov    hInstance,eax
        invoke GetCommandLine
        mov CommandLine,eax
        invoke WinMain, hInstance,NULL,CommandLine, SW_SHOWDEFAULT
        invoke ExitProcess,eax
WinMain proc
hInst:HINSTANCE,hPrevInst:HINSTANCE,CmdLine:LPSTR,CmdShow:DWORD
    LOCAL wc:WNDCLASSEX
    LOCAL msg:MSG
    LOCAL hwnd:HWND
    mov   wc.cbSize,SIZEOF WNDCLASSEX
    mov   wc.style, CS_HREDRAW or CS_VREDRAW
    mov   wc.lpfnWndProc, OFFSET WndProc
    mov   wc.cbClsExtra,NULL
```

```
    mov   wc.cbWndExtra,NULL
    push  hInst
    pop   wc.hInstance
    mov   wc.hbrBackground,COLOR_WINDOW+1
    mov   wc.lpszMenuName,OFFSET MenuName
    mov   wc.lpszClassName,OFFSET ClassName
    invoke LoadIcon,NULL,IDI_APPLICATION
    mov   wc.hIcon,eax
    mov   wc.hIconSm,eax
    invoke LoadCursor,NULL,IDC_ARROW
    mov   wc.hCursor,eax
    invoke RegisterClassEx, addr wc
    invoke CreateWindowEx, WS_EX_CLIENTEDGE,\
                           ADDR ClassName,\
                           ADDR AppName,\
                           WS_OVERLAPPEDWINDOW,\
                           CW_USEDEFAULT,\
                           CW_USEDEFAULT,\
                           300,\
                           200,\
                           NULL,\
                           NULL,\
                           hInst,\
                           NULL
    mov   hwnd,eax
    invoke ShowWindow, hwnd,SW_SHOWNORMAL
    invoke UpdateWindow, hwnd
    .WHILE TRUE
        invoke GetMessage, ADDR msg,NULL,0,0
        .BREAK .IF (!eax)
        invoke TranslateMessage, ADDR msg
        invoke DispatchMessage, ADDR msg
    .ENDW
    mov     eax,msg.wParam
    ret
WinMain endp
WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
    .IF uMsg==WM_CREATE
        invoke GetMenu,hWnd
        mov  hMenu,eax
        mov ofn.lStructSize,SIZEOF ofn
        push hWnd
        pop  ofn.hWndOwner
        push hInstance
        pop  ofn.hInstance
        mov  ofn.lpstrFilter, OFFSET FilterString
        mov  ofn.lpstrFile, OFFSET buffer
        mov  ofn.nMaxFile,MAXSIZE
    .ELSEIF uMsg==WM_DESTROY
        .if hMapFile!=0
            call CloseMapFile
        .endif
```

```
    invoke PostQuitMessage,NULL
.ELSEIF uMsg==WM_COMMAND
    mov eax,wParam
    .if lParam==0
        .if ax==IDM_OPEN
            mov  ofn.Flags,  OFN_FILEMUSTEXIST or \
                             OFN_PATHMUSTEXIST or \
                             OFN_LONGNAMES or\
                             OFN_EXPLORER or \
                             OFN_HIDEREADONLY
            invoke GetOpenFileName, ADDR ofn
            .if eax==TRUE
                invoke CreateFile, ADDR buffer,\
                                   GENERIC_READ ,\
                                   0,\
                                   NULL,\
                                   OPEN_EXISTING,\
                                   FILE_ATTRIBUTE_ARCHIVE,\
                                   NULL
                mov hFileRead,eax
                invoke CreateFileMapping, hFileRead,\
                                          NULL,\
                                          PAGE_READONLY,\
                                          0,\
                                          0,\
                                          NULL
                mov     hMapFile,eax
                mov     eax,OFFSET buffer
                movzx  edx,ofn.nFileOffset
                add      eax,edx
                invoke SetWindowText,hWnd,eax
                invoke EnableMenuItem,hMenu,IDM_OPEN,MF_GRAYED
                invoke EnableMenuItem,hMenu,IDM_SAVE,MF_ENABLED
            .endif
        .elseif ax==IDM_SAVE
            mov ofn.Flags, OFN_LONGNAMES or\
                           OFN_EXPLORER or \
                           OFN_HIDEREADONLY
            invoke GetSaveFileName, ADDR ofn
            .if eax==TRUE
                invoke CreateFile,\
                         ADDR buffer,\
                         GENERIC_READ or GENERIC_WRITE ,\
                         FILE_SHARE_READ or FILE_SHARE_WRITE,\
                         NULL,\
                         CREATE_NEW,\
                         FILE_ATTRIBUTE_ARCHIVE,\
                         NULL
                mov hFileWrite,eax
                invoke MapViewOfFile,hMapFile,\
                                     FILE_MAP_READ,\
                                     0,\
```

```
                                        0,\
                                        0

                    mov pMemory,eax
                    invoke GetFileSize,hFileRead,NULL
                    invoke WriteFile,hFileWrite,\
                                    pMemory,\
                                    eax,\
                                    ADDR SizeWritten,\
                                    NULL
                    invoke UnmapViewOfFile,pMemory
                    call   CloseMapFile
                    invoke CloseHandle,hFileWrite
                    invoke SetWindowText,hWnd,ADDR AppName
                    invoke EnableMenuItem,hMenu,IDM_OPEN,MF_ENABLED
                    invoke EnableMenuItem,hMenu,IDM_SAVE,MF_GRAYED
                .endif
            .else
                invoke DestroyWindow, hWnd
            .endif
        .endif
    .ELSE
        invoke DefWindowProc,hWnd,uMsg,wParam,lParam
        ret
    .ENDIF
    xor    eax,eax
    ret
WndProc endp
CloseMapFile PROC
        invoke CloseHandle,hMapFile
        mov    hMapFile,0
        invoke CloseHandle,hFileRead
        ret
CloseMapFile endp
end start
```

حال به بررسی کدهای برنامه می پردازیم.

```
invoke CreateWindowEx, WS_EX_CLIENTEDGE,\
                       ADDR ClassName,\
                       ADDR AppName,\
                       WS_OVERLAPPEDWINDOW,\
                       CW_USEDEFAULT,\
                       CW_USEDEFAULT,\
                       300,\
                       200,\
                       NULL,\
                       NULL,\
                       hInst,\
                       NULL
```

بعد از اینکه کاربر فایل مورد نظر را انتخاب کرد ، تابع CreateFile را برای بازکردن آن فراخوانی می کنیم. فایل را بصورت فقط خواندنی باز کرده و dwShareMode را صفر قرار می دهیم زیرا نمی خواهیم Process های دیگر تغییری در فایل ما ایجاد کنند.

```
invoke CreateFileMapping,hFileRead,\
                         NULL,\
                         PAGE_READONLY,\
                         0,\
                         0,\
                         NULL
```

در این مرحله از تابع CreateFileMapping برای ایجاد نگاشت فایل مورد نظر در حافظه استفاده می کنیم. در زیر پیش تعریف این تابع را مشاهده می کنید.

```
CreateFileMapping proto hFile                   :DWORD,\
                        lpFileMappingAttributes :DWORD,\
                        flProtect               :DWORD,\
                        dwMaximumSizeHigh       :DWORD,\
                        dwMaximumSizeLow        :DWORD,\
                        lpName                  :DWORD
```

از این تابع می توانید برای نگاشت قسمت خاصی از فایل نیز استفاده کنید. اندازه حافظه مورد نیاز را می توانید توسط دو پارامتر **dwMaximumSizeLow** و **dwMaximumSizeHigh** مشخص

کنید. اگر اندازه مشخص شده بزرگتر از اندازه واقعی فایل باشد فایل اصلی به اندازه جدید در می آید. اگر می خواهید اندازه فایل نگاشت شده با فایل اصلی برابر باشد هر دو پارامتر بالا را صفر قرار دهید. می توانید برای ایجاد نگاشت فایل با خواص امنیتی پیش فرض مقدار پارامتر **lpFileMappingAttributes** را Null قرار دهید.

flProtect نحوه محافظت از فایل نگاشت شده را معین می‌کند. در این مثال از مدل PAGE_READONLY استفاده می کنیم. توجه داشته باشید که این مشخصات نباید با مشخصات تابع CreateFile تناقض داشته باشد در غیر این صورت تابع با شکست مواجه خواهد شد.

lpName به نام مجازی فایل نگاشت شده که توسط خودمان تعیین می شود اشاره می کند. اگر می‌خواهید این فایل را با Process های دیگر به اشتراک بگذارید باید این نام را معین کنید. در این مثال فقط Process برنامه خودمان از این فایل استفاده می کند. پس این پارامتر را نادیده می گیریم.

```
mov     eax,OFFSET buffer
movzx  edx,ofn.nFileOffset
add      eax,edx
invoke SetWindowText,hWnd,eax
```

در صورت موفقیت آمیز بودن فراخوانی تابع CreateFileMapping ، عنوان پنجره را به نام فایل تغییر می دهیم. این نام به همراه مسیر کامل فایل در بافری ذخیره شده اند. ولی ما می خواهیم فقط نام فایل در عنوان نمایش داده شود پس مقدار متغیر nFileOffset را که از اعضا رکورد OPENFILENAME است ، به آدرس بافر اضافه می کنیم.

```
invoke EnableMenuItem,hMenu,IDM_OPEN,MF_GRAYED
invoke EnableMenuItem,hMenu,IDM_SAVE,MF_ENABLED
```

برای اینکه کاربر نتواند چند فایل را همزمان با هم باز کند گزینه Open را غیر فعال کرده و گزینه Save را فعال می کنیم. از تابع EnableMenuItem برای تغییر خواص آیتم های منو استفاده می‌شود. پس از این کار صبر می کنیم تا کاربر از منوی فایل گزینه Save as را انتخاب کند یا از برنامه خارج شود. در صورت خروج کاربر از برنامه باید فایل نگاشت شده در حافظه را به همراه فایل اصلی ببندیم.

```
.ELSEIF uMsg==WM_DESTROY
        .if hMapFile!=0
            call CloseMapFile
        .endif
        invoke PostQuitMessage,NULL
```

وقتی که زیر برنامه پنجره پیغام WM_DESTROY را دریافت کرد ، ابتدا مقدار متغیر hMapFile را چک می کنیم. در صورتی که مقدار آن غیر صفر باشد ، تابع CloseMapFile را به صورت زیر فراخوانی می کنیم.

```
CloseMapFile PROC
        invoke CloseHandle,hMapFile
        mov    hMapFile,0
        invoke CloseHandle,hFileRead
        ret
CloseMapFile endp
```

این تابع فایل نگاشت شده و فایل اصلی را بسته و منابع سیستم را آزاد می سازد. با انتخاب آیتم Save as ، برنامه یک دیالوگ Save را نمایش می دهد. پس از انتخاب فایل توسط کاربر ، آنرا توسط تابع CreateFile ایجاد می کنیم.

```
invoke MapViewOfFile,hMapFile,\
                     FILE_MAP_READ,\
                     0,\
                     0,\
                     0
mov pMemory,eax
```

پس از ایجاد فایل خروجی ، تابع MapViewOfFile را برای قرار دادن قسمت مورد نظر از فایل نگاشت شده در حافظه فراخوانی می کنیم. پیش تعریف این تابع بصورت زیر است.

```
MapViewOfFile        proto        hFileMappingObject:DWORD,\
                    dwDesiredAccess:DWORD,\
                    dwFileOffsetHigh:DWORD,\
                    dwFileOffsetLow:DWORD,\
                                    dwNumberOfBytesToMap:DWORD
```

dwDesiredAccess نوع دسترسی به فایل را معین می کند. در این مثال فقط می خواهیم اطلاعات را از فایل بخوانیم ، در نتیجه از ثابت FILE_MAP_READ استفاده می کنیم.

dwFileOffsetLow و **dwFileOffsetHigh** افست شروع و پایان تکه ای از فایل که قصد نگاشت آنرا به حافظه داریم ، مشخص می کند. در این مثال می خواهیم کل فایل را بخوانیم پس شروع نگاشت را از افست صفر در نظر می گیریم.

dwNumberOfByteToMap تعداد بایتهایی را که باید به حافظه منتقل شوند مشخص می کند. اگر می خواهید کل فایل را در حافظه نگاشت کنید ، به این پارامتر مقدار صفر بدهید.

پس از فراخوانی تابع MapViewOfFile , قسمت مورد نظر فایل در حافظه بارگذاری شده است و شما اشاره گری به این قسمت از حافظه را دریافت خواهید کرد که حاوی اطلاعات مورد نظر از فایل است.

```
invoke GetFileSize,hFileRead,NULL
```

با فراخوانی تابع GetFileSize اندازه فایل در eax قرار می گیرد. اگر اندازه فایل بیش از ٤ گیگابایت باشد قسمت High DWORD از آن در پارامتر دوم تابع که FileSizeHighWord نام دارد ذخیره می شود. ما انتظار کار با چنین فایلهایی را نداریم پس آن را نادیده می گیریم.

```
invoke WriteFile, hFileWrite,\
                  pMemory,\
                  eax,\
                  ADDR SizeWritten,\
                  NULL
```

اطلاعات را در فایل خروجی می نویسیم.

```
invoke UnmapViewOfFile,pMemory
```

نگاشت فایل را از حافظه خارج می کنیم.

```
call   CloseMapFile
invoke CloseHandle,hFileWrite
```

کلیه فایلها را می بندیم.

```
invoke SetWindowText,hWnd,ADDR AppName
```

بازگردانی عنوان پنجره.

```
invoke EnableMenuItem,hMenu,IDM_OPEN,MF_ENABLED
invoke EnableMenuItem,hMenu,IDM_SAVE,MF_GRAYED
```

حال آیتم open را فعال کرده و آیتم save را غیر فعال می کنیم.

Process

کدها و فایل‌های مربوط به این قسمت در CD ضمیمه موجود می‌باشد

SourceCodes\Asm32\Chapter13

Process عبارتست از یک برنامه اجرایی که شامل فضای حافظه اختصاصی ، کد ، داده ها و منابع خاص خود است.

همانطور که در تعریف بالا مشاهده کردید ، Process شامل فضای حافظه ، روند اجرایی و هر چیزی است که بوسیله این روند اجرایی ایجاد و یا باز می شود. هر Process حداقل شامل یک صف دستورات (Thread) است.

وقتی ویندوز یک Process را شروع می کند ، برای آن یک صف دستورات ایجاد می کند. این صف ، اجرای دستورات را از اولین دستور آغاز می کند . اگر در مراحل بعدی Process نیاز به صف دستورات دیگری داشت می تواند به سادگی آنرا ایجاد کند.

وقتی ویندوز درخواستی را برای ایجاد یک Process دریافت می کند ، یک فضای حافظه مجازی اختصاصی برای آن ایجاد کرده و فایل اجرایی را در این فضا نگاشت می کند. پس از آن صف دستورات اصلی برنامه توسط ویندوز ایجاد می شود. در Win32 می توانید با فراخوانی تابع CreateProcess ، Process جدیدی را ایجاد کنید. در زیر پیش تعریف این تابع را مشاهده می کنید.

```
CreateProcess proto lpApplicationName          :DWORD,\
                    lpCommandLine              :DWORD,\
                    lpProcessAttributes        :DWORD,\
                    lpThreadAttributes         :DWORD,\
                    bInheritHandles            :DWORD,\
                    dwCreationFlags            :DWORD,\
                    lpEnvironment              :DWORD,\
                    lpCurrentDirectory         :DWORD,\
                    lpStartupInfo              :DWORD,\
                    lpProcessInformation       :DWORD
```

lpApplicationName نام فایل اجرایی است که می تواند حاوی مسیر یا بدون آن باشد. اگر مقدار آن را Null قرار دهید باید نام فایل اجرایی را در پارامتر lpCommandLine مشخص کنید.

lpCommandLine آرگومانهای خط فرمان را مشخص می کند. همانطور که ذکر شد در صورتی که مقدار lpApplicationName را Null قرار داده باشید این پارامتر باید شامل فایل اجرایی نیز باشد. به عنوان مثال "notepad.exe my.txt".

lpThreadAttributes و **lpProseccAttributes** مشخصات امنیتی را برای Process و صف دستورات اصلی معین می کند.

bInheritHandles مشخص می کند که آیا Process جدید به تمامی شماره های دسترسی ایجاد شده توسط Process پدر دسترسی داشته باشد یا نه.

dwCreationFlag شامل چندین فلگ است که نحوه عملکرد Process جدید را مشخص می کنند. به عنوان مثال می توانید مشخص کنید وقتی که Process جدید ایجاد شد ، به سرعت متوقف شود برای اینکه شما بتوانید در این فرصت آنرا آزمایش کرده و یا تغییراتی را در آن ایجاد کنید. همچنین به وسیله این فلگ ها می توانید تقدم صف دستورات Process جدید را نسبت به سایر صفهای موجود در سیستم مشخص کنید. معمولاً از حالت NORMAL_PRIORITY_CLASS برای تقدم استفاده می شود.

lpEnvironment اشاره گری به بلوکی از رشته ها است که برای طراحی محیط Process از آنها استفاده می شود. اگر به این پارامتر مقدار Null بدهید ، Process جدید رشته های مورد نیاز خود را از Process پدر به ارث می برد.

lpCurrentDirectory اشاره گر به رشته ای است که مسیر فعلی را برای Process جدید مشخص می کند. در صورت قرار دادن Null , Process جدید آنرا از پدر خود به ارث می برد.

lpStartupInfo اشاره گری به رکورد STARTUPINFO است که نحوه ظهور پنجره اصلی Process جدید را تعیین می کند. این رکورد شامل پارامتر های زیادی است که شکل ظاهری پنجره اصلی Process را مشخص می کنند. می توانید با استفاده از تابع GetStartupInfo اعضا این رکورد را با مقادیر آنها در Process پدر پر کنید.

lpProcessInformaion اشاره گری به رکورد PROCESS_INFORMATION است که اطلاعات شناسایی Process جدید ، توسط تابع در آن ذخیره می شود. در زیر اعضا این رکورد را مشاهده می کنید.

```
PROCESS_INFORMATION STRUCT
    hProcess            HANDLE ?
    hThread             HANDLE ?
    dwProcessId         DWORD  ?
```

```
    dwThreadId          DWORD  ?
PROCESS_INFORMATION ENDS
```

شماره دسترسی Process (Process Handle) و شناسه Process (Process ID) دو آیتم متفاوت هستند. شناسه Process یک شماره یکتا برای Process در سیستم است. شماره دسترسی Process عددی است که توسط ویندوز برای استفاده توسط دیگر توابع API بازگردانده می شود. ا ز این شماره به دلیل یکتا نبودن نمی توانید برای شناسایی یک Process استفاده کنید.

پس از فراخوانی تابع CreateProcess می توانید به وسیله تابع GetExitCodeProcess چک کنید که آیا Process فعال است یا نه. زیرا در هنگام بارگذاری ممکن است مشکلات زیادی برای فایل اجرایی ایجاد شود که مانع از نگاشت صحیح آن در حافظه می گردد.

در زیر ساختار تابع GetExitCodeProcess را مشاهده می کنید.

```
GetExitCodeProcess proto hProcess:DWORD, lpExitCode:DWORD
```

در صورت موفقیت ، تابع وضعیت خاتمه Process را توسط پارامتر lpExitCode مشخص خواهد کرد. اگر این مقدار برابر STILL_ACTIVE باشد یعنی Process در حال اجرا است. می توانید توسط تابع TerminateProcess , Process مورد نظر را به سرعت متوقف کنید.

در زیر ساختار این تابع را مشاهده می کنید.

```
TerminateProcess proto hProcess:DWORD, uExitCode:DWORD
```

استفاده از این تابع روش خوبی برای متوقف کردن برنامه نیست زیرا خاتمه برنامه را به dll هایی که به برنامه متصل هستند اطلاع نمی دهد و آنها همچنان در حافظه باقی می مانند.

حال به سراغ مثال این بخش می رویم.

زمانیکه کاربر آیتم Create Process را انتخاب می کند ، این برنامه Process جدیدی را ایجاد می‌کند که برنامه msgbox.exe است. حال اگر کاربر بخواهد Process ایجاد شده را متوقف کند می‌تواند آیتم Terminate Process را انتخاب کند.

```
.386
.model flat,stdcall
option casemap:none
WinMain proto :DWORD,:DWORD,:DWORD,:DWORD
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
.const
IDM_CREATE_PROCESS equ 1
IDM_TERMINATE equ 2
IDM_EXIT equ 3
.data
ClassName db "Win32ASMProcessClass",0
AppName  db "Win32 ASM Process Example",0
MenuName db "FirstMenu",0
processInfo PROCESS_INFORMATION <>
programname db "msgbox.exe",0
.data?
hInstance HINSTANCE ?
CommandLine LPSTR ?
hMenu HANDLE ?
ExitCode DWORD ?
.code
start:
        invoke GetModuleHandle, NULL
        mov    hInstance,eax
        invoke GetCommandLine
        mov CommandLine,eax
        invoke WinMain, hInstance,NULL,CommandLine, SW_SHOWDEFAULT
        invoke ExitProcess,eax
WinMain proc
hInst:HINSTANCE,hPrevInst:HINSTANCE,CmdLine:LPSTR,CmdShow:DWORD
    LOCAL wc:WNDCLASSEX
    LOCAL msg:MSG
    LOCAL hwnd:HWND
    mov   wc.cbSize,SIZEOF WNDCLASSEX
    mov   wc.style, CS_HREDRAW or CS_VREDRAW
    mov   wc.lpfnWndProc, OFFSET WndProc
    mov   wc.cbClsExtra,NULL
    mov   wc.cbWndExtra,NULL
    push  hInst
    pop   wc.hInstance
    mov   wc.hbrBackground,COLOR_WINDOW+1
    mov   wc.lpszMenuName,OFFSET MenuName
    mov   wc.lpszClassName,OFFSET ClassName
    invoke LoadIcon,NULL,IDI_APPLICATION
    mov   wc.hIcon,eax
    mov   wc.hIconSm,eax
    invoke LoadCursor,NULL,IDC_ARROW
    mov   wc.hCursor,eax
```

```
    invoke RegisterClassEx, addr wc
    invoke CreateWindowEx, WS_EX_CLIENTEDGE,\
                           ADDR ClassName,\
                           ADDR AppName,\
                           WS_OVERLAPPEDWINDOW,\
                           CW_USEDEFAULT,\
                           CW_USEDEFAULT,\
                           300,\
                           200,\
                           NULL,\
                           NULL,\
                           hInst,\
                           NULL
    mov   hwnd,eax
    invoke ShowWindow, hwnd,SW_SHOWNORMAL
    invoke UpdateWindow, hwnd
    invoke GetMenu,hwnd
    mov  hMenu,eax
    .WHILE TRUE
                invoke GetMessage, ADDR msg,NULL,0,0
                .BREAK .IF (!eax)
                invoke TranslateMessage, ADDR msg
                invoke DispatchMessage, ADDR msg
    .ENDW
    mov     eax,msg.wParam
    ret
WinMain endp
WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
    LOCAL startInfo:STARTUPINFO
    .IF uMsg==WM_DESTROY
        invoke PostQuitMessage,NULL
    .ELSEIF uMsg==WM_INITMENUPOPUP
        invoke GetExitCodeProcess,processInfo.hProcess,ADDR ExitCode

        .if eax==TRUE
            .if ExitCode==STILL_ACTIVE
                invoke EnableMenuItem,hMenu,\
                                      IDM_CREATE_PROCESS,\
                                      MF_GRAYED
                invoke EnableMenuItem,hMenu,\
                                      IDM_TERMINATE,\
                                      MF_ENABLED
            .else
                invoke EnableMenuItem,hMenu,\
                                      IDM_CREATE_PROCESS,\
                                      MF_ENABLED

                invoke EnableMenuItem,hMenu,IDM_TERMINATE,MF_GRAYED
            .endif
        .else
            invoke EnableMenuItem,hMenu,\
                                  IDM_CREATE_PROCESS,\
```

```
                                    MF_ENABLED

            invoke EnableMenuItem,hMenu,IDM_TERMINATE,MF_GRAYED
        .endif
    .ELSEIF uMsg==WM_COMMAND
        mov eax,wParam
        .if lParam==0
            .if ax==IDM_CREATE_PROCESS
                .if processInfo.hProcess!=0
                    invoke CloseHandle,processInfo.hProcess
                    mov processInfo.hProcess,0
                .endif
                invoke GetStartupInfo,ADDR startInfo
                invoke CreateProcess,ADDR programname,\
                                     NULL,\
                                     NULL,\
                                     NULL,\
                                     FALSE,\
                                     NORMAL_PRIORITY_CLASS,\
                                     NULL,\
                                     NULL,\
                                     ADDR startInfo,\
                                     ADDR processInfo
                invoke CloseHandle,processInfo.hThread
            .elseif ax==IDM_TERMINATE
                invoke GetExitCodeProcess,processInfo.hProcess,\
                                          ADDR ExitCode

                .if ExitCode==STILL_ACTIVE
                    invoke TerminateProcess,processInfo.hProcess,0
                .endif
                invoke CloseHandle,processInfo.hProcess
                mov processInfo.hProcess,0
            .else
                invoke DestroyWindow,hWnd
            .endif
        .endif
    .ELSE
        invoke DefWindowProc,hWnd,uMsg,wParam,lParam
        ret
    .ENDIF
    xor    eax,eax
    ret
WndProc endp
end start
```

حال به بررسی کد های برنامه می پردازیم.

برنامه ابتدا پنجره اصلی را ایجاد کرده و شماره دسترسی منو را برای استفاده های بعدی ذخیره می کند. هنگامی که کاربر منوی Process را انتخاب کند برنامه پیغام WM_INITMENUPOPUP را دریافت می کند . از این فرصت برای تغییر خصوصیات آیتم های زیر منو قبل از نمایش آنها استفاده می کنیم.

```
.ELSEIF uMsg==WM_INITMENUPOPUP
    invoke GetExitCodeProcess,processInfo.hProcess,ADDR ExitCode

    .if eax==TRUE
        .if ExitCode==STILL_ACTIVE
            invoke EnableMenuItem,hMenu,\
                                  IDM_CREATE_PROCESS,\
                                  MF_GRAYED
            invoke EnableMenuItem,hMenu,IDM_TERMINATE,MF_ENABLED

        .else
            invoke EnableMenuItem,hMenu,\
                                  IDM_CREATE_PROCESS,\
                                  MF_ENABLED

            invoke EnableMenuItem,hMenu,IDM_TERMINATE,MF_GRAYED
        .endif
    .else
        invoke EnableMenuItem,hMenu,\
                              IDM_CREATE_PROCESS,\
                              MF_ENABLED

        invoke EnableMenuItem,hMenu,IDM_TERMINATE,MF_GRAYED
    .endif
```

در این مثال اگر Process مورد نظر هنوز شروع نشده باشد باید آیتم Start Process را فعال و آیتم Terminate Process را غیر فعال کنیم. و در صورتی که Process قبلا شروع شده باشد باید عکس این عمل را انجام دهیم. ابتدا با استفاده از تابع GerExitCodeProcess چک می کنیم که آیا Process در حال اجرا است یا نه. در صورتی که تابع مقدار Flase را برگرداند به این معنی است که Process قبلا شروع نشده است. پس باید گزینه Terminate Process را غیر فعال کنیم. ولی اگر تابع مقدار True را برگرداند نشان دهنده این است که Process مورد نظر شروع شده است. در

مرحله بعدی چک می کنیم که آیا Process هنوز در حال اجرا هست یا نه. به این منظور مقدار ExitCode را با مقدار STILL_ACTIVE مقایسه می کنیم. در صورت برابری این دو مقدار ، Process هنوز در حال اجرا است پس گزینه Start Process را غیر فعال می کنیم.

```
.if ax==IDM_CREATE_PROCESS
    .if processInfo.hProcess!=0
        invoke CloseHandle,processInfo.hProcess
        mov processInfo.hProcess,0
    .endif
    invoke GetStartupInfo,ADDR startInfo
                invoke CreateProcess,ADDR programname,\
                                     NULL,\
                                      NULL,\
                                      NULL,\
                                      FALSE,\
                                      NORMAL_PRIORITY_CL
                                      ASS,\
                                      NULL,\
                                      NULL,\
                                      ADDR startInfo,\
                                      ADDR processInfo
    invoke CloseHandle,processInfo.hThread
```

در صورت انتخاب آیتم Start Process چک می کنیم که آیا hProcess که از اعضا رکورد PROCESS_INFORMATION است قبلا بسته شده یا نه. مقدار اولیه این متغیر همیشه صفر خواهد بود زیرا ما رکورد PROCESS_INFORMATION را در قسمت data. تعریف کرده ایم. ولی اگر مقدار hProcess مخالف صفر باشد یعنی Process فرزند خاتمه یافته ولی شماره دسترسی Process هنوز بسته نشده است. پس در این مرحله این کار را انجام می دهیم.

در مرحله بعد تابع GetStartupInfo را برای پر کردن اعضا رکورد Startinfo فراخوانی می کنیم. از این رکورد به عنوان پارامتر ورودی برای تابع CreateProcess استفاده می کنیم. حال تابع CreateProcess را فراخوانی می کنیم. متذکر می شویم که در این مثال ما مقدار برگشتی از این تابع را چک نکرده ایم چون مثال را پیچیده تر می کرد ولی در واقع باید مقدار برگشتی آنرا چک کنیم. به سرعت پس از فراخوانی این تابع ، شماره دسترسی به صف اصلی دستورات را که یکی از اعضا رکورد Processinfo است می بندیم. بستن شماره دسترسی به معنای توقف صف دستورات نیست. فقط به دلیل اینکه نیازی به استفاده از این شماره دسترسی برای ارجاع به صف دستورات Process فرزند نداریم این کار را انجام می دهیم و در حقیقت با انجام این کار از اتلاف منابع سیستم جلوگیری می کنیم.

```
.elseif ax==IDM_TERMINATE
    invoke GetExitCodeProcess,processInfo.hProcess,\
                              ADDR ExitCode

    .if ExitCode==STILL_ACTIVE
        invoke TerminateProcess,processInfo.hProcess,0
    .endif
    invoke CloseHandle,processInfo.hProcess
    mov processInfo.hProcess,0
```

در صورت انتخاب آیتم Terminate Process توسط کاربر ، ابتدا چک می کنیم که Process جدید فعال است یا نه. در صورت فعال بودن ، با استفاده از تابع TerminateProcess , Process را از بین می بریم. در مرحله نهایی نیز شماره دسترسی به Process فرزند را می بندیم.

Multithreading

کدها و فایل‌های مربوط به این قسمت در CD ضمیمه موجود می‌باشد

SourceCodes\Asm32\Chapter14

در این بخش روش ایجاد و کنترل برنامه های Multithread را مورد بررسی قرار می دهیم. همچنین در مورد متدهای ارتباطی بین صف های دستورات نیز نکاتی خواهید آموخت.

در بخش قبل دریافتید که هر برنامه حداقل حاوی یک صف دستورات اصلی می باشد. شما می توانید در صورت لزوم در برنامه خود صف های اضافی نیز ایجاد کنید. در حقیقت صف دستورات یک تابع است که همزمان با برنامه اصلی اجرا می شود و می توانید بر حسب نیاز خود ، این توابع را ایجاد و از آنها استفاده کنید. این قابلیت ها تنها در Win32 قابل استفاده هستند و در Win16 نمونه ای برای آن وجود ندارد.

تمام صف هایی که در یک پروسه اجرا می شوند می توننند به هر یک از منابع آن پروسه از قبیل شماره های دسترسی یا متغیر های عمومی دسترسی داشته باشند. هر یک از این صف ها دارای Stack مخصوص به خود هستند و متغیر های عمومی آنها بصورت خصوصی می باشند. هر صف ثباتهای مخصوص به خود را دارد و دلیل آن اینست که وقتی ویندوز به صف دیگری می رود صف قبلی بتواند آخرین وضعیت خود را در خود ذخیره داشته باشد تا در موقع نیاز بتواند حالت گذشته خود را بازیابی کند.

صف ها به دو نوع تقسیم می شوند.

۱- **صف های رابط کاربر** : این نوع صف ها پنجره خاص خود را ایجاد کرده و پیغام های ویندوز را دریافت می کنند. و از طریق این پنجره به کاربر پاسخ می دهند.

۲- **صف های کاری** : این نوع صف ها پنجره ای ایجاد نمی کنند و نمی توانند پیغام های ویندوز را دریافت کنند و اصولا برای انجام کار در پشت زمینه طراحی شده اند.

برای استفاده بهتر از توانایی های Multithreading در ویندوز روش زیر را به شما توصیه می‌کنیم.

صف اصلی را مسئول کارهای رابط کاربر قرار دهید و دیگر صف ها در پشت زمینه کارهای سخت تر را انجام دهند. در این روش صف اصلی مانند رئیس بوده و دیگر صف ها به عنوان کارمند محسوب می شوند.

این رئیس همزمان با کنترل رابط کاربر ، کارها را به کارمندان خود واگذار می کند. کارمندان نیز کارهای مورد نظر را انجام داده و به رئیس گزارش می دهند. اگر قرار بود که رئیس تمام کارها را خودش انجام دهد دیگر وقتی برای توجه به امور مهم تر باقی نمی ماند. برنامه ها نیز به همین صورت عمل می کنند و با ایجاد صف های اضافی کارهای طولانی را به این صف ها محول می کنند و صف اصلی وظیفه پاسخگویی به دستورات کاربر را بر عهده می گیرد.

برای ایجاد صف دستورات از تابع CreateThread استفاده می کنیم که پیش تعریف آن را در زیر مشاهده می کنید.

```
CreateThread proto lpThreadAttributes   :DWORD,\
                   dwStackSize          :DWORD,\
                   lpStartAddress       :DWORD,\
                   lpParameter          :DWORD,\
                   dwCreationFlags      :DWORD,\
                   lpThreadId           :DWORD
```

lpThreadAttributes متد امنیتی مورد نیاز برای صف جدید را مشخص می کند. در صورتی که می خواهید صف جدید از متدهای پیش فرض استفاده کند به این پارامتر مقدار Null بدهید.

dwStackSize اندازه پشته مورد نظر را مشخص می کند. در صورت استفاده از مقدار Null اندازه پشته صف جدید به اندازه صف اصلی خواهد بود.

lpStartAddress آدرس تابع صف دستورات را مشخص می کند. این تابع عملیات مورد نظر صف جدید را انجام می دهد. تابع مذکور یک پارامتر ورودی 32 بیتی را گرفته و یک مقدار 32 بیتی نیز باز می گرداند.

lpParameter پارامتری است که قصد دارید آن را به صف ارسال کنید. dwCreationFlag وضعیت اجرایی صف را مشخص می کند. مقدار صفر به این معنا است که دستورات صف ، بعد از ایجاد شدن به سرعت اجرا شوند. مقادیر مخالف صفر ، فلگ CREATE_SUSPENDED را مشخص می کنند.

lpThreadID شماره دسترسی به صف جدید را مشخص می کند. اگر فراخوانی تابع موفقیت آمیز باشد مقدار این شماره در متغیر lpThreadID قرار می گیرد. در غیر این صورت مقدار این متغیر Null خواهد بود.

در صورت عدم استفاده از فلگ CREATE_SUSPENDED ، پس از فراخوانی تابع CreateThread ، تابع صف شروع به اجرا می کند.

در صورت استفاده از فلگ CREATE_SUSPENDED ، صف دستورات متوقف می شود . برای اجرای آن می توانید از تابع ResumeThread استفاده کنید.

وقتی تابع صف با دستورات Ret به کار خود خاتمه دهد ، ویندوز به طور ضمنی تابع ExitThread را فراخوانی می کند. البته خودتان نیز می توانید در تابع صف از آن استفاده کنید.

در صورت نیاز می توانید با استفاده از تابع TerminateThread ، به فعالیت صف مورد نظر خاتمه دهید. ولی از این تابع فقط در مواقع ضروری استفاده کنید زیرا با فراخوانی این تابع صف مورد نظر به سرعت خاتمه می یابد و هیچ فرصتی برای پاک سازی و انجام عملیات پایانی به آن داده نمی شود.

حال متدهای ارتباطی بیت صف ها را بررسی می کنیم.

سه نوع مختلف از این متد ها وجود دارند که عبارتند از:

- استفاده از متغیر های عمومی
- پیغام های ویندوز
- Event

صف های دستورات کلیه منابع پروژه را که شامل متغیرهای عمومی نیز می باشند به اشتراک می گذارند. پس می توانید از این متغیر های عمومی برای ایجاد ارتباط بین صف ها استفاده کنید. در استفاده از این روش باید با احتیاط عمل کنید و همزمانی عملکرد صف ها را در نظر داشته باشید. به عنوان مثال وقتی دو صف از یک رکورد استفاده می کنند چه روی می دهد اگر ویندوز کنترل را از روی صفی که در حال مقدار دهی رکورد است به صف دیگر منتقل کند؟ مسلما صف جدید با یک سری اطلاعات ناقص در رکورد مواجه خواهد شد. سعی کنید که مرتکب این گونه اشتباهات نشوید زیرا دیباگ کردن و رفع اشکالات برنامه های MultiThread بسیار سخت است. برطرف کردن این گونه مشکلات که ممکن است به صورت تصادفی اتفاق بیفتد می تواند ساعتها وقت شما را تلف کند.

همچنین می توانید از پیغام های ویندوز برای ارتباط بین صف ها استفاده کنید.

اگر هر دو صف از نوع رابط کاربر باشند هیچ مشکلی به وجود نمی آید و می توان از این روش به عنوان یک رابط دو طرفه استفاده کرد. تنها کاری که باید انجام دهید تعریف و انتخاب چند پیغام است که برای هر دو طرف مفهوم باشند. پیغام های خود را می توانید با استفاده از پیغام WM_USER به عنوان پایه ایجاد کنید. به عنوان مثال:

```
WM_MYCUSTOMMSG equ WM_USER+100h
```

ویندوز هیچ گاه از مقدار WM_USER و بالاتر از آن برای پیغام های خود استفاده نمی کند و شما می توانید با استفاده از این مقادیر پیغام های خاصی را برای خود تعریف کنید.

اگر یکی از صف ها از نوع رابط کاربر و دیگری از نوع کاری باشد از این روش نمی توانید به عنوان یک رابطه دو طرفه استفاده کنید چون صف کاری ، پنجره مخصوص به خود ندارد و نمی تواند پیغام های شما را دریافت کند.

در زیر مقدار های ارتباطی بین صف ها را مشاهده می کنید.

```
User interface Thread --> global variable(s) --> Worker thread
Worker Thread  ---> custom window message(s) --> User Interface Thread
```

در واقع در مثال این بخش نیز از این روش استفاده کرده ایم.

آخرین متد ارتباطی استفاده از شی Event است. می توانید به این شی همانند یک فلگ نگاه کنید. اگر این فلگ در حالت غیر فعال باشد ویندوز صف دستورات را به راه می اندازد صف دستورات نیز شروع به انجام کارهای مورد نظر می کند.

حال به سراغ مثال این بخش می رویم.

با انتخاب آیتم Savage Calculation برنامه عمل" add eax , eax" را 600,000,000 بار انجام می دهد. متذکر می شویم که به طور معمول در طی انجام این عملیات نمی توانید هیچ کاری با پنجره اصلی انجام دهید. به عنوان مثال نمی توانید آنرا جابجا کرده و یا منوی آنرا فعال کنید. پس از اتمام محاسبات ، یک MessageBox ظاهر شده و بعد از آن پنجره ، پیغام های شما را قبول می کند.

برای جلوگیری از اینگونه مشکلات می توانید کل محاسبات را به یک صف کاری جدید انتقال دهید و صف اصلی به کار پاسخگویی به دستورات کاربر ادامه دهد.

```
.386
.model flat,stdcall
option casemap:none
WinMain proto :DWORD,:DWORD,:DWORD,:DWORD
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
.const
IDM_CREATE_THREAD equ 1
IDM_EXIT equ 2
WM_FINISH equ WM_USER+100h
.data
ClassName db "Win32ASMThreadClass",0
AppName  db "Win32 ASM MultiThreading Example",0
MenuName db "FirstMenu",0
SuccessString db "The calculation is completed!",0
.data?
hInstance HINSTANCE ?
CommandLine LPSTR ?
hwnd HANDLE ?
ThreadID DWORD ?
.code
start:
    invoke GetModuleHandle, NULL
    mov    hInstance,eax
    invoke GetCommandLine
    mov CommandLine,eax
    invoke WinMain, hInstance,NULL,CommandLine, SW_SHOWDEFAULT
    invoke ExitProcess,eax
WinMain proc
hInst:HINSTANCE,hPrevInst:HINSTANCE,CmdLine:LPSTR,CmdShow:DWORD
    LOCAL wc:WNDCLASSEX
    LOCAL msg:MSG
    mov   wc.cbSize,SIZEOF WNDCLASSEX
    mov   wc.style, CS_HREDRAW or CS_VREDRAW
    mov   wc.lpfnWndProc, OFFSET WndProc
    mov   wc.cbClsExtra,NULL
    mov   wc.cbWndExtra,NULL
    push  hInst
    pop   wc.hInstance
    mov   wc.hbrBackground,COLOR_WINDOW+1
    mov   wc.lpszMenuName,OFFSET MenuName
    mov   wc.lpszClassName,OFFSET ClassName
    invoke LoadIcon,NULL,IDI_APPLICATION
    mov   wc.hIcon,eax
    mov   wc.hIconSm,eax
    invoke LoadCursor,NULL,IDC_ARROW
```

```
    mov   wc.hCursor,eax
    invoke RegisterClassEx, addr wc
    invoke CreateWindowEx, WS_EX_CLIENTEDGE,\
                           ADDR ClassName,\
                           ADDR AppName,\
                           WS_OVERLAPPEDWINDOW,\
                           CW_USEDEFAULT,\
                           CW_USEDEFAULT,\
                           300,\
                           200,\
                           NULL,\
                           NULL,\
                           hInst,\
                           NULL
    mov   hwnd,eax
    invoke ShowWindow, hwnd,SW_SHOWNORMAL
    invoke UpdateWindow, hwnd
    .WHILE TRUE
            invoke GetMessage, ADDR msg,NULL,0,0
            .BREAK .IF (!eax)
            invoke TranslateMessage, ADDR msg
            invoke DispatchMessage, ADDR msg
    .ENDW
    mov     eax,msg.wParam
    ret
WinMain endp
WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
    .IF uMsg==WM_DESTROY
        invoke PostQuitMessage,NULL
    .ELSEIF uMsg==WM_COMMAND
        mov eax,wParam
        .if lParam==0
            .if ax==IDM_CREATE_THREAD
                mov  eax,OFFSET ThreadProc
                invoke CreateThread,NULL,\
                                    NULL,\
                                    eax,\
                                    0,\
                                    ADDR ThreadID

                invoke CloseHandle,eax
            .else
                invoke DestroyWindow,hWnd
            .endif
        .endif
    .ELSEIF uMsg==WM_FINISH
        invoke MessageBox,NULL,\
                          ADDR SuccessString,\
                          ADDR AppName,\
                          MB_OK

    .ELSE
```

```
        invoke DefWindowProc,hWnd,uMsg,wParam,lParam
        ret
    .ENDIF
    xor    eax,eax
    ret
WndProc endp
ThreadProc PROC USES ecx Param:DWORD
        mov  ecx,600000000
Loop1:
        add  eax,eax
        dec  ecx
        jz   Get_out
        jmp  Loop1
Get_out:
        invoke PostMessage,hwnd,WM_FINISH,NULL,NULL
        ret
ThreadProc ENDP
end start
```

حال به تجزیه و تحلیل کدهای برنامه می پردازیم.

با انتخاب آیتم CreateThread توسط کاربر ، برنامه صف جدیدی را به صورت زیر ایجاد می کند.

```
.if ax==IDM_CREATE_THREAD
    mov  eax,OFFSET ThreadProc
    invoke CreateThread,NULL,\
                        NULL,\
                        eax,\
                        NULL,\
                        0,\
                        ADDR ThreadID

    invoke CloseHandle,eax
```

تابع بالا یک صف جدید ایجاد می کند که زیر برنامه ای به نام ThreadProc را همزمان با صف اصلی اجرا می کند. در صورت موفقیت آمیز بودن فراخوانی ، تابع CreateThread به سرعت باز می گردد و زیر برنامه ThreadProc شروع به کار می کند. به دلیل اینکه قصد استفاده از شماره دسترسی صف دستورات را نداریم آنرا می بندیم زیرا باعث اتلاف منبع سیستم می گردد. بستن این شماره دسترسی به معنی بستن صف دستورات نیست و تنها اثر آن اینست که دیگر نمی توانید از این شماره استفاده کنید.

```
ThreadProc PROC USES ecx Param:DWORD
        mov  ecx,600000000
Loop1:
        add  eax,eax
        dec  ecx
        jz   Get_out
        jmp  Loop1
Get_out:
        invoke PostMessage,hwnd,WM_FINISH,NULL,NULL
        ret
ThreadProc ENDP
```

همان طور که می بینید زیر برنامه ThreadProc محاسبات را انجام داده و پس از اتمام آنها پیغام WM_FINISH را به پنجره اصلی می فرستد. پیغام WM_FINISH از پیغام هایی است که خودمان تعریف کرده ایم. در زیر ، تعریف آنرا مشاهده می کنید.

```
WM_FINISH equ WM_USER+100h
```

نیازی به اضافه کردن 100h به WM_USER نیست ولی برای امنیت بیشتر این کار را انجام داده ایم. این پیغام تنها برای برنامه ما شناخته شده است. با دریافت این پیغام توسط پنجره اصلی ، برنامه یک MessageBox را نمایش می دهد و اعلام می کند که محاسبات به پایان رسیده است.

در این مثال ارتباط یک طرفه بود. اگر می خواهید پنجره اصلی هم به صف دستورات کاری پیغام بفرستد می توانید بصورت زیر عمل کنید.

- به آیتم های منو آیتمی با نام KillThread اضافه کنید.
- یک متغیر عمومی تعریف کنید که از آن به عنوان فلگ استفاده خواهیم کرد. در فلگ مقدار True به معنی توقف و False به معنی ادامه روند اجرایی دستورات صف جدید است.
- تغییر در زیر برنامه ThreadProc برای اینکه مقدار این فلگ را در حلقه چک کند.

در صورت انتخاب آیتم KillThread توسط کاربر ، برنامه مقدار فلگ را برابر True قرار می دهد. زیر برنامه ThreadProc با مشاهده این مقدار از حلقه خارج شده و بازگشت می کند و به این ترتیب صف دستورات خاتمه می یابد.

شئ Event

کدها و فایل‌های مربوط به این قسمت در CD ضمیمه موجود می‌باشد

SourceCodes\Asm32\Chapter15

در فصل قبل دیدیم که چگونه صف های دستورات با پیغام های خاص با یکدیگر ارتباط برقرار می کنند. در این فصل دو روش دیگر را برای برقراری ارتباط بین صف ها بررسی می کنیم که عبارتند از: متغیر های عمومی و شئ Event . شئ Event همانند یک سوییچ عمل می کند که فقط دو حالت خاموش و روشن دارد. وقتی که این شئ روشن است یعنی در حالت Signalled است و هنگامی که خاموش است یعنی در حالت Nonsignalled قرار دارد. شما تنها باید یک Event را ایجاد کرده و با چند دستور کوتاه در صف های مربوطه حالت آنرا بررسی کنید. وقتی که یک Event در حالت Nonsignalled باشد صف هایی که منتظر آن هستند تقریبا در حالت غیر فعال به سر می برند و به میزان ناچیزی از منابع CPU استفاده می کنند.

برای ایجاد یک شئ Event از تابع CreateEvent استفاده می کنیم که ساختار آن را در زیر مشاهده می کنید.

```
CreateEvent proto lpEventAttributes     :DWORD,\
                  bManualReset          :DWORD,\
                  bInitialState         :DWORD,\
                  lpName:DWORD
```

lpEventAttribute : وضعیت امنیتی Event را مشخص می کند که با قرار دادن مقدار Null به جای آن از وضعیت پیش فرض استفاده خواهد شد.

bManualReset : اگر می خواهید که ویندوز بطور خودکار پس از فراخوانی تابع WaitForSingalObject شئ Event را به حالت Nonsignalled ببرد ، به این پارامتر مقدار False بدهید در غیر این صورت باید خودتان این عمل را با استفاده از تابع ResetEvent انجام دهید.

bInitialState : اگر می خواهید که Event با حالت Signaled ایجاد شود ، مقدار این پارامتر را True قرار دهید. در غیر این صورت Event با حالت اولیه Nonsignalled ایجاد خواهد شد.

lpName : اشاره گر به یک رشته که نام این رویداد را مشخص می کند. از این نام به عنوان پارامتر ورودی برای تابع OpenEvent استفاده می شود. در صورت موفقیت آمیز بودن فراخوانی تابع

CreateEvent مقدار بازگشتی آن شماره دسترسی به Event ایجاد شده را مشخص می کند. در غیر این صورت مقدار برگشتی Null خواهد بود.

حالت شی Event ایجاد شده را می توانید توسط دو تابع API تغییر دهید. تابع SetEvent شئ Event مورد نظر را در حالت Signaled قرار می دهد و تابع ResetEvent آنرا در حالت Nonsignalled قرار می دهد. برای بررسی حالت Event از تابع WaitForSingleObject در صف مورد نظر استفاده می کنیم. در زیر ساختار این تابع را مشاهده می کنید.

```
WaitForSingleObject proto hObject:DWORD, dwTimeout:DWORD
```

hObject شماره دسترسی به یک شئ همزمان ساز را مشخص می کند. شئ Event از انواع اشیا همزمان ساز به شمار می آید.

dwTimeOut حداکثر مدت زمانی است که تابع برای ورود شی به حالت Signaled منتظر می‌ماند. با پایان یافتن زمان مشخص شده تابع بازگشت می کند. اگر می خواهید برای همیشه این حالت انتظار ادامه یابد از مقدار INFINITE استفاده کنید.

حال به سراغ مثال این بخش می رویم.

در این مثال ابتدا پنجره برنامه نمایش داده می شود. در صورت انتخاب آیتم Run Thread توسط کاربر ، صف جدید انجام محاسبات را شروع می کند و در پایان هم یک MessageBox ظاهر شده و اتمام محاسبات را اطلاع می دهد. در طول انجام این محاسبات کاربر می تواند با انتخاب آیتم Stop Thread صف ایجاد شده را متوقف کند.

```
.386
.model flat,stdcall
option casemap:none
WinMain proto :DWORD,:DWORD,:DWORD,:DWORD
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
.const
IDM_START_THREAD equ 1
IDM_STOP_THREAD equ 2
IDM_EXIT equ 3
WM_FINISH equ WM_USER+100h
.data
ClassName db "Win32ASMEventClass",0
AppName  db "Win32 ASM Event Example",0
```

```
MenuName db "FirstMenu",0
SuccessString db "The calculation is completed!",0
StopString db "The thread is stopped",0
EventStop BOOL FALSE
.data?
hInstance HINSTANCE ?
CommandLine LPSTR ?
hwnd HANDLE ?
hMenu HANDLE ?
ThreadID DWORD ?
ExitCode DWORD ?
hEventStart HANDLE ?
.code
start:
    invoke GetModuleHandle, NULL
    mov    hInstance,eax
    invoke GetCommandLine
    mov CommandLine,eax
    invoke WinMain, hInstance,NULL,CommandLine, SW_SHOWDEFAULT
    invoke ExitProcess,eax
WinMain proc
hInst:HINSTANCE,hPrevInst:HINSTANCE,CmdLine:LPSTR,CmdShow:DWORD
    LOCAL wc:WNDCLASSEX
    LOCAL msg:MSG
    mov   wc.cbSize,SIZEOF WNDCLASSEX
    mov   wc.style, CS_HREDRAW or CS_VREDRAW
    mov   wc.lpfnWndProc, OFFSET WndProc
    mov   wc.cbClsExtra,NULL
    mov   wc.cbWndExtra,NULL
    push  hInst
    pop   wc.hInstance
    mov   wc.hbrBackground,COLOR_WINDOW+1
    mov   wc.lpszMenuName,OFFSET MenuName
    mov   wc.lpszClassName,OFFSET ClassName
    invoke LoadIcon,NULL,IDI_APPLICATION
    mov   wc.hIcon,eax
    mov   wc.hIconSm,eax
    invoke LoadCursor,NULL,IDC_ARROW
    mov   wc.hCursor,eax
    invoke RegisterClassEx, addr wc
    invoke CreateWindowEx, WS_EX_CLIENTEDGE,\
                           ADDR ClassName,\
                           ADDR  AppName,\
                           WS_OVERLAPPEDWINDOW,\
                           CW_USEDEFAULT,\
                           CW_USEDEFAULT,\
                           300,\
                           200,\
                           NULL,\
                           NULL,\
                           hInst,\
                           NULL
```

```
    mov   hwnd,eax
    invoke ShowWindow, hwnd,SW_SHOWNORMAL
    invoke UpdateWindow, hwnd
    invoke GetMenu,hwnd
    mov  hMenu,eax
    .WHILE TRUE
            invoke GetMessage, ADDR msg,NULL,0,0
            .BREAK .IF (!eax)
            invoke TranslateMessage, ADDR msg
            invoke DispatchMessage, ADDR msg
    .ENDW
    mov     eax,msg.wParam
    ret
WinMain endp
WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
    .IF uMsg==WM_CREATE
        invoke CreateEvent,NULL,FALSE,FALSE,NULL
        mov  hEventStart,eax
        mov  eax,OFFSET ThreadProc
        invoke CreateThread,NULL,\
                            NULL,\
                            eax,\
                            NULL,\
                            0,\
                            ADDR ThreadID
        invoke CloseHandle,eax
    .ELSEIF uMsg==WM_DESTROY
        invoke PostQuitMessage,NULL
    .ELSEIF uMsg==WM_COMMAND
        mov eax,wParam
        .if lParam==0
            .if ax==IDM_START_THREAD
                invoke SetEvent,hEventStart
                invoke EnableMenuItem,hMenu,\
                                      IDM_START_THREAD,\
                                      MF_GRAYED

                invoke EnableMenuItem,hMenu,\
                                      IDM_STOP_THREAD,\
                                      MF_ENABLED

            .elseif ax==IDM_STOP_THREAD
                mov  EventStop,TRUE
                invoke EnableMenuItem,hMenu,\
                                      IDM_START_THREAD,\
                                      MF_ENABLED

                invoke EnableMenuItem,hMenu,\
                                      IDM_STOP_THREAD,\
                                      MF_GRAYED

            .else
```

```
                invoke DestroyWindow,hWnd
            .endif
        .endif
    .ELSEIF uMsg==WM_FINISH
        invoke MessageBox,NULL,ADDR SuccessString,ADDR AppName,MB_OK

    .ELSE
        invoke DefWindowProc,hWnd,uMsg,wParam,lParam
        ret
.ENDIF
    xor    eax,eax
    ret
WndProc endp
ThreadProc PROC USES ecx Param:DWORD
        invoke WaitForSingleObject,hEventStart,INFINITE
        mov  ecx,600000000
        .WHILE ecx!=0
                .if EventStop!=TRUE
                        add  eax,eax
                        dec  ecx
                .else
                        invoke MessageBox,hwnd,\
                                        ADDR StopString,\
                                        ADDR AppName,\
                                        MB_OK
                        mov  EventStop,FALSE
                        jmp ThreadProc
                .endif
        .ENDW
        invoke PostMessage,hwnd,WM_FINISH,NULL,NULL
        invoke EnableMenuItem,hMenu,IDM_START_THREAD,MF_ENABLED
        invoke EnableMenuItem,hMenu,IDM_STOP_THREAD,MF_GRAYED
        jmp   ThreadProc
        ret
ThreadProc ENDP
end start
```

حال به بررسی کدهای این برنامه می پردازیم.

```
    .IF uMsg == WM_CREATE
        invoke CreateEvent,NULL,FALSE,FALSE,NULL
        mov  hEventStart,eax
        mov  eax,OFFSET ThreadProc
        invoke CreateThread,NULL,\
                            NULL,\
                            eax,\
                            NULL,\
                            0,\
                            ADDR ThreadID
        invoke CloseHandle,eax
```

شئ Event و صف دستورات را در طول پردازش پیغام WM_CREATE ایجاد می کنیم. همانطور که در کد پایین مشاهده می کنید صف دستورات به علت اینکه در انتظار حالت Signaled است ، اجرا نمی شود.

```
ThreadProc PROC USES ecx Param:DWORD
        invoke WaitForSingleObject,hEventStart,INFINITE
        mov  ecx,600000000
```

اولین خط از زیر برنامه ، فراخوانی تابع WaitForSingleObject است. این تابع منتظر حالت Signaled از شئ Event می ماند و قبل از آن بازگشت نمی کند. با این کار پس از ایجاد صف جدید آنرا در حالت ساکن نگه می داریم. وقتی کاربر آیتم Run Thread را انتخاب کرد ، به صورت زیر Event را در حالت Signaled قرار می دهیم.

```
.if ax==IDM_START_THREAD
    invoke SetEvent,hEventStart
```

تابع SetEvent شئ Event را به حالت Signaled می برد و تابع WaitForSignalObject را مجبور به بازگشت کرده و صف دستورات کار خود را شروع می کند. با انتخاب آیتم Stop Thread مقدار متغیر عمومی EventStop را True قرار می دهیم.

```
.if EventStop==FALSE
        add  eax,eax
        dec  ecx
.else
        invoke MessageBox, hwnd,\
                           ADDR StopString,\
                           ADDR AppName,\
                           MB_OK
        mov  EventStop,FALSE
        jmp ThreadProc
.endif
```

این عمل صف دستورات را متوقف کرده و به تابع WaitForSignalObject پرش می کند. متذکر می شویم که نباید خودتان Event را به حالت Nonsignalled ببرید زیرا مقدار پارامتر bManualResetاز تابع CreateEvent را False قرار داده ایم.

نحوه ساخت و استفاده از dll ها

کدها و فایلهای مربوط به این قسمت در CD ضمیمه موجود می‌باشد

SourceCodes\Asm32\Chapter16

اگر در زمینه برنامه نویسی تجربه کافی داشته باشید حتما به این نکته توجه داشته اید که برنامه ها معمولاً دارای بعضی روالهای مشترک هستند. اگر برای نوشتن هر برنامه جدید بخواهید این روالها را بازنویسی کنید مجبور به صرف وقت زیادی هستید. در زمانهای گذشته در سیستم عامل Dos برنامه نویسان این روال های عمومی را در کتابخانه هایی قرار می دادند و در صورت نیاز کتابخانه مورد نظر را به فایل Object برنامه پیوند می زدند و در مرحله کامپایل ، پیوند دهنده (Linker) ، تابع مربوطه را از کتابخانه استخراج کرده و آن را در فایل اجرایی نهایی قرار می داد. به این روش پیوند ایستا (Static Linking) گفته می شود. کتابخانه های زبان C مثال خوبی برای روش فوق هستند. اشکال این روش در این است که شما تابع یکسانی را در برنامه های استفاده کننده دارید که فضای دیسک شما را بابت نگهداری کپی های یکسان از یک تابع تلف می کند. این روش برای Dos قابل قبول است زیرا معمولاً در این سیستم عامل یک برنامه در حافظه فعال است و از آن استفاده می کند پس این امر باعث اتلاف حافظه سیستم نمی گردد.

در ویندوز اوضاع بسیار وخیم تر است زیرا در آن واحد چندین برنامه در حال اجرا هستند و در صورت استفاده از روش مذکور ، حافظه سیستم به سرعت مصرف می شود. راه حل ویندوز برای برطرف کردن این مشکل کتابخانه هایی با لینک پویا (Dynamic Link Libraries) است که به اختصار به آنها dll گفته می شود. در واقع dll ها به عنوان مخازنی برای ذخیره و نگهداری توابع و منابع محسوب می شوند. اگر شما چندین نمونه از برنامه خود را در آن واحد اجرا کنید ، ویندوز تنها یک نمونه از dll هایی که برنامه شما به آنها نیاز دارد را به حافظه بارگذاری می کند. حال در این مورد بیشتر توضیح می دهیم.

در واقع هر کدام از برنامه هایی که از یک dll استفاده می کنند کپی اختصاصی از آن dll را برای خود دارند. به نظر می رسد که با این وجود باید کپی های متعددی از یک dll درون حافظه وجود داشته باشد ولی در حقیقت ویندوز کدهای dll را بین تمام برنامه هایی که از آن استفاده می کنند به اشتراک می گذارد. پس در حافظه فیزیکی تنها یک کپی از کدهای dll وجود دارد ولی هر برنامه بخش داده های dll (Data Section) خاص خود را دارد. در ویندوز برنامه ها در زمان اجرا به dll های

مورد نیاز لینک می شوند که با سیستم لینک ایستا در Dos کاملا متفاوت است. به همین علت است که به آن لینک پویا گفته می شود.

در صورت عدم نیاز به dll ، در زمان اجرا می توانید به سادگی آنرا از حافظه برنامه خود خارج کنید. اگر برنامه شما تنها برنامه ای باشد که از آن dll استفاده می کرده است با این عمل dll از حافظه سیستم نیز خارج می شود. ولی اگر هنوز برنامه های دیگری در حال استفاده از آن باشند ، dll در حافظه سیستم باقی خواهد ماند.

ولی حالا برنامه پیوند دهنده در موقع تصحیح آدرس ها برای فایل اجرایی نهایی مسئولیت سنگین تری دارد. زیرا دیگر نمی تواند توابع را استخراج کرده و درون فایل اجرایی قرار دهد پس باید اطلاعات کافی در مورد dll ها و توابعی که در برنامه استفاده شده است را در فایل اجرایی نهایی قرار دهد تا برنامه در موقع اجرا بتواند آنها را پیدا کرده و بارگذاری کند. اینجا است که کار کتابخانه های ورودی (Import Libraries) اهمیت پیدا می کند. این کتابخانه ها اطلاعات لازم برای استفاده از dll ها را در خود دارند. برنامه پیوند دهنده اطلاعات مورد نیاز در مورد dll ها را از این فایل استخراج کرده و درون فایل اجرایی قرار می دهد.

وقتی بارگذار فایلهای اجرایی ویندوز برنامه شما را بارگذاری می کند ، با مشاهده اینکه برنامه به یک dll لینک شده است ، ابتدا به دنبال dll گشته و سپس با تصحیح آدرس ارجاعها به توابع در برنامه آنرا با فضای حافظه اختصاصی برنامه هماهنگ می کند.

شما همچنین می توانید بدون نیاز به بارگذار ویندوز ، خودتان dll های مورد نیاز را بارگذاری کنید. این روش دارای محاسب و معایب خاص خود است. که در زیر آنها را بررسی می کنیم.

- این روش به کتابخانه ورودی برای معرفی dll احتیاجی ندارد و این بدان معنا است که می توانید از dll هایی که کتابخانه ورودی برای آنها وجود ندارد نیز استفاده کنید. ولی باید اطلاعات کافی در مورد توابع داخلی آن dll و نیز پارامترهایی که می گیرند داشته باشید.
- اگر بارگذاری dll ها را بر عهده ویندوز بگذارید ، در صورتی که بارگذار نتواند آنها را پیدا کند ، با صدور پیغام خطایی مانع از اجرای برنامه شما می شود با وجود اینکه ممکن است آن dll برای اجرا برنامه ضروری نباشد. ولی اگر خودتان dll را بارگذاری کنید ، در صورت بروز چنین مشکلی می توانید تنها با یک پیغام خطا کاربر را از آن باخبر کرده و دوباره به کار خود ادامه دهید و روند اجرایی برنامه را دنبال کنید.
- اگر از تابع LoadLibrary برای بارگذاری dll استفاده کنید ، برای فراخوانی هر تابع از آن ، ابتدا باید با استفاده از تابع GetProcessAddress آدرس آنرا پیدا کرده و سپس از آن استفاده کنید که این امر باعث تاثیر اندکی در حجم کد برنامه و سرعت اجرای آن می گردد.

حال که با مزایا و معایب استفاده از تابع LoadLibrary آشنا شدید به سراغ جزئیات ساخت یک dll می رویم.

کد پایین ساختار کلی یک dll را نشان می دهد.

```
.386
.model flat,stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
.data
.code
DllEntry proc hInstDLL:HINSTANCE, reason:DWORD, reserved1:DWORD
        mov  eax,TRUE
        ret
DllEntry Endp

TestFunction proc
    ret
TestFunction endp
End DllEntry


LIBRARY   DLLSkeleton
EXPORTS   TestFunction
```

هر dll باید یک تابع مدخل داشته باشد که ویندوز آنرا در مواقع زیر فراخوانی می کند.

۱- وقتی که dll برای اولین بار بارگذاری می شود.

۲- وقتی dll از حافظه خارج می شود.

۳- وقتی که برنامه استفاده کننده یک صف دستورات جدید ایجاد می کند.

٤- وقتی که برنامه استفاده کننده یک صف دستورات را از بین می برد.

در مورد انتخاب نام تابع مدخل آزاد هستید. این تابع سه پارامتر ورودی می گیرد که آنها را در زیر مشاهده می کنید.

```
DllEntry proc hInstDLL:HINSTANCE, reason:DWORD, reserved1:DWORD
        mov  eax,TRUE
```

```
        ret
DllEntry Endp
```

hInstDLL شماره دسترسی به dll است. این شماره مثل شماره دسترسی برنامه نیست و اگر می خواهید در آینده از آن استفاده کنید باید آنرا ذخیره کنید زیرا دیگر به سادگی نمی توانید آنرا بدست آورید.

Reason می تواند یکی از ٤ مقدار زیر را در بر داشته باشد.

- DLL_PROCESS_ATTACH : dll زمانی این مقدار را دریافت می کند که برای بار اول به حافظه برنامه وارد می شود. شما می توانید از آن به عنوان فرصتی برای انجام کارهای مقدماتی و آماده سازی استفاده کنید.
- DLL_PROCESS_DETACH: این مقدار نشان دهنده اینست که dll در حال خارج شدن از فضای حافظه برنامه است و می توانید از آن به عنوان فرصتی برای پاک سازی و بازگردانی حافظه های رزرو شده به سیستم و کارهای نهایی استفاده کنید.
- DLL_THREAD_DETACH: نشان دهنده اینست که برنامه استفاده کننده یک صف از دستورات را از بین برده است.
- DLL_THREAD_ATTACH : نشان دهنده اینست که برنامه استفاده کننده یک صف از دستورات را ایجاد کرده است.

در صورتیکه می خواهید dll به روند اجرایی خود ادامه دهد ، به eax مقدار True می دهید. در صورت برگرداندن مقدار False ویندوز از بارگذاری dll جلوگیری خواهد کرد. به عنوان مثال اگر کدهای مقدماتی شما که باید مقداری حافظه را به برنامه اختصاص دهند در این کار ناموفق بمانند تابع مدخل با برگرداندن مقدار False می تواند از بارگذاری dll جلوگیری کند.

می توانید توابع خود را قبل یا بعد از تابع مدخل تعریف کنید ولی اگر می خواهید توابع شما از طریق برنامه های دیگر قابل فراخوانی باشند باید نام آنها را در لیست صدور (Export) قرار دهید. این لیست در فایل تعریف (def.) قرار دارد. از این فایل در مرحله نهایی ساخت dll استفاده می شود. حال نگاهی به ساختار این فایل می اندازیم.

```
LIBRARY    DLLSkeleton
EXPORTS    TestFunction
```

بطور معمول باید سطر اول را بنویسید. دستور LIBRARY نام داخلی dll را تعریف می کند که باید آنرا با نام فایل dll مطابقت دهید. دستور EXPORT به برنامه پیوند دهنده می گوید که کدام

یک از توابع می توانند توسط برنامه های دیگر مورد استفاده قرار بگیرند. در این مثال قصد داریم که اجازه استفاده از تابع TestFunction را بدهیم. پس نام آنرا در لیست EXPORT قرار می دهیم.

کار دیگری که باید انجام دهید اضافه کرده دو switch برای لینک کردن dll است. که یکی "/dll" و دیگری" <آدرس فایل تعریف>:def/" است. برای مثال

```
link /DLL /SUBSYSTEM:WINDOWS /DEF:DLLSkeleton.def
/LIBPATH:c:\masm32\DLLSkeleton.obj
```

switch های اسمبلر برای ایجاد فایل obj همانند قبل"/c/coff/cp" هستند. پس از لینک کردن فایل obj ، یک فایل dll و یک فایل Lib ایجاد خواهد شد. فایل Lib همان کتابخانه ورودی است که برنامه های دیگر برای استفاده از فایل dll و توابع درونی این فایل به آن نیاز دارند.

حال با مثال زیر نشان می دهیم که چگونه تابع LoadLibrary را برای بارگذاری و استفاده از یک فایل dll به کار بگیریم.

```
.386
.model flat,stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\user32.lib
.data
LibName db "DLLSkeleton.dll",0
FunctionName db "TestHello",0
DllNotFound db "Cannot load library",0
AppName db "Load Library",0
FunctionNotFound db "TestHello function not found",0
.data?
hLib dd ?
TestHelloAddr dd ?
.code
start:
        invoke LoadLibrary,addr LibName

        .if eax==NULL
                invoke MessageBox,NULL,\
                                  addr DllNotFound,\
                                  addr AppName,\
                                  MB_OK
```

```
        .else
                mov hLib,eax
                invoke GetProcAddress,hLib,addr FunctionName
                .if eax==NULL
                        invoke MessageBox,NULL,\
                                          addr FunctionNotFound,\
                                          addr AppName,\
                                          MB_OK
                .else
                        mov TestHelloAddr,eax
                        call [TestHelloAddr]

                .endif
                invoke FreeLibrary,hLib

        .endif
        invoke ExitProcess,NULL
end start
```

کنترل های عمومی

کدها و فایل‌های مربوط به این قسمت در CD ضمیمه موجود می‌باشد

SourceCodes\Asm32\Chapter17

در این بخش یاد می گیرید که کنترل های عمومی ویندوز چه هستند و چگونه می توانید از آنها استفاده کنید. در حقیقت این بخش یک مقدمه سریع و کوتاه در این زمینه محسوب می شود.

ویندوز ۹۵ با افزودن چندین واسط کاربر استاندارد به ویندوز 3.1 ، رابط گرافیکی کاربر را بسیار غنی تر کرد. تعدادی از آنها مانند ToolBar وStatusBar به طور گسترده قبل از ویندوز ۹۵ نیز به کار گرفته می شدند. ولی برنامه نویسان مجبور بودند که خود آنها را ایجاد کنند. اما اکنون مایکروسافت آنها را در WinNT و Win9X جای داده است. این کنترل ها عبارتند از:

- Toolbar
- Tooltip
- Status bar
- Property sheet
- Property page
- Tree view
- List view
- Animation
- Drag list
- Header
- Hot-key
- Image list
- Progress bar
- Right edit
- Tab
- Trackbar
- Up-down

البته تعداد زیادی از این نوع کنترل ها وجود دارند ولی بارگذاری آنها در حافظه باعث اتلاف منابع سیستم می شود. همه آنها به جز کنترل RichEdit در فایل Comctl32.dll قرار دارند و برنامه ها می توانند در صورت نیاز از این کنترل ها استفاده کنند.

InitCommonControls نام تابعی در Comctl32.dll است که در صورت وجود هر گونه ارجاع به آن در برنامه ، فایل Comctl32.dll به طور خودکار توسط ویندوز به حافظه برنامه شما بارگذاری می شود. در حقیقت خود این تابع کار خاصی را انجام نمی دهد.

کنترل های عمومی نیز مانند کنترل های فرزند بوسیله کلاسهای خاص خود ایجاد می شوند. RichEdit در این زمینه با کنترل های دیگر کاملا متفاوت است و برای ایجاد و استفاده از آن باید از تابع LoadLibrary کمک بگیرید. حال نحوه ایجاد این کنترل ها را بررسی می کنیم. شما می توانید از یک ویرایشگر منبع برای اضافه کردن آنها به دیالوگ باکس خود استفاده کنید و یا اینکه خودتان آنها را ایجاد کنید.

تقریبا تمام کنترل های عمومی با استفاده از توابع CreateWindowEx یا CreateWindow ایجاد می شوند. ولی برخی از آنها توابع خاصی برای ایجاد خود دارند که عملکرد آنها از توابع CreateWindow بسیار ساده تر است. در زیر نام این توابع را مشاهده می کنید.

- CreateToolbarEx
- CreateStatusWindow
- CreatePropertySheetPage
- PropertySheet
- ImageList_Create

برای ایجاد کنترل های عمومی لازم است نام کلاسهای آنها را بدانید. در زیر لیستی از این کنترل ها را به همراه نام کلاسهای مربوطه مشاهده می کنید.

Class Name	Common Control
ToolbarWindow32	Toolbar
tooltips_class32	Tooltip
msctls_statusbar32	Status bar
SysTreeView32	Tree view
SysListView32	List view
SysAnimate32	Animation
SysHeader32	Header
msctls_hotkey32	Hot-key
msctls_progress32	Progress bar
RICHEDIT	Rich edit
msctls_updown32	Up-down
SysTabControl32	Tab

توابع سازنده خاص خود را دارند. Property Sheet وProperty Page, ImageList DragListBox یک نوع List Box است و به همین دلیل دارای کلاسهای خاص خود نیست. این کنترل ها می توانند از سبکهای پنجره های معمولی مانند WS_CHILD استفاده کنند. ولی آنها دارای سبکهای خاص خود نیز هستند. برای مثال TVS_XXXX برای کنترل TreeView و LVS_XXXX برای کنترل ListView. بهترین مرجع در این موارد Win32 API Refrence است.

حال که می دانیم چگونه کنترل های عمومی را ایجاد کنیم ، به سراغ نحوه ارتباط آنها با پنجره پدر خود می رویم. بر خلاف کنترل‌های فرزند ، کنترل های عمومی به جای پیغام WM_COMMAND

از پیغام WM_NOTIFY برای ارتباط با پنجره های پدر خود استفاده می کنند. پیغام های جدید زیادی نیز برای این کنترل ها وجود دارد. برای جزئیات بیشتر در این مورد می توانید به Win32 API Refrence مراجعه کنید.

حال به سراغ مثال این بخش می رویم.

در این مثال نحوه ایجاد و استفاده از Status Bar و Progress Bar را بررسی می کنیم.

```
 .386
.model flat,stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
include \masm32\include\comctl32.inc
includelib \masm32\lib\comctl32.lib
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
WinMain PROTO :DWORD,:DWORD,:DWORD,:DWORD
.const
IDC_PROGRESS equ 1            ; control IDs
IDC_STATUS equ 2
IDC_TIMER  equ 3
.data
db "CommonControlWinClass",0  ClassName
db "Common Control Demo",0  AppName
,0           ProgressClass  db "msctls_progress32"
db "Finished!",0    Message
dd 0    TimerID
.data?
hInstance  HINSTANCE ?
hwndProgress dd ?
hwndStatus dd ?
CurrentStep dd ?
.code
start:
    invoke GetModuleHandle, NULL
    mov    hInstance,eax
    invoke WinMain, hInstance,NULL,NULL, SW_SHOWDEFAULT
    invoke ExitProcess,eax
    invoke InitCommonControls
WinMain proc
hInst:HINSTANCE,hPrevInst:HINSTANCE,CmdLine:LPSTR,CmdShow:DWORD
    LOCAL wc:WNDCLASSEX
```

```
    LOCAL msg:MSG
    LOCAL hwnd:HWND
    mov   wc.cbSize,SIZEOF WNDCLASSEX
    mov   wc.style, CS_HREDRAW or CS_VREDRAW
    mov   wc.lpfnWndProc, OFFSET WndProc
    mov   wc.cbClsExtra,NULL
    mov   wc.cbWndExtra,NULL
    push  hInst
    pop   wc.hInstance
    mov   wc.hbrBackground,COLOR_APPWORKSPACE
    mov   wc.lpszMenuName,NULL
    mov   wc.lpszClassName,OFFSET ClassName
    invoke LoadIcon,NULL,IDI_APPLICATION
    mov   wc.hIcon,eax
    mov   wc.hIconSm,eax
    invoke LoadCursor,NULL,IDC_ARROW
    mov   wc.hCursor,eax
    invoke RegisterClassEx, addr wc
    invoke CreateWindowEx, WS_EX_CLIENTEDGE,\
                           ADDR ClassName,\
                           ADDR AppName,\
                           WS_OVERLAPPED + WS_CAPTION +\
                           WS_SYSMENU + WS_MINIMIZEBOX+\
                           WS_MAXIMIZEBOX + WS_VISIBLE,\
                           CW_USEDEFAULT,\
                           CW_USEDEFAULT,\
                           CW_USEDEFAULT,\
                           CW_USEDEFAULT,\
                           NULL,\
                           NULL,\
                           hInst,\
                           NULL
    mov   hwnd,eax
    .while TRUE
         invoke GetMessage, ADDR msg,NULL,0,0
        .BREAK .IF (!eax)
        invoke TranslateMessage, ADDR msg
        invoke DispatchMessage, ADDR msg
    .endw
    mov eax,msg.wParam
    ret
WinMain endp
WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
    .if uMsg==WM_CREATE
         invoke CreateWindowEx,NULL,ADDR ProgressClass,NULL,\
            WS_CHILD+WS_VISIBLE,100,\
            200,300,20,hWnd,IDC_PROGRESS,\
            hInstance,NULL
        mov hwndProgress,eax
        mov eax,1000  ; the lParam of PBM_SETRANGE message contains the range
        mov CurrentStep,eax
        shl eax,16    ; the high range is in the high word
```

```
        invoke SendMessage,hwndProgress,PBM_SETRANGE,0,eax
        invoke SendMessage,hwndProgress,PBM_SETSTEP,10,0
        invoke CreateStatusWindow,WS_CHILD+WS_VISIBLE,\
                                  NULL,\
                                  hWnd,\
                                  IDC_STATUS

        mov hwndStatus,eax
        invoke SetTimer,hWnd,IDC_TIMER,100,NULL
        mov TimerID,eax
    .elseif uMsg==WM_DESTROY
        invoke PostQuitMessage,NULL
        .if TimerID!=0
            invoke KillTimer,hWnd,TimerID
        .endif
    .elseif uMsg==WM_TIMER        ; when a timer event occurs
        invoke SendMessage,hwndProgress,PBM_STEPIT,0,0
        sub CurrentStep,10
        .if CurrentStep==0
            invoke KillTimer,hWnd,TimerID
            mov TimerID,0
            invoke SendMessage,hwndStatus,SB_SETTEXT,0,addr Message
            invoke MessageBox,hWnd,\
                              addr Message,\
                              addr AppName,\
                              MB_OK+MB_ICONINFORMATION

            invoke SendMessage,hwndStatus,SB_SETTEXT,0,0
            invoke SendMessage,hwndProgress,PBM_SETPOS,0,0
        .endif
    .else
        invoke DefWindowProc,hWnd,uMsg,wParam,lParam
        ret
    .endif
    xor eax,eax
    ret
WndProc endp
end start
```

حال به بررسی کدهای این برنامه می پردازیم.

```
invoke WinMain, hInstance,NULL,NULL, SW_SHOWDEFAULT
invoke ExitProcess,eax
invoke InitCommonControls
```

فراخوانی تابع InitCommonControls را بعد از ExitProcess قرار داده ایم فقط به خاطر اینکه نشان دهیم این تابع تنها به عنوان یک ارجاع به Comctl32.dll به کار می رود و در صورت عدم اجرا نیز کنترل های عمومی به درستی کار خواهند کرد.

```
.if uMsg==WM_CREATE
invoke CreateWindowEx, WS_EX_CLIENTEDGE,\
                       ADDR ClassName,\
                       ADDR AppName,\
                       WS_OVERLAPPED + WS_CAPTION +\
                       WS_SYSMENU + WS_MINIMIZEBOX+\
                       WS_MAXIMIZEBOX + WS_VISIBLE,\
                       CW_USEDEFAULT,\
                       CW_USEDEFAULT,\
                       CW_USEDEFAULT,\
                       CW_USEDEFAULT,\
                       NULL,\
                       NULL,\
                       hInst,\
                       NULL
    mov hwndProgress,eax
```

در این مرحله کنترل های عمومی را ایجاد می کنیم. متذکر می شویم که تابع CreateWindowEx شماره دسترسی به پنجره پدر و ID کنترل را به عنوان پارامتر ورودی دریافت می کند ولی به دلیل داشتن شماره دسترسی به پنجره کنترل ، از این ID دیگر استفاده نمی کنیم. کلیه کنترل های فرزند و کنترلهای عمومی باید از سبک WS_CHILD استفاده کنند.

```
    mov eax,1000
    mov CurrentStep,eax
    shl eax,16
    invoke SendMessage,hwndProgress,PBM_SETRANGE,0,eax
    invoke SendMessage,hwndProgress,PBM_SETSTEP,10,0
```

پس از ایجاد Progress Bar باید محدوده آنرا مشخص کنیم. حالت پیش فرض از صفر تا صد است. در صورت نیاز می توانید با استفاده از پیغام PBM_SETRANGE محدوده مورد نظر خود را مشخص کنید. قسمت lParam از این پیغام محدوده را مشخص می کند ، به اینصورت که مقدار ماکزیمم در قسمت High word و مقدار مینیمم در قسمت Low word این متغیر قرار می گیرد. با استفاده از پیغام PBM_SETSTEP می توانید مقدار پرش را مشخص کنید. در این مثال این مقدار را 10 در نظر گرفته ایم یعنی با هر پیغام PBM_STEPIT که به Progress Bar فرستاده می شود ، نوار آبی به اندازه 10 واحد افزایش طول پیدا می کند. همچنین می توانید با فرستادن پیغام PBM_SETPOS طول نوار آبی را به طور دلخواه تنظیم کنید که این امر کنترل بهتری را نسبت به حالت قبل به شما می دهد.

```
invoke CreateStatusWindow,WS_CHILD+WS_VISIBLE,\
                          NULL,\
                          hWnd,\
                          IDC_STATUS

mov hwndStatus,eax
invoke SetTimer,hWnd,\
                IDC_TIMER,\
                100,\
                NULL        ; create a timer

mov TimerID,eax
```

در این مرحله با استفاده از تابع CreateStatusWindow یک Status Bar ایجاد می کنیم. پس از ایجاد این کنترل ، یک Timer نیز ایجاد می کنیم. در این مثال قصد داریم مقدار Progress Bar را در بازه های زمانی 100 میلی ثانیه افزایش دهیم. پس به یک کنترل Timer احتیاج داریم. در زیر ، پیش تعریف تابع SetTimer را مشاهده می کنید که از آن برای ایجاد یک Timer استفاده می کنیم.

```
SetTimer PROTO hWnd:DWORD, TimerID:DWORD, TimeInterval:DWORD,
lpTimerProc:DWORD
```

hWnd شماره دسترسی به پنجره پدر است.

TimeID شناسه غیر صفر برای تایمر.

TimerInterval بر حسب میلی ثانیه است و بازه های زمانی timer را مشخص می کند.

lpTimerProc آدرس تابع Timer که وقتی زمان Intervial به پایان رسید باید فراخوانی شود. در صورت استفاده از Null به جای آدرس تابع ، تایمر پیغام WM_TIMER را به پنجره پدر خواهد فرستاد.

اگر این فراخوانی موفقیت آمیز باشد مقدار ID بازگردانده می شود. در غیر این صورت مقدار صفر بازگردانده می شود. به همین دلیل است که مقدار TimeID را نباید صفر قرار دهید.

```
    .elseif uMsg==WM_TIMER
        invoke SendMessage,hwndProgress,PBM_STEPIT,0,0
        sub CurrentStep,10
        .if CurrentStep==0
            invoke KillTimer,hWnd,TimerID
            mov TimerID,0
            invoke SendMessage,hwndStatus,SB_SETTEXT,0,addr Message
            invoke MessageBox,hWnd,\
                              addr Message,\
                              addr AppName,\
                              MB_OK+MB_ICONINFORMATION
invoke SendMessage,hwndStatus,SB_SETTEXT,0,0
            invoke SendMessage,hwndProgress,PBM_SETPOS,0,0
        .endif
```

وقتی زمان مشخص شده در Interval به پایان رسید ، تایمر پیغام WM_TIMER را به پنجره پدر می فرستد. کدهایی را که می خواهید در این مرحله اجرا شوند باید در این قسمت قرار دهید. در این مثال همزمان با افزایش Progress Bar ، چک می کنیم که آیا این مقدار به مقدار ماکزیمم رسیده است یا نه. اگر پاسخ مثبت باشد بوسیله تابع KillTimer تایمر را از بین می بریم و با استفاده از پیغام SB_SETTEXT متنی را درون Status Bar چاپ می کنیم. سپس یک MessageBox را به کاربر نشان می دهیم. با انتخاب دکمه OK توسط کاربر ، ابتدا متن درون Status Bar را پاک کرده و سپس Progress Bar را به حالت اولیه خود در می آوریم.

Subclassing

کدها و فایلهای مربوط به این قسمت در CD ضمیمه موجود می‌باشد

SourceCodes\Asm32\Chapter18

در این بخش می آموزیم که چگونه از Subclassing در ویندوز استفاده کرده و از مزایای این تکنیک بهره مند شویم.

اگر در ویندوز برنامه نویسی کرده باشید احتمالا با شرایطی مواجه شده اید که یک کنترل قابلیتهای مورد نظر را دارد ولی برای استفاده در برنامه باید آنها را تغییر دهید. به عنوان مثال آیا تا کنون با شرایطی مواجه شده اید که نیاز به یک کنترل Edit داشته باشید که برخی از کاراکتر های خاص را قبول نکند؟ یک راه حل اینست که خودتان پنجره های مورد نیاز را ایجاد کنید ولی این کار بسیار سخت و وقت گیر است. در این زمینه Subclassing می تواند جوابگوی مشکلات شما باشد. حال با یک مثال این موضوع را روشن تر می کنیم.

فرض کنید که شما به یک کنترل Edit احتیاج دارید که فقط اعداد در مبنای ۱۶ را قبول می کند. در صورت استفاده از کنترل معمولی کنترل مستقیمی روی کاراکترهای ورودی نمی توانید داشته باشید. در حقیقت شما نیاز به یک توانایی در درون این کنترل دارید که در هنگام تایپ ، کاراکترهای ورودی را با خواسته شما که اعداد مبنای ۱۶ هستند تطبیق دهد.

هنگامی که کاربر چیزی را تایپ می کند ، ویندوز پیغام WM_CHAR را به پروسیجر پنجره کنترل Edit می فرستد. این پروسیجر در داخل ویندوز قرار گرفته است و ما نمی توانیم هیچ تغییری در آن ایجاد کنیم ولی می توانیم پیغام های مورد نظر را به پروسیجر اصلی بفرستید. با این روش پروسیجر ما بین ویندوز و کنترل Edit قرار خواهد گرفت.

```
Before Subclassing
> edit control's window procedure ---Windows
After Subclassing
Windows ---> our window procedure ---> edit control's window
procedure
```

متذکر می شویم که استفاده از Subclassing محدود به کنترل ها نیست و برای هر نوع پنجره ای می تواند مورد استفاده قرار گیرد.

همانطور که می دانید آدرس پروسیجر پنجره در متغیر lpfnWndProc قرار دارد که از اعضای رکورد WNDCLASSEX است. اگر بتوانیم مقدار این عضو از کلاس WNDCLASSEX را با

آدرس پروسیجر خودمان جایگزین کنیم ، از این به بعد پیغام ها به پروسیجر پنجره ما فرستاده خواهند شد. این کار را با استفاده از تابع SetWindowLong انجام خواهیم داد.

```
SetWindowLong PROTO hWnd:DWORD, nIndex:DWORD, dwNewLong:DWORD
```

hWnd شماره دسترسی پنجره مورد نظر را مشخص می کند.

nIndex پارامتری که قصد تغییر آن را داریم مشخص می کند که می توانید یکی از مقادیر زیر باشد.

- GWL_EXSTYLE: یک سبک اضافی جدید برای پنجره.
- GWL_STYLE: یک سبک جدید برای پنجره.
- GWL_WNDPROC: یک آدرس جدید برای زیر برنامه پنجره.
- GWL_HINSTANCE: یک شماره جدید دسترسی به برنامه.
- GWL_ID: یک شناسه جدید برای پنجره.
- GWL_USERDATA: یک مقدار 32 بیتی که اطلاعات اضافی کاربر را در خود دارد.

dwNewLong مقدار جدید پارامتر مورد نظر را مشخص می کند.

پس کار ما بسیار ساده است. ابتدا یک پروسیجر پنجره جدید برای کنترل Edit ایجاد می کنیم. سپس تابع SetWindowLong را با مقدار GWL_WNDPROC برای متغیر nIndex فراخوانی کرده و آدرس پروسیجر جدید را به عنوان پارامتر سوم به آن می فرستیم. اگر تابع عملیات خود را با موفقیت انجام دهد مقدار بازگشتی یک مقدار 32 بیتی است که آدرس پروسیجر اصلی پنجره را در خود دارد. این مقدار را برای استفاده های بعدی ذخیره می کنیم. یادآوری می کنیم که پیغام های زیادی وجود دارند که ما کاری به آنها نداریم و آنها را با استفاده از تابع CallWindowProc به پروسیجر اصلی می فرستیم.

```
CallWindowProc PROTO lpPrevWndFunc :DWORD,\
                     hWnd          :DWORD,\
                     Msg           :DWORD,\
                     wParam        :DWORD,\
                     lParam        :DWORD
```

lpPrevWndFunc آدرس پروسیجر اصلی پنجره را مشخص می کند.

چهار پارامتر دیگر دقیقا همان هایی هستند که به پروسیجر پنجره ما فرستاده شده اند. و ما هم ، آنها را به همان صورت به تابع CallWindowProc می فرستیم.

```
.386
.model flat,stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
include \masm32\include\comctl32.inc
includelib \masm32\lib\comctl32.lib
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
WinMain PROTO :DWORD,:DWORD,:DWORD,:DWORD
EditWndProc PROTO :DWORD,:DWORD,:DWORD,:DWORD
.data
ClassName  db "SubclassWinClass",0
AppName    db "Subclassing Demo",0
EditClass  db "EDIT",0
Message  db "You pressed Enter in the text box!",0
.data?
hInstance  HINSTANCE ?
hwndEdit dd ?
OldWndProc dd ?
.code
start:
    invoke GetModuleHandle, NULL
    mov    hInstance,eax
    invoke WinMain, hInstance,NULL,NULL, SW_SHOWDEFAULT
    invoke ExitProcess,eax
WinMain proc
hInst:HINSTANCE,hPrevInst:HINSTANCE,CmdLine:LPSTR,CmdShow:DWORD
    LOCAL wc:WNDCLASSEX
    LOCAL msg:MSG
    LOCAL hwnd:HWND
    mov   wc.cbSize,SIZEOF WNDCLASSEX
    mov   wc.style, CS_HREDRAW or CS_VREDRAW
    mov   wc.lpfnWndProc, OFFSET WndProc
    mov   wc.cbClsExtra,NULL
    mov   wc.cbWndExtra,NULL
    push  hInst
    pop   wc.hInstance
    mov   wc.hbrBackground,COLOR_APPWORKSPACE
    mov   wc.lpszMenuName,NULL
    mov   wc.lpszClassName,OFFSET ClassName
    invoke LoadIcon,NULL,IDI_APPLICATION
    mov   wc.hIcon,eax
```

```
    mov   wc.hIconSm,eax
    invoke LoadCursor,NULL,IDC_ARROW
    mov   wc.hCursor,eax
    invoke RegisterClassEx, addr wc
    invoke CreateWindowEx, WS_EX_CLIENTEDGE,\
                           ADDR ClassName,\
                           ADDR AppName,\
                           WS_OVERLAPPED + WS_CAPTION +\
                           WS_SYSMENU + WS_MINIMIZEBOX +\
                           WS_MAXIMIZEBOX + WS_VISIBLE,\
                           CW_USEDEFAULT,\
                           CW_USEDEFAULT,\
                           350,\
                           200,\
                           NULL,\
                           NULL,\
                           hInst,\
                           NULL
    mov   hwnd,eax
    .while TRUE
        invoke GetMessage, ADDR msg,NULL,0,0
        .BREAK .IF (!eax)
        invoke TranslateMessage, ADDR msg
        invoke DispatchMessage, ADDR msg
    .endw
    mov eax,msg.wParam
    ret
WinMain endp
WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
    .if uMsg==WM_CREATE
        invoke CreateWindowEx,WS_EX_CLIENTEDGE,\
                              ADDR EditClass,\
                              NULL,\
                              WS_CHILD + WS_VISIBLE + WS_BORDER ,\
                              20,\
                              20,\
                              300,\
                              25,\
                              hWnd,\
                              NULL,\
                              hInstance,\
                              NULL
        mov hwndEdit,eax
        invoke SetFocus,eax
        ;---------------------------------------
        ; Subclass it!
        ;---------------------------------------
        invoke SetWindowLong,hwndEdit,GWL_WNDPROC,addr EditWndProc
        mov OldWndProc,eax
    .elseif uMsg==WM_DESTROY
        invoke PostQuitMessage,NULL
    .else
```

```
        invoke DefWindowProc,hWnd,uMsg,wParam,lParam
        ret
    .endif
    xor eax,eax
    ret
WndProc endp
EditWndProc PROC hEdit:DWORD,uMsg:DWORD,wParam:DWORD,lParam:DWORD
    .if uMsg==WM_CHAR
        mov eax,wParam
        .if (al >= "0" && al <= "9") || \
            (al >= "A" && al <= "F") || \
            (al>="a" && al<="f")     || \
            al==VK_BACK
            .if al >= "a" && al <= "f"
                sub al,20h
            .endif
            invoke CallWindowProc,OldWndProc,hEdit,uMsg,eax,lParam
            ret
        .endif
    .elseif uMsg==WM_KEYDOWN
        mov eax,wParam
        .if al==VK_RETURN
            invoke MessageBox,hEdit,\
                              addr Message,\
                              addr AppName,\
                              MB_OK + MB_ICONINFORMATION
            invoke SetFocus,hEdit
        .else
            invoke CallWindowProc,OldWndProc,\
                                  hEdit,\
                                  uMsg,\
                                  wParam,\
                                  lParam

            ret
        .endif
    .else
        invoke CallWindowProc,OldWndProc,hEdit,uMsg,wParam,lParam
        ret
    .endif
    xor eax,eax
    ret
EditWndProc endp
```

حال به بررسی کدهای این برنامه می پردازیم.

```
        invoke SetWindowLong,hwndEdit,GWL_WNDPROC,addr EditWndProc
        mov OldWndProc,eax
```

پس از ایجاد کنترل Edit ، با استفاده از تابع SetWindowLong آدرس پروسیجر پنجره آنرا با آدرس پروسیجر جدید جایگزین می کنیم. متذکر می شویم که آدرس پروسیجر اصلی را برای استفاده در تابع CallWindowProc ذخیره می کنیم.

```
.if uMsg==WM_CHAR
      mov eax,wParam
      .if (al>="0" && al<="9") || \
          (al>="A" && al<="F") || \
          (al>="a" && al<="f") || \
          al==VK_BACK
          .if al>="a" && al<="f"
              sub al,20h
          .endif
          invoke CallWindowProc,OldWndProc,hEdit,uMsg,eax,lParam
          ret
      .endif
```

در داخل تابع EditWndProc پیغامهای WM_CHAR را فیلتر می کنیم. در صورتی که کاراکتر ورودی 0-9 یا a-f بود ، با فرستادن پیغام به پروسیجر اصلی آنرا می پذیریم. اگر کاراکتر از حروف کوچک بود ، با اضافه کردن مقدار 20h آنرا به حروف بزرگ تبدیل می کنیم. و در صورتی که کاراکتر ورودی مورد قبول نبود آنرا به پروسیجر اصلی نمی فرستیم و در حقیقت آنرا به دور می اندازیم. در نتیجه اگر کاربر چیزی غیر از کاراکترهای مجاز تایپ کند نشان داده نخواهد شد.

```
.elseif uMsg==WM_KEYDOWN
    mov eax,wParam
    .if al==VK_RETURN
        invoke MessageBox,hEdit,\
                          addr Message,\
                          addr AppName,\
                          MB_OK + MB_ICONINFORMATION
        invoke SetFocus,hEdit
    .else
        invoke CallWindowProc,OldWndProc,\
                              hEdit,\
                              uMsg,\
                              wParam,\
                              lParam

        ret
    .end
```

پروسیجر EditWndProc پیغام WM_KEYDOWN را چک می کند و در صورت فشرده شدن کلید Enter (VK_RETURN) یک MessageBox را نمایش می دهد.

شما می توانید از تکنیک Subclassing به منظور ایجاد کنترل روی سایر پنجره ها نیز استفاده کنید.

Superclassing

کدها و فایل‌های مربوط به این قسمت در CD ضمیمه موجود می‌باشد

SourceCodes\Asm32\Chapter19

در این فصل مطالبی را درباره تکنیک Superclassing در ویندوز خواهید آموخت. همچنین نحوه استفاده از کلید Tab را برای تغییر فوکوس روی کنترل ها بررسی خواهیم کرد.

احتمالا در برنامه نویسی با شرایطی مواجه شده اید که نیاز به یک سری کنترل داشته باشید که با یکدیگر تفاوتهای جزیی دارند. به عنوان مثال ممکن است که نیاز به ده کنترل Edit داشته باشید که فقط اعداد را دریافت کنند. راههای مختلفی برای رسیدن به این هدف وجود دارد.

۱- خودتان کلاس مورد نظر را ایجاد کنید.

۲- کنترل های مورد نظر را ساخته و تمام آنها را Subclass کنید.

۳- استفاده از روش Superclassing.

روش اول بسیار خسته کننده است زیرا مجبور خواهید بود کلیه توابع مربوط به این کنترل ها را خود بنویسید. روش دوم نسبت به روش اول بهتر است ولی باز هم به کار زیادی نیاز دارد. این روش برای تعداد کمی از کنترلها بسیار مناسب است اما انجام این کار بر روی تعداد زیادی از این کنترلها شبیه به یک کابوس است. در این گونه موارد می توانید از تکنیک Superclassing استفاده کنید. در این روش ابتدا تغییرات مورد نظر را در کلاس پنجره ایجاد کرده و سپس کنترل های خود را با استفاده از کلاس جدید می سازیم.

در زیر مراحل این روش را بررسی خواهیم کرد.

۱- فراخوانی تابع GetClassInfoEx برای بدست آوردن اطلاعات موجود در کلاس پنجره ای که می خواهید عمل Superclassing را روی آن انجام دهید. این تابع اشاره گری به رکورد WNDCLASSEX را به عنوان پارامتر های ورودی دریافت کرده و در صورت موفقیت ، اطلاعات مورد نظر را در آن قرار خواهد داد.

۲- تغییر اعضا کلاس WNDCLASSEX. در این مثال ما در سه عضو از این کلاس تغییراتی خواهیم داد.

- hInstance: شماره دسترسی برنامه را در این عضو قرار خواهیم داد.

- lpszClassName: اشاره گر به نام کلاس جدید را در این عضو قرار خواهیم داد.
- lpfnWndProc: اشاره گر به پروسیجر جدید پنجره در این عضو قرار خواهیم داد.

۳- رجیستر کردن رکورد تغییر یافته.

٤- ایجاد پنجره از روی کلاس جدید.

اگر می خواهید تعداد زیادی کنترل با خصوصیات یکسان داشته باشید بهتر است از این روش استفاده کنید.

```
.386
.model flat,stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
WM_SUPERCLASS equ WM_USER+5
WinMain PROTO :DWORD,:DWORD,:DWORD,:DWORD
EditWndProc PROTO :DWORD,:DWORD,:DWORD,:DWORD
.data
ClassName  db "SuperclassWinClass",0
AppName    db "Superclassing Demo",0
EditClass  db "EDIT",0
OurClass   db "SUPEREDITCLASS",0
Message    db "You pressed the Enter key in the text box!",0
.data?
hInstance dd ?
hwndEdit dd 6 dup(?)
OldWndProc dd ?
.code
start:
    invoke GetModuleHandle, NULL
    mov    hInstance,eax
    invoke WinMain, hInstance,NULL,NULL, SW_SHOWDEFAULT
    invoke ExitProcess,eax
WinMain proc
hInst:HINSTANCE,hPrevInst:HINSTANCE,CmdLine:LPSTR,CmdShow:DWORD
    LOCAL wc:WNDCLASSEX
    LOCAL msg:MSG
    LOCAL hwnd:HWND
    mov wc.cbSize,SIZEOF WNDCLASSEX
    mov wc.style, CS_HREDRAW or CS_VREDRAW
```

```
    mov wc.lpfnWndProc, OFFSET WndProc
    mov wc.cbClsExtra,NULL
    mov wc.cbWndExtra,NULL
    push hInst
    pop wc.hInstance
    mov wc.hbrBackground,COLOR_APPWORKSPACE
    mov wc.lpszMenuName,NULL
    mov wc.lpszClassName,OFFSET ClassName
    invoke LoadIcon,NULL,IDI_APPLICATION
    mov wc.hIcon,eax
    mov wc.hIconSm,eax
    invoke LoadCursor,NULL,IDC_ARROW
    mov wc.hCursor,eax
    invoke RegisterClassEx, addr wc
    invoke CreateWindowEx, WS_EX_CLIENTEDGE + WS_EX_CONTROLPARENT,\
                           ADDR ClassName,\
                           ADDR AppName,\
                           WS_OVERLAPPED + WS_CAPTION +\
                           WS_SYSMENU + WS_MINIMIZEBOX +\
                           WS_MAXIMIZEBOX + WS_VISIBLE,\
                           CW_USEDEFAULT,\
                           CW_USEDEFAULT,\
                           350,\
                           220,\
                           NULL,\
                           NULL,\
                           hInst,\
                           NULL
    mov hwnd,eax
    .while TRUE
        invoke GetMessage, ADDR msg,NULL,0,0
        .BREAK .IF (!eax)
        invoke TranslateMessage, ADDR msg
        invoke DispatchMessage, ADDR msg
    .endw
     mov eax,msg.wParam
    ret
WinMain endp
WndProc proc uses ebx edi hWnd:HWND, uMsg:UINT, wParam:WPARAM,
lParam:LPARAM
    LOCAL wc:WNDCLASSEX
    .if uMsg==WM_CREATE
        mov wc.cbSize,sizeof WNDCLASSEX
        invoke GetClassInfoEx,NULL,addr EditClass,addr wc
        push wc.lpfnWndProc
        pop OldWndProc
        mov wc.lpfnWndProc, OFFSET EditWndProc
        push hInstance
        pop wc.hInstance
        mov wc.lpszClassName,OFFSET OurClass
        invoke RegisterClassEx, addr wc
        xor ebx,ebx
```

```
        mov edi,20
        .while ebx<6
            invoke CreateWindowEx,WS_EX_CLIENTEDGE,\
                                  ADDR OurClass,\
                                  NULL,\
                                  WS_CHILD + WS_VISIBLE +
                                  WS_BORDER,\
                                  20,\
                                  edi,\
                                  300,\
                                  25,\
                                  hWnd,\
                                  ebx,\
                                  hInstance,\
                                  NULL
            mov dword ptr [hwndEdit+4*ebx],eax
            add edi,25
            inc ebx
        .endw
        invoke SetFocus,hwndEdit
    .elseif uMsg==WM_DESTROY
        invoke PostQuitMessage,NULL
    .else
        invoke DefWindowProc,hWnd,uMsg,wParam,lParam
        ret
    .endif
    xor eax,eax
    ret
WndProc endp
EditWndProc PROC hEdit:DWORD,uMsg:DWORD,wParam:DWORD,lParam:DWORD
    .if uMsg==WM_CHAR
        mov eax,wParam
        .if (al>="0" && al<="9") || \
(al>="A" && al<="F") || \
(al>="a" && al<="f") || \
al==VK_BACK
            .if al>="a" && al<="f"
               sub al,20h
            .endif
            invoke CallWindowProc,OldWndProc,hEdit,uMsg,eax,lParam
            ret
        .endif
    .elseif uMsg==WM_KEYDOWN
        mov eax,wParam
        .if al==VK_RETURN
            invoke MessageBox,hEdit,\
                              addr Message,\
                              addr AppName,\
                              MB_OK + MB_ICONINFORMATION
            invoke SetFocus,hEdit
        .elseif al==VK_TAB
            invoke GetKeyState,VK_SHIFT
```

```
            test eax,80000000
            .if ZERO?
                invoke GetWindow,hEdit,GW_HWNDNEXT
                .if eax==NULL
                    invoke GetWindow,hEdit,GW_HWNDFIRST
                .endif
            .else
                invoke GetWindow,hEdit,GW_HWNDPREV
                .if eax==NULL
                    invoke GetWindow,hEdit,GW_HWNDLAST
                .endif
            .endif
            invoke SetFocus,eax
            xor eax,eax
            ret
        .else
            invoke CallWindowProc,OldWndProc,\
                                  hEdit,\
                                  uMsg,\
                                  wParam,\
                                  lParam

            ret
        .endif
    .else
        invoke CallWindowProc,OldWndProc,hEdit,uMsg,wParam,lParam
        ret
    .endif
    xor eax,eax
    ret
EditWndProc endp
end start
```

حال به بررسی کدهای این برنامه می پردازیم.

برنامه یک پنجره ساده با شش کنترل Edit تغییر یافته در منطقه کاری ایجاد می کند. این کنترل ها فقط اعداد در مبنای ۱۶ را قبول می کنند. در حقیقت این مثال را با ایجاد تغییرات اندکی در مثال بخش قبل ایجاد کرده ایم.

```
    .if uMsg==WM_CREATE
         mov wc.cbSize,sizeof WNDCLASSEX
        invoke GetClassInfoEx,NULL,addr EditClass,addr wc
```

در ابتدا باید رکورد WNDCLASSEX را با اطلاعات موجود در کلاسی که می خواهیم عمل Superclassing را روی آن انجام دهیم ، پر کنیم. به خاطر داشته باشید که قبل از فراخوانی تابع

GetClassInfoEx باید عضو cbSize از رکورد WNDCLASSEX را نیز مقدار دهی کنید. در غیر این صورت احتمالا اطلاعات ناقصی را دریافت خواهید کرد. پس از بازگشت تابع ، رکورد WC با اطلاعات لازم برای ایجاد کلاس جدید پر خواهد شد.

```
push wc.lpfnWndProc
pop OldWndProc
mov wc.lpfnWndProc, OFFSET EditWndProc
push hInstance
pop wc.hInstance
mov wc.lpszClassName,OFFSET OurClass
```

اکنون باید بعضی از اعضای WC را تغییر دهیم. به دلیل اینکه قصد داریم پروسیجر پنجره خود را با پروسیجر اصلی پیوند دهیم ، باید آدرس آنرا در یک متغیر ذخیره کنیم تا بتوانیم توسط تابع CallWindowProc آنرا فراخوانی کنیم. این تکنیک همانند Subclassing است با این تفاوت که WNDCLASSEX را مستقیما بدون استفاده از تابع SetWindowLong تغییر می دهیم. دو عضو دیگر را نیز باید تغییر دهیم. در غیر این صورت نخواهیم توانست کلاس جدید را رجیستر کنیم. این دو عضو عبارتند از: lpsClassName و hInstance . مقدار hInstance را باید با شماره دسترسی برنامه خود جایگزین کرده و برای کلاس جدید خود یک نام انتخاب کنید.

```
invoke RegisterClassEx, addr wc
```

حال کلاس را رجیستر کرده ایم و می توانیم بر پایه آن پنجره ها را ایجاد کنیم.

```
xor ebx,ebx
mov edi,20
.while ebx<6
 invoke CreateWindowEx,WS_EX_CLIENTEDGE,\
                       ADDR OurClass,\
                       NULL,\
                       WS_CHILD + WS_VISIBLE + WS_BORDER,\
                       20,\
                       edi,\
                       300,\
                       25,\
                       hWnd,\
                       ebx,\
                       hInstance,\
                       NULL
    mov dword ptr [hwndEdit+4*ebx],eax
    add edi,25
    inc ebx
.endw
invoke SetFocus,hwndEdit
```

از eax به عنوان شمارنده ای برای تعداد پنجره های ایجاد شده استفاده می کنیم. از edi نیز برای هماهنگ کردن مختصات پنجره ها استفاده می کنیم. هنگامی که پنجره ای ایجاد می شود ، شماره دسترسی آن در آرایه ای از DWORD ذخیره می شود. زمانیکه تمام پنجره ها ایجاد شدند ، فوکوس را به اولین پنجره منتقل می کنیم. در این مرحله شش کنترل Edit داریم که فقط اعداد در مبنای ١٦ را قبول می کنند. پروسیجر پنجره همانند مثال بخش قبل وظیفه فیلتر کردن کاراکترهای ورودی را بر عهده دارد.

به طور معمول اگر کنترل ها را بر روی یک دیالوگ باکس قرار دهید، عملیات حرکت بر روی آنها توسط کلید Tab صورت می گیرد. به این صورت که کلید Tab فوکوس را به کنترل بعد منتقل کرده و کلیدهای Shift+Tab فوکوس را به کنترل قبلی انتقال می دهند. این کار را Dialog Box Manager ویندوز بر عهده می گیرد. ولی اگر کنترل های خود را بر روی یک پنجره ساده قرار دهید ، مجبور خواهید بود خودتان این کار را با Subclass کردن آنها انجام دهید. در این مثال نیازی نیست که تک تک کنترل ها را Subclass کنیم زیرا قبلا تمام آنها را Superclass کرده ایم و می توانیم یک کنترل مرکزی روی آنها داشته باشیم.

```
.elseif al==VK_TAB
    invoke GetKeyState,VK_SHIFT
    test eax,80000000
    .if ZERO?
        invoke GetWindow,hEdit,GW_HWNDNEXT
        .if eax==NULL
            invoke GetWindow,hEdit,GW_HWNDFIRST
        .endif
    .else
        invoke GetWindow,hEdit,GW_HWNDPREV
        .if eax==NULL
            invoke GetWindow,hEdit,GW_HWNDLAST
        .endif
    .endif
    invoke SetFocus,eax
    xor eax,eax
    ret
```

کد بالا قسمتی از پروسیجر EditWndClass است. این قسمت فشرده شدن کلید Tab را بررسی کرده و در صورت مثبت بودن تابع GetKeyState را برای گرفتن وضعیت کلید Shift فراخوانی می کند. مقدار برگشتی این تابع مشخص می کند که آیا کلید مورد نظر فشار داده شده است یا نه. در صورت مثبت بودن High bit از eax را ست می کند. پس مقدار بازگشتی را با 80000000h چک می کنیم. تساوی این دو مقدار نشانگر فشرده شدن کلیدهای Shift+Tab می باشد.

اگر کاربر تنها کلید Tab را زده باشد ، تابع GetWindow را برای بدست آوردن شماره دسترسی کنترل بعدی فراخوانی می کنیم. از فلگ GW_HWNDNEXT برای دریافت شماره دسترسی به کنترل بعدی استفاده می کنیم. در صورتیکه این تابع مقدار Null را بازگرداند ، یعنی این کنترل آخرین کنترل است پس با استفاده از تابع GetWindow و فلگ GW_HWNDFIRST به کنترل اول باز می گردیم. در مورد Shift+Tab برعکس این موارد را انجام می دهیم.

Bitmap

کدها و فایلهای مربوط به این قسمت در CD ضمیمه موجود می‌باشد

SourceCodes\Asm32\Chapter20

بیت مپ یا نقشه بیتی روشی برای ذخیره تصاویر بر روی کامپیوتر است.

فرمتها و روش های زیادی برای این کار وجود دارند ولی ویندوز به طور پیش فرض فقط از فرمت نقشه بیتی (Bitmap) پشتیبانی می کند. ساده ترین راه برای استفاده از بیت مپ ها ، فایلهای منبع هستند. برای انجام این کار دو روش وجود دارد.

- می توانید بیت مپ را در فایل منبع (.rc) معرفی کنید به عنوان مثال:

```
#define IDB_MYBITMAP   100
IDB_MYBITMAP  BITMAP  "c:\project\example.bmp"
```

در این روش از یک ثابت به عنوان نماینده ای برای بیت مپ استفاده می شود. خط اول ثابتی به نام IDB_MYBITMAP را تعریف می کند که دارای مقدار 100 است. از این ثابت برای رجوع به بیت مپ در برنامه استفاده می کنیم. خط بعدی منبع بیت مپ را معرفی می کند و به کامپایلر می گوید که فایل bmp واقعی را از کجا پیدا کند.

- می توانید از یک نام به عنوان نماینده ای برای بیت مپ استفاده کنید.

```
MyBitMap  BITMAP "c:\project\example.bmp"
```

در این روش برای ارجاع به بیت مپ تنها از رشته MyBitmap استفاده می کنیم.

هر دو روش مذکور به خوبی کار می کنند و انتخاب اینکه از کدامیک استفاده کنید به عهده خود شما می باشد. حال که بیت مپ را در فایل منبع معرفی کردیم ، باید آنرا در منطقه کاری فرم خود نمایش دهیم. برای این کار مراحل زیر را دنبال می کنیم.

۱- تابع LoadBitmap را برای گرفتن شماره دسترسی به بیت مپ فراخوانی می کنیم. در زیر ، ساختار این تابع را مشاهده می کنید.

```
LoadBitmap proto hInstance:HINSTANCE, lpBitmapName:LPSTR
```

این تابع شماره دسترسی به بیت مپ مورد نظر را بر می گرداند. hInstance شماره دسترسی برنامه شما را مشخص می کند. lpBitmapName اشاره گری به رشته نام بیت مپ است که قبلا آنرا معرفی کرده اید (روش دوم معرفی). اگر از یک ثابت برای ارجاع به بیت مپ استفاده کرده اید (مانند IDB_MYBITMAP) ، می توانید از مقدار آن برای ارجاع استفاده کنید.

مثال کوتاه بعد مراحل کار را به سادگی نشان می دهد.

```
First Method:
.386
.model flat, stdcall
.................
.const
IDB_MYBITMAP    equ 100
.................
.data?
hInstance  dd ?
...............
.code
..............
    invoke GetModuleHandle,NULL
    mov hInstance,eax
.............
    invoke LoadBitmap,hInstance,IDB_MYBITMAP
............
Second Method:
.386
.model flat, stdcall
.................
.data
BitmapName  db "MyBitMap",0
.................
.data?
hInstance  dd ?
...............
.code
..............
    invoke GetModuleHandle,NULL
```

```
    mov hInstance,eax
.............
    invoke LoadBitmap,hInstance,addr BitmapName
............
```

۲- شماره دسترسی به ابزار (DC) را با استفاده از تابع GetDC بدست می آوریم.

۳- یک حافظه DC با مشخصات یکسان با DC مرحله قبل ایجاد می کنیم. ایده این بخش اینست که ابتدا یک محل مخفی برای کشیدن بیت مپ ایجاد کرده و سپس محتویات آنرا به DC اصلی که صفحه نمایش یا پرینتر است ، کپی کنیم. این محل مخفی را می توانید توسط تابع CreateCompatibleDC ایجاد کنید.

```
CreateCompatibleDC  proto  hdc:HDC
```

در صورت موفقیت ، تابع شماره دسترسی به حافظه DC را بر می گرداند. این حافظه با ابزار خروجی شما سازگار است. به این روش نمایش تصاویر ، Double_Buffering گفته می شود که از سرعت بالایی نیز برخوردار است.

٤- حال که صفحه مخفی را ایجاد کرده اید ، می توانید بیت مپ را انتخاب کرده و روی آن بکشید. این کار توسط تابع SelectObject صورت می گیرد که ساختار آنرا در زیر مشاهده می کنید.

```
SelectObject   proto  hdc:HDC, hGdiObject:DWORD
```

این تابع دو پارامتر ورودی می گیرد که یکی شماره دسترسی به حافظه DC و دیگری شماره دسترسی به بیت مپ است.

۵- در این مرحله تصویر بر روی حافظه مخفی DC کشیده شده است و تنها کاری که ما باید انجام دهیم اینست که تصویر را بر روی وسیله خروجی که صفحه نمایش یا پرینتر است نیز کپی کنیم. توابع زیادی از قبیل StretchBlt و BitBlt برای این کار وجود دارند.

BitBlt تمام محتویات DC مبدا را بدون تغییر به DC مقصد کپی می کند. به همین دلیل از سرعت بالایی نیز برخوردار است.

StretchBlt می تواند تصویر را کشیده تر یا فشرده تر کند تا با ابعاد منطقه خروجی مطابقت داشته باشد.

در این مثال برای سادگی از BitBlt استفاده می کنیم. در زیر ، ساختار این تابع را مشاهده می کنید.

```
BitBlt  proto  hdcDest  :DWORD,\
               nxDest   :DWORD,\
               nyDest   :DWORD,\
               nWidth   :DWORD,\
               nHeight  :DWORD,\
               hdcSrc   :DWORD,\
               nxSrc    :DWORD,\
               nySrc    :DWORD,\
               dwROP    :DWORD
```

hdcDest شماره دسترسی به DC مقصد را مشخص می کند.

nxDest و nyDest مختصات گوشه سمت چپ بالای منطقه خروجی را تعیین می کند.

nWidth و nHeight عرض و ارتفاع منطقه خروجی را مشخص می کند.

hdcSrc شماره دسترسی به DC منبع را مشخص می کند که تصویر درون آن قرا دارد.

nxSrc و nySrc مختصات گوشه سمت چپ بالای منبع تصویر را مشخص می کنند.

dwROP روش ترکیب رنگهای منبع با مقصد را مشخص می کند. به عنوان مثال می توان تصویر مبدا را با مقصد ترکیب (AND) کرد. ولی در اکثر مواقع می خواهیم که عملیات کپی بدون در نظر گرفتن رنگ مقصد صورت بگیرد.

٦- وقتی کارتان با بیت مپ تمام شد ، می توانید با استفاده از تابع DeleteObject آنرا از حافظه خارج کنید.

حال به سراغ مثال این بخش می رویم.

```
.386
.model flat,stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
include \masm32\include\gdi32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\gdi32.lib
WinMain proto :DWORD,:DWORD,:DWORD,:DWORD
IDB_MAIN   equ 1
```

```
.data
ClassName db "SimpleWin32ASMBitmapClass",0
AppName   db "Win32ASM Simple Bitmap Example",0
.data?
hInstance   HINSTANCE ?
CommandLine LPSTR ?
hBitmap     dd ?
.code
start:
 invoke GetModuleHandle, NULL
 mov    hInstance,eax
 invoke GetCommandLine
 mov    CommandLine,eax
 invoke WinMain, hInstance,NULL,CommandLine, SW_SHOWDEFAULT
 invoke ExitProcess,eax
WinMain proc
hInst:HINSTANCE,hPrevInst:HINSTANCE,CmdLine:LPSTR,CmdShow:DWORD
 LOCAL wc:WNDCLASSEX
 LOCAL msg:MSG
 LOCAL hwnd:HWND
 mov   wc.cbSize,SIZEOF WNDCLASSEX
 mov   wc.style, CS_HREDRAW or CS_VREDRAW
 mov   wc.lpfnWndProc, OFFSET WndProc
 mov   wc.cbClsExtra,NULL
 mov   wc.cbWndExtra,NULL
 push  hInstance
 pop   wc.hInstance
 mov   wc.hbrBackground,COLOR_WINDOW+1
 mov   wc.lpszMenuName,NULL
 mov   wc.lpszClassName,OFFSET ClassName
 invoke LoadIcon,NULL,IDI_APPLICATION
 mov   wc.hIcon,eax
 mov   wc.hIconSm,eax
 invoke LoadCursor,NULL,IDC_ARROW
 mov   wc.hCursor,eax
 invoke RegisterClassEx, addr wc
 INVOKE CreateWindowEx,NULL,\
                       ADDR ClassName,\
                       ADDR AppName,\
                       WS_OVERLAPPEDWINDOW,\
                       CW_USEDEFAULT,\
                       CW_USEDEFAULT,\
                       CW_USEDEFAULT,\
                       CW_USEDEFAULT,\
                       NULL,\
                       NULL,\
                       hInst,\
                       NULL
 mov   hwnd,eax
 invoke ShowWindow, hwnd,SW_SHOWNORMAL
 invoke UpdateWindow, hwnd
 .while TRUE
```

```
  invoke GetMessage, ADDR msg,NULL,0,0
  .break .if (!eax)
  invoke TranslateMessage, ADDR msg
  invoke DispatchMessage, ADDR msg
 .endw
 mov     eax,msg.wParam
 ret
WinMain endp
WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
   LOCAL ps:PAINTSTRUCT
   LOCAL hdc:HDC
   LOCAL hMemDC:HDC
   LOCAL rect:RECT
   .if uMsg==WM_CREATE
      invoke LoadBitmap,hInstance,IDB_MAIN
      mov hBitmap,eax
   .elseif uMsg==WM_PAINT
      invoke BeginPaint,hWnd,addr ps
      mov    hdc,eax
      invoke CreateCompatibleDC,hdc
      mov    hMemDC,eax
      invoke SelectObject,hMemDC,hBitmap
      invoke GetClientRect,hWnd,addr rect
      invoke BitBlt,hdc,\
                    0,\
                    0,\
                    rect.right,\
                    rect.bottom,\
                    hMemDC,\
                    0,\
                    0,\
                    SRCCOPY

      invoke DeleteDC,hMemDC
      invoke EndPaint,hWnd,addr ps
 .elseif uMsg==WM_DESTROY
  invoke DeleteObject,hBitmap
  invoke PostQuitMessage,NULL
 .ELSE
  invoke DefWindowProc,hWnd,uMsg,wParam,lParam
  ret
 .ENDIF
 xor eax,eax
 ret
WndProc endp
end start
```

```
;-----------------------------------------------------------------
;                              The resource script
;-----------------------------------------------------------------
#define IDB_MAIN 1
IDB_MAIN BITMAP "tweety78.bmp"
```

حال به تجزیه و تحلیل کدهای این بخش می پردازیم.

```
.if uMsg==WM_CREATE
   invoke LoadBitmap,hInstance,IDB_MAIN
   mov hBitmap,eax
```

در پاسخ به پیغام WM_CREATE تابع LoadBitmap را برای بارگذاری تصویر از منبع فراخوانی کرده و پارامتر های مورد نیاز را به آن منتقل می کنیم. در صورت موفقیت ، تابع شماره دسترسی به بیت مپ را به عنوان خروجی در eax قرار خواهد داد.

حال که بیت مپ بارگذاری شده است می توانیم آنرا روی منطقه کاری فرم خود بکشیم.

```
.elseif uMsg==WM_PAINT
   invoke BeginPaint,hWnd,addr ps
   mov    hdc,eax
   invoke CreateCompatibleDC,hdc
   mov    hMemDC,eax
   invoke SelectObject,hMemDC,hBitmap
   invoke GetClientRect,hWnd,addr rect
   invoke BitBlt,hdc,\
               0,\
               0,\
               rect.right,\
               rect.bottom,\
               hMemDC,\
               0,\
               0,\
               SRCCOPY

   invoke DeleteDC,hMemDC
   invoke EndPaint,hWnd,addr ps
```

باید در پاسخ به پیغام WM_PAINT بیت مپ را دوباره بکشیم. ابتدا تابع BeginPaint را برای گرفتن DC ابزار مقصد فراخوانی می کنیم. برای گرفتن ابعاد منطقه کاری فرم ، از تابع GetClientRect استفاده می کنیم که ابعاد فرم را در یک ساختار RECT کپی می کند. در این

ساختار Right مشخص کننده عرض و Bottom مشخص کننده ارتفاع منطقه کاری است. سپس با فراخوانی تابع BitBlt محتویات DC مبدا را به مقصد که منطقه کاری پنجره است ، کپی می کنیم. حال که چاپ تصویر به پایان رسید ، دیگر نیازی به DC جانبی که به عنوان بافر از حافظه گرفته بودیم نداریم پس با استفاده از تابع EndPaint پایان کار کشیدن تصویر را اعلام می کنیم. در آخر هم چون دیگر نیازی به بیت مپ نداریم آنرا با استفاده از تابع DeleteObject از حافظه خارج می‌کنیم.

Win32 Debug API (بخش ۱)

کدها و فایلهای مربوط به این قسمت در CD ضمیمه موجود می‌باشد

SourceCodes\Asm32\Chapter21

در این بخش اصول اولیه طراحی یک Debugger را مورد بررسی قرار خواهیم داد.

دیباگ به معنای رفع مشکلات اجرایی برنامه است که معمولاً با اجرای خط به خط برنامه و بررسی وضعیت متغیرها و ثباتها صورت می گیرد. به برنامه ای که امکان این کار را به ما می دهد Debugger گفته می شود. معمولاً این برنامه ها توانایی ایجاد تغییرات را نیز دارا می باشند.

Win32 دارای چندین تابع API می باشد که توانایی debug کردن را به برنامه نویسان می دهند. این توابع Win32 Debug API نامیده می شوند که توسط آنها می توانید:

- برنامه ای را برای دیباگ کردن بارگذاری کرده و یا به یک برنامه در حال اجرا متصل شوید.
- اطلاعات بسیار دقیقی در رابطه با برنامه ای که دیباگ می کنید بدست آورید.
- از رویداد هایی از قبیل شروع و خاتمه صف دستورات ، بارگذاری dll ها و ... باخبر شوید.
- تغییراتی را در Process و یا صف دستورات خود ایجاد کنید.

در ادامه مراحل کار با Win32 Debug API را بررسی خواهیم کرد.

۱- ایجاد یک Process و یا اتصال به یک Process در حال اجرا ، اولین قدم در استفاده از Win32 Debug API محسوب می شود. Process و یا برنامه ای که دیباگ می شود ، Debuggee نامیده می شود. به دو روش می توانید یک Debuggee ایجاد کنید.

- می توانید با استفاده از تابع CreateProcess یک Process جدید برای دیباگ ایجاد کنید. به منظور ایجاد این Process باید از فلگ DEBUG_PROCESS استفاده کنید. این فلگ به ویندوز اعلام می کند که Process جدید تنها به منظور دیباگ کردن ایجاد شده است. ویندوز نیز اطلاعات مهم در مورد رویدادهای Debuggee را برای برنامه شما می فرستد. Debuggee پس از ایجاد به سرعت متوقف می شود. اگر Debuggee حاوی Process های فرزند باشد ، ویندوز رویدادهای دیباگ مربوط به آنها را نیز به برنامه شما می‌فرستد. در صورت عدم نیاز می توانید با ترکیب دو فلگ DEBUG_PROCESS و

DEBUG_ONLY_THIS_PROCESS در تابع CreateProcess ، از فرستاده شدن پیغام های مربوط به Process های فرزند توسط ویندوز جلوگیری کنید.

- می توانید با استفاده از تابع DebugActiveProcess به یک Process در حال اجرا متصل شوید.

۲- انتظار برای رویدادهای دیباگ. پس از ایجاد Debuggee صف دستورات اصلی آن متوقف شده و به این توقف تا زمانی که برنامه شما تابع WaitForDebugEvent را فراخوانی کند ، ادامه می دهد. این تابع همانند دیگر توابع WaitForXXX عمل می کند و صف دستوراتی که آنرا فراخوانی کرده است را تا زمان رویداد مشخص متوقف می کند. در این حالت این تابع منتظر رویدادهای دیباگ فرستاده شده توسط ویندوز می ماند. در زیر ساختار این تابع را مشاهده می کنید.

```
WaitForDebugEvent proto lpDebugEvent:DWORD, dwMilliseconds:DWORD
```

lpDebugEvent اشاره گری به رکورد DEBUG_EVENT است که با اطلاعاتی در مورد دیباگ که در Debuggee رخ می دهد پر می شود.

dwMilliseconds حداکثر زمانی است که این تابع منتظر رویدادهای دیباگ می ماند. اگر این زمان به پایان برسد و هیچ رویدادی رخ ندهد ، تابع بازگشت خواهد کرد.

حال به بررسی جزئیات رکورد DEBUG_EVENT می پردازیم.

```
DEBUG_EVENT STRUCT
   dwDebugEventCode dd ?
   dwProcessId      dd ?
   dwThreadId       dd ?
   u DEBUGSTRUCT    <>
DEBUG_EVENT ENDS
```

dwDebugEventCode نوع رویداد را مشخص می کند. برنامه ما باید این مقدار را بررسی کند تا بتواند تشخیص دهد که چه نوع رویدادی اتفاق افتاده است و چه پاسخی باید به آن بدهد. مقادیر ممکن عبارتند از:

- CREATE_PROCESS_DEBUG_EVENT مشخص کننده ایجاد Process است. این رویداد در زمان ایجاد Process و یا هنگامی که برنامه به یک Process در حال اجرا متصل

می‌شود ، ایجاد می گردد. در حقیقت این رویداد اولین رویدادی است که برنامه شما دریافت خواهد.

- EXIT_PROCESS_DEBUG_EVENT اتمام کار Process را اعلام می کند.
- EXIT_THREAD_DEBUG_EVENT مشخص می کند که یک صف دستورات در Debuggee خاتمه پیدا کرده است. برنامه شما برای صف دستورات اصلی این رویداد را دریافت نخواهد کرد.
- CREATE_THREAD_DEBUG_EVENT مشخص می کند که یک صف دستورات جدید در Debuggee ایجاد شده و یا برنامه شما به یک Process در حال اجرا متصل شده است. هنگامی که صف دستورات اصلی در Debuggee ایجاد می شود ، این رویداد را دریافت نخواهید کرد.
- LOAD_DLL_DEBUG_EVENT مشخص می کند که Debuggee فایل dll ای را بارگذاری کرده است. این رویداد را هنگامی دریافت می کنید که بارگذار فایلهای اجرایی برای اولین بار با dll های برنامه ارتباط برقرار کرده و یا Debuggee تابع LoadLibrary را فراخوانی می کند.
- UNLOAD_DLL_DEBUG_EVENT مشخص می کند که یک استثناء یا حالت خاص در Debuggee رخ داده است. این استثناء ممکن است خطایی در روند اجرایی برنامه و یا وقفه دیباگ
(int 3h) باشد. این وقفه درست هنگامی اتفاق می افتد که Debuggee اولین دستور خود را اجرا می کند. در صورتی که می خواهید به Debuggee بازگردید می توانید تابع ContinueDebugEvent را با فلگ DBG_CONTINUE فراخوانی کنید. برای این کار از فلگ DBG_EXCEPTION_NOT_HANDLED استفاده نکنید چون در این صورت برنامه Debuggee در ویندوز NT به درستی اجرا نخواهد شد. ولی در این وضعیت Win9X تفاوتی بین این دو فلگ قائل نیست و Debuggee در این سیستم عامل به درستی کار خواهد کرد.
- OUTPUT_DEBUG_STRING_EVENT مشخص می کند که Debuggee تابع DebugOutputString را برای فرستادن یک رشته پیغام به برنامه شما فراخوانی کرده است.
- RIP_EVENT مشخص کننده بروز یک اشکال سیستمی در روند دیباگ کردن است.

dwThreadId و dwProcessId همان شماره شناسه Process و صف دستورات هستند که رویدادهای دیباگ در آن اتفاق افتاده است. یادآوری می کنیم که اگر از تابع CreateProcess برای بارگذاری Debuggee استفاده کنید ، می توانید شماره شناسه Process و صف دستورات را از رکورد PROCESS_INFO بدست آورید. در صورت عدم استفاده از فلگ DEBUG_ONLY_THIS_PROCESS می توانید از این مقادیر برای تمایز بین رویدادهای Debuggee و Process های فرزند استفاده کنید.

U حاوی اطلاعات بیشتری در مورد رویدادهای دیباگ می باشد که می تواند بسته به مقدار dwDebugEventCode یکی از رکوردهای زیر باشد.

Value in dwDebugEventCode	Interpretation of u
CREATE_PROCESS_DEBUG_EVENT	A CREATE_PROCESS_DEBUG_INFO structure named CreateProcessInfo
EXIT_PROCESS_DEBUG_EVENT	An EXIT_PROCESS_DEBUG_INFO structure named ExitProcess
CREATE_THREAD_DEBUG_EVENT	A CREATE_THREAD_DEBUG_INFO structure named CreateThread
EXIT_THREAD_DEBUG_EVENT	An EXIT_THREAD_DEBUG_EVENT structure named ExitThread
LOAD_DLL_DEBUG_EVENT	A LOAD_DLL_DEBUG_INFO structure named LoadDll
UNLOAD_DLL_DEBUG_EVENT	An UNLOAD_DLL_DEBUG_INFO structure named UnloadDll
EXCEPTION_DEBUG_EVENT	An EXCEPTION_DEBUG_INFO structure named Exception
OUTPUT_DEBUG_STRING_EVENT	An OUTPUT_DEBUG_STRING_INFO structure named DebugString
RIP_EVENT	A RIP_INFO structure named RipInfo

در این بخش فقط رکورد CREATE-PROCESS_DEBUG_INFO را مورد بررسی قرار خواهیم داد. اولین کاری که پس از بازگشت تابع WaitForDebugEvent باید انجام دهیم ، بررسی مقدار dwDebugEventCode به منظور تعیین نوع رویداد است. به عنوان مثال اگر مقدار این عضو برابر CREATE_PROCESS_DEBUG_EVENT باشد ، می توانید عضو درونی u را رکوردی از نوع CREATE_PROCESS_DEBUG_INFO و با نام CreateProcessInfo در نظر گرفته و بصورت زیر به آن دسترسی داشته باشید.

```
U.CreateProcessInfo.<member name>
```

۳- عملیات مورد نظر را در پاسخ به رویدادهای دیباگ انجام دهید. زمانیکه که یک رویداد دیباگ در Debuggee رخ داده و یا زمان انتظار به پایان برسد ، تابع WaitForDebugEvent باز می گردد. حال برنامه باید مقدار موجود در dwDebugEventCode را برای عکس العملی مناسب مورد تجزیه قرار دهد. این کار دقیقا همانند پاسخ گویی به رویدادها پنجره است.

٤- به Debuggee اجازه دهید که روند اجرایی خود را ادامه دهد. زمانی که یک رویداد دیباگ رخ دهد ، ویندوز Debuggee را متوقف می کند. وقتی که کار شما با رویداد مورد نظر تمام شد ، می توانید با استفاده از تابع ContinueDebugEvent , Debuggee را برای ادامه کار خود تحریک کنید.

در زیر ، ساختار این تابع را مشاهده می کنید.

```
ContinueDebugEvent proto dwProcessId      :DWORD,\
                         dwThreadId       :DWORD,\
                         dwContinueStatus :DWORD
```

dwThreadId و **dwProcessId** شماره شناسایی Process و صف دستورات را مشخص می کنند. این دو مقدار را می توانید از رکورد DEBUG_EVENT بدست آورید.

dwContinueStatus چگونگی ادامه صف دستورات را مشخص می کند که می تواند یکی از دو مقدار DBG_EXCEPTION_NOT_HANDLE یا DBG_CONTINUE را داشته باشد. به جز رویداد EXCEPTION_DEBUG_EVENT در مورد سایر رویدادها این دو فلگ کار یکسانی را انجام می دهند که همان ادامه کار صف دستورات است. در زمان روی دادن یک استثناء ، در صورتی که از فلگ DBG_CONNTINUE استفاده کنید ، صف دستورات آنرا را نادیده گرفته و به کار خود ادامه می دهد. در اینصورت برنامه شما باید قبل از اینکه صف دستورات را با استفاده از این فلگ بازگرداند ، استثناء را مورد بررسی قرار داده و رفع کرده باشد ، در غیر اینصورت این

استثناء باز هم تکرار خواهد شد. با استفاده از فلگ DBG_EXCEPTION_NOT_HANDLED برنامه شما به ویندوز اعلام می کند که مدیریت این استثناء را بر عهده نخواهد گرفت و خود ویندوز باید این کار را انجام داده و به صورت پیش فرض به آن پاسخ دهد. بنابراین فقط در صورتی از فلگ DBG_CONTINUE استفاده کنید که برنامه شما علت بروز این استثناء را از بین برده باشد. تنها در یک حالت همیشه باید از فلگ DBG_CONTINUE استفاده کنید و آن هم اولین EXCEPTION_DEBUG_EVENT است که حاوی مقدار EXCEPTION_BREAKPOINT در عضو ExceptionCode می باشد.

همان‌طور که گفته شد زمانیکه Debuggee قصد اجرای اولین دستور خود را دارد برنامه شما یک رویداد استثناء را دریافت خواهد کرد که در واقع یک نقطه توقف (int 3h) است . در این حالت اگر شما تابع ContinueDebugEvent را با فلگ DBG_EXCEPTION_NOT_HANDLED فراخوانی کنید ، ویندوز NT از اجرای Debuggee سر باز خواهد زد . پس در این حالت باید همیشه از فلگ DBG_CONTINUE برای ادامه کار صف دستورات استفاده کنید .

۵- این چرخه را تا خاتمه Debuggee ادامه دهید . برنامه Debuggee باید همانند حلقه پیغام در پروسیجر پنجره ، وضعیت خاتمه Debuggee را در یک حلقه بی نهایت بررسی کند .

در زیر ساختار کلی این حلقه را مشاهده می کنید .

```
.while TRUE
    invoke WaitForDebugEvent, addr DebugEvent, INFINITE
   .break .if DebugEvent.dwDebugEventCode==EXIT_PROCESS_DEBUG_EVENT
   <Handle the debug events>
   invoke ContinueDebugEvent, DebugEvent.dwProcessId,\
                              DebugEvent.dwThreadId,\
                              DBG_EXCEPTION_NOT_HANDLED
.endw
```

باید توجه داشته باشید که وقتی دیباگ کردن یک برنامه را شروع می کنید ، تا زمانیکه این برنامه خاتمه نیافته است نمی توانید ازآن جدا شوید . حال مراحل کار را بصورت خلاصه بیان می کنیم .

- ایجاد Process و یا اتصال به یک Process در حال اجرا .
- انتظار برای رویدادهای دیباگ
- پاسخ گویی به رویدادها
- ادامه اجرای صف دستورات در Debuggee

- ادامه این چرخه تا زمانیکه Debuggee Process خاتمه یابد .

حال به سراغ مثال این بخش می رویم .

این مثال یک برنامه Win32 را دیباگ کرده و اطلاعات مهمی نظیر شماره دسترسی و شناسه Process و .... را نشان می دهد .

```
.386
.model flat,stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
include \masm32\include\comdlg32.inc
include \masm32\include\user32.inc
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\comdlg32.lib
includelib \masm32\lib\user32.lib
.data
AppName db "Win32 Debug Example no.1",0
ofn OPENFILENAME <>
FilterString db "Executable Files",0,"*.exe",0
             db "All Files",0,"*.*",0,0
ExitProc  db "The Debuggee exits",0
NewThread db "A new thread is created",0
EndThread db "A thread is destroyed",0
ProcessInfo db "File Handle: %lx ",0dh,0Ah
            db "Process Handle: %lx",0Dh,0Ah
            db "Thread Handle: %lx",0Dh,0Ah
            db "Image Base: %lx",0Dh,0Ah
            db "Start Address: %lx",0
.data?
buffer db 512 dup(?)
startinfo STARTUPINFO <>
pi PROCESS_INFORMATION <>
DBEvent DEBUG_EVENT <>
.code
start:
mov ofn.lStructSize,sizeof ofn
mov ofn.lpstrFilter, offset FilterString
mov ofn.lpstrFile, offset buffer
mov ofn.nMaxFile,512
mov ofn.Flags, OFN_FILEMUSTEXIST or \
OFN_PATHMUSTEXIST or \
OFN_LONGNAMES or \
OFN_EXPLORER or \
OFN_HIDEREADONLY
invoke GetOpenFileName, ADDR ofn
.if eax==TRUE
```

```
invoke GetStartupInfo,addr startinfo
invoke CreateProcess, addr buffer,\
                      NULL,\
                      NULL,\
                      NULL,\
                      FALSE,\
                      DEBUG_PROCESS + DEBUG_ONLY_THIS_PROCESS,\
                      NULL,\
                      NULL,\
                      addr startinfo,\
                      addr pi
.while TRUE
   invoke WaitForDebugEvent, addr DBEvent, INFINITE
   .if DBEvent.dwDebugEventCode==EXIT_PROCESS_DEBUG_EVENT
       invoke MessageBox,0,\
                        addr ExitProc,\
                        addr AppName,\
                        MB_OK+MB_ICONINFORMATION
       .break
   .elseif DBEvent.dwDebugEventCode==CREATE_PROCESS_DEBUG_EVENT
       invoke wsprintf, addr buffer,\
                        addr ProcessInfo,\
                        DBEvent.u.CreateProcessInfo.hFile,\
                        DBEvent.u.CreateProcessInfo.hProcess,\
                        DBEvent.u.CreateProcessInfo.hThread,\
                        DBEvent.u.CreateProcessInfo.lpBaseOfImage,\
                        DBEvent.u.CreateProcessInfo.lpStartAddress
       invoke MessageBox,0,\
                        addr buffer,\
                        addr AppName,\
                        MB_OK+MB_ICONINFORMATION
   .elseif DBEvent.dwDebugEventCode==EXCEPTION_DEBUG_EVENT
       .if DBEvent.u.Exception.pExceptionRecord.ExceptionCode ==
EXCEPTION_BREAKPOINT
         invoke ContinueDebugEvent, DBEvent.dwProcessId,\
                                    DBEvent.dwThreadId,\
                                    DBG_CONTINUE
        .continue
       .endif
   .elseif DBEvent.dwDebugEventCode == CREATE_THREAD_DEBUG_EVENT
       invoke MessageBox,0,\
                        addr NewThread,\
                        addr AppName,\
                        MB_OK+MB_ICONINFORMATION
   .elseif DBEvent.dwDebugEventCode==EXIT_THREAD_DEBUG_EVENT
       invoke MessageBox,0,\
                        addr EndThread,\
                        addr AppName,\
                        MB_OK+MB_ICONINFORMATION
   .endif
   invoke ContinueDebugEvent, DBEvent.dwProcessId,\
                              DBEvent.dwThreadId,\
```

```
                                DBG_EXCEPTION_NOT_HANDLED
.endw
invoke CloseHandle,pi.hProcess
invoke CloseHandle,pi.hThread
.endif
invoke ExitProcess, 0
end start
```

در ابتدا اعضای رکورد OPENFILENAME را مقداردهی کرده و تابع GetOpenFileName را فراخوانی می کنیم تا به کاربر اجازه دهیم که برنامه مورد نظر خود را انتخاب کند . پس از اینکه کاربر برنامه ای را برای دیباگ شدن انتخاب کرد ، تابع GetStartupInfo را برای مقدار دهی اعضای رکورد STARTUPINFO با مقادیر آنها در برنامه خودمان ، فراخوانی می کنیم .

```
invoke GetStartupInfo,addr startinfo
invoke CreateProcess, addr buffer,\
                      NULL,\
                      NULL,\
                      NULL,\
                      FALSE,\
                      DEBUG_PROCESS + DEBUG_ONLY_THIS_PROCESS,\
                      NULL,\
                      NULL,\
                      addr startinfo,\
                      addr pi
.while TRUE
   invoke WaitForDebugEvent, addr DBEvent, INFINITE
```

وقتی که Debuggee بارگذاری شد، وارد حلقه پیغام‌های دیباگ شده و تابع WaitForDebugEvent را فراخوانی می کنیم . این تابع تا زمانیکه رویدادی رخ ندهد بازگشت نمیکند چون مقدار INFINITY را برای پارامتر دوم آن در نظر گرفته ایم . هنگامیکه یک رویداد دیباگ اتفاق بیفتد ، تابع WaitForDebugEvent باز می گردد و رکورد DBEvent با اطلاعاتی در مورد رویداد مذکور پر می شود .

```
   .if DBEvent.dwDebugEventCode==EXIT_PROCESS_DEBUG_EVENT
       invoke MessageBox, 0, \
                          addr ExitProc,\
                          addr AppName,\
                          MB_OK+MB_ICONINFORMATION
       .break
```

در این قسمت عضو dwDebugEventCode را مورد بررسی قرار میدهیم . در صورتیکه مقدار آن برابر EXIT_PROCESS _DEBUG_EVENT باشد ، توسط یک Messagebox خاتمه Debuggee را اعلام کرده و از حلقه پیغامها خارج می شویم .

```
   .elseif DBEvent.dwDebugEventCode==CREATE_PROCESS_DEBUG_EVENT
       invoke wsprintf, addr buffer,\
                        addr ProcessInfo,\
                        DBEvent.u.CreateProcessInfo.hFile,\
                        DBEvent.u.CreateProcessInfo.hProcess,\
                        DBEvent.u.CreateProcessInfo.hThread,\
                        DBEvent.u.CreateProcessInfo.lpBaseOfImage,\
                        DBEvent.u.CreateProcessInfo.lpStartAddress
       invoke MessageBox,0,\
                         addr buffer,\
                         addr AppName,\
                         MB_OK+MB_ICONINFORMATION
```

درصورتیکه مقدار dwDebugEventCode برابر CREATE_PROCESS_DEBUG_EVENT باشد، توسط یک MessageBox اطلاعاتی را در مورد Debuggee نمایش خواهیم داد . همانطور که گفته شد ، این اطلاعات را از رکورد u.CreateProcessInfo بدست خواهیم آورد . CreateProcessInfo رکوردی از نوع CREATE_PROCESS_DEBVG_INFO است . برای دریافت اطلاعات بیشتر در مورد این رکورد می توانید به Win32 API Reference مراجعه کنید .

```
   .elseif DBEvent.dwDebugEventCode==EXCEPTION_DEBUG_EVENT
       .if DBEvent.u.Exception.pExceptionRecord.ExceptionCode ==
EXCEPTION_BREAKPOINT
         invoke ContinueDebugEvent, DBEvent.dwProcessId,\
                                    DBEvent.dwThreadId,\
                                    DBG_CONTINUE
        .continue
       .endif
```

اگر مقدار dwDebugEventCode برابر EXCEPTION_DEBUG_EVENT باشد ، باید نوع استثناء را مورد بررسی قرار دهیم . نوع استثناء را می توانید بصورت بالا با بررسی ExceptionCode تعیین کنید . اگر مقدار این عضو برابر EXCEPTION_BREAKPOINT بوده و برای اولین بار رخ داده باشد, نتیجه می گیریم که این استثناء یک نقطه توقف است که در زمان اجرای اولین دستور Debuggee توسط ویندوز ایجاد شده است . اگر بدانید که برنامه Debuggee حاوی نقطه توقف (int 3h) نیست ، می توانید مطمئن باشید که این استثناء همان نقطه توقف اولیه

است و دیگر لزومی به بررسی زمان وقوع آن نیست . پس می توانید تابع ContinueDebugEvent را با فلگ DBG_CONTINUE فراخوانی کرده وبه Debggee اجازه دهید که روند اجرا یی خود را دنبال کند . پس از این کار باید منتظر رویداد بعدی بمانیم .

```
.elseif DBEvent.dwDebugEventCode == CREATE_THREAD_DEBUG_EVENT
    invoke MessageBox,0,\
                      addr NewThread,\
                      addr AppName,\
                      MB_OK+MB_ICONINFORMATION
.elseif DBEvent.dwDebugEventCode==EXIT_THREAD_DEBUG_EVENT
    invoke MessageBox,0,\
                      addr EndThread,\
                      addr AppName,\
                      MB_OK+MB_ICONINFORMATION
.endif
```

اگر مقدار dwDebugEventCode برابر CRATE_THREAD_DEBUG_EVENT یا EXIT_THREAD_DEBUG_EVENT باشد ، توسط MessageBox این رویداد ها را گزارش می دهیم.

```
   invoke ContinueDebugEvent, DBEvent.dwProcessId,\
                              DBEvent.dwThreadId,\
                              DBG_EXCEPTION_NOT_HANDLED
.endw
```

به جز حالت EXCEPTION_DEBUG_EVENT ، در بقیه حالتها برای بازگشت به Debuggee باید تابع ContinueDebugEvent را بافلگ DBG_EXCEPTION_NOT_HANDLED فراخوانی کنید .

```
invoke CloseHandle,pi.hProcess
invoke CloseHandle,pi.hThread
```

با خاتمه Debuggee ، شماره دسترسی Process و صف دستورات آنرا می بندیم.

Win32 Debug API(بخش۲)

کدها و فایل‌های مربوط به این قسمت در CD ضمیمه موجود می‌باشد

SourceCodes\Asm32\Chapter22

دراین بخش مبحث Win32 Debug API را ادامه داده و به بررسی چگونگی ایجاد تغییرات در Debuggee می پردازیم .

در فصل قبل یاد گرفتید که چگونه یک برنامه را برای دیباگ شدن بار گذاری کرده ورویدادهای دیباگ مربوط به آنرا مدیریت کنید . ولی اگر نتوانید تغییری در Debuggee ایجاد کنید ، این رویدادها نمی توانند برای شما مفید باشند . برای این منظور چندین دستور API وجود دارد که در این بخش مطالبی را در مورد آنها بیان خواهیم کرد.

ReadprocessMemory : این تابع به شما اجازه می دهد که حافظه اختصاص یافته به یک Process را بخوانید. ساختار این تابع بصورت زیر است .

```
ReadProcessMemory proto hProcess  :DWORD,\
lpBaseAddress                     :DWORD,\
lpBuffer                          :DWORD,\
nSize                             :DWORD,\
lpNumberOfBytesRead               :DWORD
```

hProcess شماره دسترسی Process مورد نظر را مشخص می کند .

lpBaseAddress آدرس شروع را مشخص می کند . به عنوان مثال اگر می خواهید ٤ بایت از حافظه Debuggee را از آدرس 401000h بخوانید ، باید مقدار پارامتر را 401000h قرا ر دهید .

lpBuffer آدرس بافری است که اطلاعات خوانده شده از Process در آن ذخیره می شود .

nSize تعداد بایتی است که قصد داریم بخوانیم .

lpNumberOfBytesRead تعداد واقعی بایتهای خوانده شده را مشخص می کند . در صورت عدم نیاز می توانید از مقدار Null استفاده کنید .

WriteProcessMemory نقطه مقابل تابع ReadProcessMemory است و به شما توانایی نوشتن بر روی حافظه Process مورد نظر را می دهد . پارامترهای این تابع دقیقا" همانند پارامترهای تابع ReadProcessMemory هستند .

دو تابع API بعدی نیاز به یک پیش زمینه ذهنی دارند .

سیستم عاملهایی مانند ویندوز ، امکان اجرای چند برنامه را در یک زمان دارند . در این حالت ویندوز به هر صف دستور یک بازه زمانی اختصاص می دهد . با اتمام این زمان ، ویندوز صف مذکور را متوقف کرده و به صف بعدی که بیشترین اولویت را دارد اجازه اجرا می دهد . قبل از انجام این کار ویندوز مقادیر موجود در رجیسترهای آن صف را ذخیره می کند ، تا در هنگام بازگشت بتواند وضعیت قبلی را دوباره ایجاد کند . مقادیر ذخیره شده در رجیسترها مجموعا" Context نامیده می شوند .

در برنامه ، هنگامی که یک رویداد دیباگ رخ می دهد ، ویندوز اجرای Debuggee را متوقف کرده و Context آنرا ذخیره می کند . تا زمانیکه اجرای Debuggee متوقف شده است ، می توانید مطمئن باشید که Context آن بدون تغییر باقی خواهد ماند . مقادیر این Context را می توانید توسط تابع GetThreadContext دریافت کرده و با استفاده از تابع SetThreadContext آنها را تغییر دهید . این دو تابع بسیار قوی هستند و با استفاده از آنها کنترل بسیار خوبی بر روی Debuggee خواهید داشت . شما می توانید این مقادیر ذخیره شده را قبل از اینکه Debuggee به اجرا باز گردد تغییر دهید . به این ترتیب مقادیر در ریجسترها نوشته خواهند شد و هر تغییری که در Context ایجاد کرده باشید اثر خود را نشان خواهد داد . به عنوان مثال می توانید با تغییر مقدار رجیستر EIP ، روند اجرایی را به هر نقطه دلخواه منتقل کنید . توجه داشته باشید که انجام اینگونه کارها در شرایط عادی امکان پذیر نیست . در زیر ، ساختار تابع GetThreadContext را مشاهده می کنید .

```
GetThreadContext proto hThread:DWORD, lpContext:DWORD
```

hThread شماره صف دستور مورد نظر را مشخص می کند که قصد دارید Context آنرا بدست آورید.

lpContext اشاره گری به رکورد Context است که با اطلاعاتی در مورد رجیسترها پر خواهد شد .

ساختار و پارامترهای تابع SetThreadContext نیز دقیقا" همانند GetThreadContext است . حال رکورد Context را مورد بررسی قرار می دهیم .

```
CONTEXT STRUCT
ContextFlags dd ?

iDr0 dd ?
iDr1 dd ?
iDr2 dd ?
iDr3 dd ?
iDr6 dd ?
iDr7 dd ?
;-----------------------------------------------------------------
; This section is returned if ContextFlags contains the value
CONTEXT_FLOATING_POINT
;-----------------------------------------------------------------
FloatSave FLOATING_SAVE_AREA <>
;-----------------------------------------------------------------
; This section is returned if ContextFlags contains the value
CONTEXT_SEGMENTS
;-----------------------------------------------------------------
regGs dd ?
regFs dd ?
regEs dd ?
regDs dd ?
;-----------------------------------------------------------------
; This section is returned if ContextFlags contains the value
CONTEXT_INTEGER
;-----------------------------------------------------------------
regEdi dd ?
regEsi dd ?
regEbx dd ?
regEdx dd ?
regEcx dd ?
regEax dd ?
;-----------------------------------------------------------------
; This section is returned if ContextFlags contains the value
CONTEXT_CONTROL
;-----------------------------------------------------------------
dd ?   egEbp
dd ?  regEip
dd ?   regCs
dd ?     regFlag
dd ?  regEsp
dd ?   regSs
;-----------------------------------------------------------------
; This section is returned if ContextFlags contains the value
CONTEXT_EXTENDED_REGISTERS
;-----------------------------------------------------------------
ExtendedRegisters db MAXIMUM_SUPPORTED_EXTENSION dup(?) CONTEXT ENDS
```

همانطور که مشاهده می کنید، اعضای این رکورد به تقلید از رجیسترهای واقعی CPU ایجاد شده اند . پیش از استفاده از این رکورد باید گروه مورد نظر را توسط عضو ContextFlags مشخص کنید . به عنوان مثال اگر بخواهید از همه رجیسترها استفاده کنید ، باید به این عضو مقدار CONTEXT_FULL بدهید و اگر قصد دارید که فقط از regFlags , regCs , regEip , regEbp , regEsp و یا regSs استفاده کنید ، باید مقدار CONTEXT_CONTROL را به کار ببرید .

نکته ای که باید یادآور شویم اینست که آدرس شروع رکورد Context در حافظه باید صفر بی از DWORD باشد . در غیر اینصورت در ویندوز NT اطلاعات غلطی را دریافت خواهید کرد . این کار را به بصورت زیر انجام می دهیم .

```
align dword
MyContext CONTEXT <>
```

حال به سراغ مثالهای این بخش می رویم .

مثال اول روش استفاده از تابع DebugActiveProcess را نشان خواهد داد . ابتدا برنامه هدف که نام آن Win.exe است را اجرا کنید . این برنامه قبل از اینکه پنجره خود را به نمایش بگذارد ، وارد یک حلقه بی نهایت می شود . در مرحله بعد مثال اول را اجرا کنید . این برنامه خود را به Process Win.exe چسبانده و قسمتی از کد آنرا تغییر می دهد . با این تغییر ، برنامه Win.exe از حلقه بی نهایت خارج شده و پنجره خود را نمایش می دهد .

```
.386
.model flat,stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
include \masm32\include\comdlg32.inc
include \masm32\include\user32.inc
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\comdlg32.lib
includelib \masm32\lib\user32.lib

.data
db "Win32 Debug Example no.2",0         AppName
db "SimpleWinClass",0   ClassName
db "Cannot find the target process",0  SearchFail
db "Target patched!",0   TargetPatched
dw 9090h     buffer

.data?
DEBUG_EVENT <>         DBEvent
dd ?   ProcessId
dd ?    ThreadId
dword       align
CONTEXT <>         context

.code
start:
invoke FindWindow, addr ClassName, NULL
.if eax!=NULL
    invoke GetWindowThreadProcessId, eax, addr ProcessId
    mov ThreadId, eax
    invoke DebugActiveProcess, ProcessId
    .while TRUE
       invoke WaitForDebugEvent, addr DBEvent, INFINITE
       .break .if DBEvent.dwDebugEventCode==EXIT_PROCESS_DEBUG_EVENT

       .if DBEvent.dwDebugEventCode==CREATE_PROCESS_DEBUG_EVENT
          mov context.ContextFlags, CONTEXT_CONTROL
          invoke GetThreadContext,\
                 DBEvent.u.CreateProcessInfo.hThread,\
                 addr context
          invoke WriteProcessMemory,
                   DBEvent.u.CreateProcessInfo.hProcess,\
                   context.regEip,\
                   addr buffer,\
                   2,\
                   NULL
          invoke MessageBox, 0,\
                             addr TargetPatched,\
                             addr AppName,\
                             MB_OK + MB_ICONINFORMATION
```

```
       .elseif DBEvent.dwDebugEventCode==EXCEPTION_DEBUG_EVENT
          .if DBEvent.u.Exception.pExceptionRecord.ExceptionCode\
==EXCEPTION_BREAKPOINT
             invoke ContinueDebugEvent, DBEvent.dwProcessId,\
                                        DBEvent.dwThreadId,\
                                        DBG_CONTINUE
             .continue
          .endif
       .endif
       invoke ContinueDebugEvent, DBEvent.dwProcessId,\
                                  DBEvent.dwThreadId,
                                  DBG_EXCEPTION_NOT_HANDLED
   .endw
.else
    invoke MessageBox, 0,\
                       addr SearchFail,\
                       addr AppName,\
                       MB_OK+MB_ICONERROR

.endif
invoke ExitProcess, 0
end start

mov wc.hIconSm,eax
invoke LoadCursor,NULL,IDC_ARROW
mov wc.hCursor,eax
invoke RegisterClassEx, addr wc
INVOKE CreateWindowEx,NULL,\
                      ADDR ClassName,\
                      ADDR AppName,\
                      WS_OVERLAPPEDWINDOW,CW_USEDEFAULT,\
                      CW_USEDEFAULT,\
                      CW_USEDEFAULT,\
                      CW_USEDEFAULT,\
                      NULL,\
                      NULL,\
                      hInst,\
                      NULL
mov hwnd,eax
jmp $
invoke ShowWindow, hwnd,SW_SHOWNORMAL
invoke UpdateWindow, hwnd
.while TRUE
   invoke GetMessage, ADDR msg,NULL,0,0
   .break .if (!eax)
   invoke TranslateMessage, ADDR msg
   invoke DispatchMessage, ADDR msg
.endw
mov eax,msg.wParam
ret
WinMain endp
```

برنامه ما باید با استفاده از تابع DebugActiveProcess خود را به یک Debuggee متصل کند. این تابع نیاز به شناسه Debuggee دارد . این شناسه را می توانید با استفاده از تابع GetWindowThreadProcessId بدست آورید . این تابع خود به شماره دسترسی پنجره به عنوان پارامتر ورودی نیاز دارد . پس ابتدا باید شماره دسترسی پنجره را بدست آورید . با داشتن نام کلاس پنجره مورد نظر می توانید توسط تابع FindWindow ، شماره دسترسی به پنجره ایجاد شده از آن کلاس را بدست آورید . اگر مقدار برگشتی Null باشد یعنی هیچ پنجره ای از آن کلاس ایجاد نشده است .

```
.if eax!=NULL
    invoke GetWindowThreadProcessId, eax, addr ProcessId
    mov ThreadId, eax
    invoke DebugActiveProcess, ProcessId
```

پس از اینکه شناسه Process را بدست آوریم ، تابع DebugActivePrpcess را فراخوانی می کنیم . سپس وارد حلقه دیباگ شده و منتظر رویداد می مانیم .

```
.if DBEvent.dwDebugEventCode==CREATE_PROCESS_DEBUG_EVENT
   mov context.ContextFlags, CONTEXT_CONTROL
   invoke GetThreadContext,\
          DBEvent.u.CreateProcessInfo.hThread,\
          addr context
```

پیغام CREATE_PROCESS_DEBUG_INFO نشانگر توقف Debuggee است . حال می توانیم تغییرات مورد نظر را در Process ایجاد کنیم . در این مثال دستور حلقه بی نهایت ($ Jmp) را با دو دستور NOP (90 h) جایگزین خواهیم کرد . توجه داشته باشید که دستور " $Jmp " از دو بایت تشکیل شده است که عبارتند از FEh و EBh .

ابتدا باید آدرس دستور " $ Jmp " را بدست آوریم . هنگامیکه برنامه ما به Debuggee متصل می شود ، Debuggee در حلقه بی نهایت است . پس eip آدرس دستور مورد نظر ما را در خود دارد . پس ابتدا باید مقدار eip را بدست آوریم . برای این کار ابتدا به عضو Context Flags از رکورد CONTEXT مقدار CONTEXT_CONTROL را می دهیم و سپس تابع

GetThreadCcontext را فراخوانی می کنیم . در حقیقت با انجام این کار ، رجیسترهای کنترل رکورد context را پر می کنیم .

```
invoke WriteProcessMemory,\
                  DBEvent.u.CreateProcessInfo.hProcess,\
                  context.regEip,\
                  addr buffer,\
                  2,\
                  NULL
```

حال که مقدار EIP را بدست آورده ایم ، می توانیم با استفاده از تابع WriteProcessMemory، برروی دستور " Jmp $ " دستورهای NOP را باز نویسی کنیم . در حقیقت با انجام این کار به Debuggee کمک می کنیم که از حلقه بی نهایت خارج شود . پس از انجام این کار ، پیغامی را به کاربر نمایش داده و تابع ContinueDebugEvent را برای باز گردانی Debuggee فراخوانی می کنیم . پس از انجام این کار Debuggee از حلقه بی نهایت خارج شده و پنجره خود را نمایش می دهد .

مثال بعد با استفاده از روش دیگری Debuggee را از حلقه بی نهایت خارج می کند.

```
.......
.......
.if DBEvent.dwDebugEventCode==CREATE_PROCESS_DEBUG_EVENT
   mov context.ContextFlags, CONTEXT_CONTROL
   invoke GetThreadContext,DBEvent.u.CreateProcessInfo.hThread,\
                           addr context

   add context.regEip,2
   invoke SetThreadContext,DBEvent.u.CreateProcessInfo.hThread,\
                           addr context

   invoke MessageBox, 0,\
                      addr LoopSkipped,\
                      addr AppName,\
                      MB_OK+MB_ICONINFORMATION

.......
.......
```

در این مثال نیز برای بدست آوردن مقدار EIP از تابع GetThreadContext استفاده می کنیم . ولی این بار به جای بازنویسی روی دستور " Jmp$ " ، مقدار رجیستر EIP را به اندازه ۲ بایت

افزایش می دهیم . با انجام این کار ، Debuggee پس از بازگشت ، اجرا را از دستور بعد از " Jmp $ " ادامه می دهد و در حقیقت یک پرش کوچک به اندازه ۲ بایت به جلو صورت می گیرد . توجه داشته باشید که اندازه دستور " Jmp $ " ۲ بایت است .

در صورت نیاز می توانید در رجیستر های دیگر نیز تغییراتی را ایجاد کنید . همچنین می توانید با وارد کردن int 3h در Debuggee ، در آن یک نقطه توقف (Break Point) ایجاد کنید .

Win32 Debug API (بخش ۳)

کدها و فایل‌های مربوط به این قسمت در CD ضمیمه موجود می‌باشد

SourceCodes\Asm32\Chapter23

در این فصل به بررسی نحوه اجرای خط به خط برنامه (Tracing) می پردازیم . اگر قبلا" از Debugger ها استفاده کرده باشید ، با نحوه Trace کردن آشنایی کافی دارید . هنگامی که برنامه ای را Trace می کنید ، برنامه پس از اجرای هر دستور متوقف شده و به شما اجازه بررسی مقادیر رجیسترها و حافظه را می دهد .

در حقیقت نام اصلی این عمل (Single_Stepping) است که یکی از خصوصیات CPU به شمار می آید . هشتمین بیت از ثبات Flag , Trap-Flag نامیده می شود . در صورت مقدار دهی این بیت ، CPU برنامه را در حالت Single-Step اجرا می کند . در این صورت CPU پس از اجرای هر دستور ، یک استثناءی دیباگ (Debug Exception) ایجاد می کند . پس از ایجاد این استثناء ، Trap-Flag به صورت خودکار پاک می شود و مقدار صفر می گیرد . در Win32 ، این کارها تماما" توسط توابع API انجام می گیرند .

در زیر مراحل استفاده از این توابع را مشاهده می کنید .

۱- برای بدست آوردن مقدار ثبات Flag (regflag) ، از تابع GetThreadContext استفاده می‌کنیم.

۲- Trap Bit را در regflag که از اعضای رکورد CONTEXT است ، ست می کنیم .

۳- تابع GetThreadContext را فراخوانی می کنیم .

٤- طبق معمول منتظر رویدادهای دیباگ می مانیم . Debugee در حالت Single_Step اجرا شده و ما پس از اجرای هر دستور ، رویداد EXCEPTION_DEBUG_EVENT را با مقدار EXCEPTION_SINGLE_STEP در u.Exception.pExceptionRecord.ExceptionCode ، در یافت خواهیم کرد .

۵- در صورت لزوم ، برای ادامه روند اجرایی ، مقدار Trap bit را دوباره ست می کنیم .

حال به سراغ مثال این بخش می رویم .

```
.386
.model flat,stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
include \masm32\include\comdlg32.inc
include \masm32\include\user32.inc
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\comdlg32.lib
includelib \masm32\lib\user32.lib

.data
AppName db "Win32 Debug Example no.4",0
ofn OPENFILENAME <>
FilterString db "Executable Files",0,"*.exe",0
            db "All Files",0,"*.*",0,0
ExitProc     db "The Debugee exits",0Dh,0Ah
            db "Total Instructions executed : %lu",0
TotalInstruction dd 0

.data?
buffer db 512 dup(?)
startinfo STARTUPINFO <>
pi PROCESS_INFORMATION <>
DBEvent DEBUG_EVENT <>
context CONTEXT <>

.code
start:
mov ofn.lStructSize,SIZEOF ofn
mov ofn.lpstrFilter, OFFSET FilterString
mov ofn.lpstrFile, OFFSET buffer
mov ofn.nMaxFile,512
mov ofn.Flags, OFN_FILEMUSTEXIST or OFN_PATHMUSTEXIST or
OFN_LONGNAMES or OFN_EXPLORER or OFN_HIDEREADONLY
invoke GetOpenFileName, ADDR ofn
.if eax==TRUE
    invoke GetStartupInfo,addr startinfo
    invoke CreateProcess, addr buffer,\
                          NULL,\
                          NULL,\
                          NULL,\
                          FALSE,\
                          DEBUG_PROCESS+ DEBUG_ONLY_THIS_PROCESS,\
                          NULL,\
                          NULL,\
                          addr startinfo,\
                          addr pi

    .while TRUE
       invoke WaitForDebugEvent, addr DBEvent, INFINITE
```

```
       .if DBEvent.dwDebugEventCode==EXIT_PROCESS_DEBUG_EVENT
          invoke wsprintf, addr buffer,\
                           addr ExitProc,\
                           TotalInstruction

          invoke MessageBox, 0,\
                             addr buffer,\
                             addr AppName,\
                             MB_OK + MB_ICONINFORMATION
          .break
       .elseif DBEvent.dwDebugEventCode==EXCEPTION_DEBUG_EVENT

.if DBEvent.u.Exception.pExceptionRecord.ExceptionCode ==\
EXCEPTION_BREAKPOINT
            mov context.ContextFlags, CONTEXT_CONTROL
            invoke GetThreadContext, pi.hThread, addr context
            or context.regFlag,100h
            invoke SetThreadContext,pi.hThread, addr context
            invoke ContinueDebugEvent, DBEvent.dwProcessId,\
DBEvent.dwThreadId,\
DBG_CONTINUE
            .continue
          .elseif DBEvent.u.Exception.pExceptionRecord.ExceptionCode
==\
EXCEPTION_SINGLE_STEP
            inc TotalInstruction
            invoke GetThreadContext,pi.hThread,\
                                    addr context or
                                    context.regFlag,\
                                    100h
            invoke SetThreadContext,pi.hThread, addr context
            invoke ContinueDebugEvent, DBEvent.dwProcessId,\
                                       DBEvent.dwThreadId,\
                                       DBG_CONTINUE
            .continue
          .endif
       .endif
       invoke ContinueDebugEvent, DBEvent.dwProcessId,\
                                  DBEvent.dwThreadId,\
                                  DBG_EXCEPTION_NOT_HANDLED
    .endw
.endif
invoke CloseHandle,pi.hProcess
invoke CloseHandle,pi.hThread
invoke ExitProcess, 0
end start
```

حال به تجزیه وتحلیل کدهای برنامه می پردازیم .

برنامه ابتدا یک دیالوگ OpenFile را نمایش داده و پس از انتخاب فایل اجرایی توسط کاربر ، آنرا بصورت مرحله به مرحله اجرا کرده و تعداد دستورات اجرا شده را تا خاتمه آن می شمارد .

```
 .elseif DBEvent.dwDebugEventCode==EXCEPTION_DEBUG_EVENT
.if DBEvent.u.Exception.pExceptionRecord.ExceptionCode ==\
EXCEPTION_BREAKPOINT
```

در این بخش مشخص می کنیم که برنامه مورد نظر باید به صورت مرحله به مرحله اجرا شود . همانطور که می دانید و یندوز پیش از اجرای اولین دستور Debugee ، رویداد EXCEPTION_BREAKPOINT را ایجاد می کند .

```
mov context.ContextFlags, CONTEXT_CONTROL
invoke GetThreadContext, pi.hThread, addr context
```

تابع GetThreadContent را برای پر کردن رکود CONTEXT بامقادیر فعلی ثبات ها ، فراخوانی می کنیم .

```
or context.regFlag,100h
```

مقدار trap bit راست می کنیم .

```
invoke SetThreadContext,pi.hThread, addr context
invoke ContinueDebugEvent, DBEvent.dwProcessId,\
                           DBEvent.dwThreadId,\
                           DBG_CONTINUE
.continue
```

تابع SetThreadContent را برای بازنویسی مقادیر جدید در رکورد CONTEXT فراخوانی کرده و برای ادامه کار Debugee ، از تابع ContinueDebugEvent و فلگ DBG_CONTINUE استفاده می کنیم .

```
.elseif DBEvent.u.Exception.pExceptionRecord.ExceptionCode ==\
EXCEPTION_SINGLE_STEP
     inc TotalInstruction
```

با اجرای هر دستور در Debugee ، یک رویداد EXCEPTION_DEBVG_EVENT دریافت می کنیم که در اینصورت باید مقدار BEvent.u.Exception.pExceptionRecord.ExceptionCode را مورد بررسی قرار دهیم . در صورتی که این مقدار EXCEPTION_SINGLE_STEP باشد ، مشخص کننده اینست که یک دستور اجرا شده است و مقدار متغیر TotalInstruction را یک واحد افزایش می دهیم .

```
invoke GetThreadContext,pi.hThread,\
                    addr context or context.regFlag,\
                    100h
invoke SetThreadContext,pi.hThread, addr context
invoke ContinueDebugEvent, DBEvent.dwProcessId,\
                           DBEvent.dwThreadId,\
                           DBG_CONTINUE
.continue
```

پس از ایجاد هر استثناء Trap bit پاک می شود . در صورتیکه می خواهید به اجرای مرحله به مرحله ادامه دهید ، باید دوباره آنرا ست کنید .

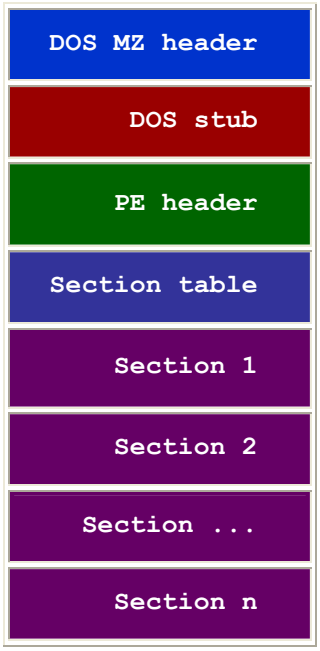
توجه داشته باشید که برای تست کردن مثال این بخش از برنامه های بزرگ استفاده نکنید زیرا ممکن است مجبور شوید مدت زمان زیادی منتظر بمانید .

ساختار فایلهای اجرایی (بخش ۱)

در این بخش ساختار کلی فایلهای اجرایی در ویندوز را مورد بررسی قرار خواهیم داد.

PE به معنای فایل اجرایی قابل حمل (Portable Executable) است که فرمت اصلی فایلهای اجرایی در Win32 می باشد . لفظ قابل حمل بودن به این معنا است که فرمت این فایلها در کلیه نسخه های ۳۲ بیتی ویندوز یکسان است . بارگذار فایلهای اجرایی در این ویندوزها این فایلها را شناسایی کرده و از آنها استفاده می کند ، با وجود اینکه ممکن است دستگاه مورد نظر با پرداز شگر دیگری غیر از Intel کار کند . کلیه فایلهای اجرایی در ویندوز از فرمت PE استفاده می کنند ، پس آشنایی با این فرمت می تواند دید ارزشمندی نسبت به ساختار و نحوه عملکرد فایلهای اجرایی در ویندوز را برای شما ایجاد کند.

حال ساختار کلی این فرصت را مورد بررسی قرا ر می دهیم .



کلیه فایلهای اجرایی در Win32 با یک سرآیند ساده Dos MZ شروع می شوند . در صورتیکه فایل اجرایی در محیط Dos اجرا شود ، این سرآیند باعث می شود که این سیستم عامل آنرا به عنوان فایل اجرایی معتبر شناسایی کرده و دستورات درون بخش Dos Stub را اجرا کند . معمولا" با اجرای Dos Stub رشته "This Program Requires Windows " به نمایش درمی آید ولی در صورت نیاز این بخش می تواند حاوی یک برنامه کامل Dos نیز باشد . این بخش معمولا" توسط کامپایلر به صورت اتوماتیک به فایل اجرایی اضافه می شود که معمولا" از وقفه ۲۱ و سرویس ۹ برای چاپ رشته مورد نظر استفاده می کند .

بخش بعدی PE Header نام دارد . درحقیقت PE Header نام دیگری برای رکورد IMAGE_NT_HEADERS محسوب می شود . این رکورد حاوی فیلدهایی است که توسط بارگزار PE مورد استفاده قرار می گیرند . در بخش های بعد با این رکورد آشنایی بیشتری پیدا خواهید کرد . در زمان اجرای برنامه ، بار گذار PE می تواند آدرس شروع PE Header را توسط سرآیند Dos MZ شناسایی کند . در نتیجه می تواند بخش Dos Stub را نادیده گرفته و به سراغ PE Header که سرآیند اصلی فایل است ، برود .

محتوای اصلی فایل اجرایی به بلوکهایی تقسیم می شود که Section نامیده می شوند . Section چیزی جز بلوکی از داده ها با یک صفت مشترک نیست . برای تصور بهتر می توانید فایل PE را همانند یک دیسک در نظر بگیرید . قسمت PE header همانند Boot Sector ، و Section ها همانند فایلها هستند . این فایلها می توانند دارای خصوصیات مختلفی باشند . به این نکته توجه داشته باشید که قرار دادن داده ها در یک Section بر اساس صفات مشترک آنها است نه بر اساس عملیات منطقی آنها . مهم نیست که چگونه از Code یا Data استفاده می کنید . اگر آنها دارای صفات مشترکی باشند می توانند با هم در یک Section قرار بگیرند . نباید یک Section را به عنوان " Code " و Section دیگر را به عنوان " Data " در نظر بگیرید . زیرا یک Section می تواند شامل هر دوی آنها نیز باشد و این اسامی تنها جنبه شناسایی و نام گذاری دارند . به عنوان مثال اگر می خواهید بلوکی از داده ها دارای صفت Read-Only باشد ، می توانید آنرا در یک Section با صفت Read-Only قرار دهید .هنگامی که بار گذار PE هر Section را به حافظه بارگذاری می کند ، ابتدا مشخصات آنرا مورد بررسی قرار داده و به بلوکی از حافظه که توسط این Section اشغال شده است ، مشخصات مورد نظر را نسبت می دهد .

اگر فرمت PE همانند یک دیسک و PE Header همانند Boot Sector و Section ها همانند فایلها باشند ، هنوز اطلاع کافی در مورد محل قرار گرفتن فایلها در دیسک نخواهیم داشت .بعد از بخش PE header ، Section Table قرار گرفته است که خود آرایه ای از رکوردها است . هر رکوردحاوی اطلاعاتی از قبیل صفات ، آفست ، آفست مجازی و ... در مورد یک Section است . به عنوان مثال

اگر یک فایل PE شامل پنج Section باشد ، این آرایه نیز شامل پنج رکورد خواهد بود . بخش Section table را می توان همانند دایرکتوری ریشه در دیسک در نظر گرفت . هر یک از اعضای این آرایه به منزله آدرس ورودی برای یک شاخه اصلی محسوب می شود . با مطالب گفته شده ، با ساختار فیزیکی یک فایل اجرایی در ویندوز آشنایی پیدا کردید. در بخشهای بعد جزئیات این ساختار را مورد بررسی قرار خواهیم داد .

ساختار فایلهای اجرایی (بخش ۲)

کدها و فایلهای مربوط به این قسمت در CD ضمیمه موجود می‌باشد

SourceCodes\Asm32\Chapter25

در این فصل به بررسی چگونگی تشخیص یک فایل اجرایی معتبر می پردازیم .

چگونه تشخیص دهیم که یک فایل ، دارای ساختار یک فایل اجرایی است؟ پاسخ این سوال مشکل است و بستگی به میزان دقت مورد نظر شما دارد . به عنوان مثال می توانید تمام رکوردهای داخل فایل اجرایی را مورد بررسی قرار دهید و یا اینکه به بررسی تنها یکی از آنها اکتفا کنید . در عمل با بررسی رکوردهای اصلی می توانید تا حد قابل قبولی اعتبار یک فایل اجرایی را مشخص کنید و دیگر نیازی به بررسی تک تک رکوردها نیست .

مهم ترین رکوردی که باید اعتبار آنرا تشخیص دهیم ، رکورد PE Header است که در ادامه به بررسی جزئیات آن خواهیم پرداخت . PE Header در واقع رکوردی از نوع IMAGE_NT_HEADERS است . حال به بررسی ساختار این رکورد می پردازیم .

```
IMAGE_NT_HEADERS STRUCT
   Signature        dd ?
   FileHeader       IMAGE_FILE_HEADER <>
   OptionalHeader   IMAGE_OPTIONAL_HEADER32 <>
IMAGE_NT_HEADERS ENDS
```

Signature متغییری از نوع dword است که حاوی مقدار 50h , 45h , 00h , 00h می باشد. به عبارت دیگر شامل رشته " PE " به همراه دو صفر است . این عضو در حقیقت شناسه فایل اجرایی است که برای تشخیص اعتبار فایل از آن استفاده می کنیم .

FileHeader رکوردی است که حاوی اطلاعاتی درباره ساختار فیزیکی فایل اجرایی می باشد . مانند تعداد Section ها ، نوع کامپیوتری که فایل روی آن اجرا می شود و ... .

Optional Header رکوردی است که حاوی اطلاعاتی درباره ساختار منطقی فایل اجرایی می باشد .

اکنون هدف ما مشخص شد . اگر مقدار عضو Signature از رکورد IMAGE_NT_HEADER برابر با "PE" به همراه دو صفر باشد ، می توانیم نتیجه بگیریم که فایل اجرایی بررسی شده معتبر

است . در واقع مایکروسافت برای مقایسه ، یک ثابت با نام IMAGE_NT_SIGNATURE را تعریف کرده است که ما از آن استفاده خواهیم کرد . در زیر ، لیست این ثابت ها و مقادیر آنها را مشاهده می کنید .

```
IMAGE_DOS_SIGNATURE             equ 5A4Dh
IMAGE_OS2_SIGNATURE             equ 454Eh
IMAGE_OS2_SIGNATURE_LE          equ 454Ch
IMAGE_VXD_SIGNATURE             equ 454Ch
IMAGE_NT_SIGNATURE              equ 4550h
```

سوال بعدی این است که چگونه آدرس رکورد PE Header را در فایل پیدا کنیم . پاسخ این سوال ساده است. رکورد DOS MZ Header ، آدرس PE Header را در خود دارد . در حقیقت DOS MZ Header رکوردی از نوع IMAGE_DOS_HEADER است که عضو e_Ifanew از این رکورد حاوی آدرس PE Header در فایل می باشد .

در نتیجه مراحل کار به صورت زیر خلاصه می شوند .

۱- برای تعیین اعتبار DOS MZ Header ، مقدار ۲ بایت اول آنرا با مقدار IMAGE_DOS_HEADER مقایسه می کنیم .

۲- در صورت معتبر بودن DOS MZ Header ، از مقدار e_Ifanew برای بدست آوردن آدرس PE Header استفاده می کنیم .

۳- مقدار دو بایت اول از PE Header را با IMAGE_NT_HEADER مقایسه می کنیم . در صورت برابری این دو مقدار ، نتیجه می گیریم فایل مورد نظر یک فایل اجرایی معتبر است .

```
.386
.model flat,stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
include \masm32\include\comdlg32.inc
include \masm32\include\user32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\comdlg32.lib

SEH struct
PrevLink dd ?
CurrentHandler dd ?
SafeOffset dd ?
PrevEsp dd ?
PrevEbp dd ?
SEH ends

.data
AppName db "PE tutorial no.2",0
ofn OPENFILENAME <>
FilterString db "Executable Files (*.exe, *.dll)",\
             0,\
             "*.exe;*.dll",\
             0,\
             "All Files",\
             0,\
             "*.*",\
             0,\
             0
db "Cannot open the file for reading",0       FileOpenError
FileOpenMappingError db "Cannot open the file for memory mapping",0
db "Cannot map the file into memory",0    FileMappingError
db "This file is a valid PE",0         FileValidPE
db "This file is not a valid PE",0       FileInValidPE

.data?
buffer db 512 dup(?)
hFile dd ?
hMapping dd ?
pMapping dd ?
ValidPE dd ?

.code
start proc
LOCAL seh:SEH
mov ofn.lStructSize,SIZEOF ofn
mov ofn.lpstrFilter, OFFSET FilterString
mov ofn.lpstrFile, OFFSET buffer
```

```
mov ofn.nMaxFile,512
mov ofn.Flags, OFN_FILEMUSTEXIST or \
OFN_PATHMUSTEXIST or \
OFN_LONGNAMES or \
OFN_EXPLORER or \
OFN_HIDEREADONLY
invoke GetOpenFileName, ADDR ofn
.if eax==TRUE
    invoke CreateFile,  addr buffer,\
                        GENERIC_READ,\
                        FILE_SHARE_READ,\
                        NULL,\
                        OPEN_EXISTING,\
                        FILE_ATTRIBUTE_NORMAL,\
                        NULL
    .if eax!=INVALID_HANDLE_VALUE
       mov hFile, eax
       invoke CreateFileMapping, hFile, NULL, PAGE_READONLY,0,0,0
       .if eax!=NULL
          mov hMapping, eax
          invoke MapViewOfFile,hMapping,FILE_MAP_READ,0,0,0
          .if eax!=NULL
             mov pMapping,eax
             assume fs:nothing
             push fs:[0]
             pop seh.PrevLink
             mov seh.CurrentHandler,offset SEHHandler
             mov seh.SafeOffset,offset FinalExit
             lea eax,seh
             mov fs:[0], eax
             mov seh.PrevEsp,esp
             mov seh.PrevEbp,ebp
             mov edi, pMapping
             assume edi:ptr IMAGE_DOS_HEADER
             .if [edi].e_magic==IMAGE_DOS_SIGNATURE
                add edi, [edi].e_lfanew
                assume edi:ptr IMAGE_NT_HEADERS
                .if [edi].Signature==IMAGE_NT_SIGNATURE
                   mov ValidPE, TRUE
                .else
                   mov ValidPE, FALSE
                .endif
             .else
                 mov ValidPE,FALSE
             .endif
FinalExit:
             .if ValidPE==TRUE
                 invoke MessageBox, 0,\
                                    addr FileValidPE,\
                                    addr AppName,\
                                    MB_OK + MB_ICONINFORMATION
```

```
            .else
               invoke MessageBox, 0,\
                                  addr FileInValidPE,\
                                  addr AppName,\
                                  MB_OK + MB_ICONINFORMATION
            .endif
            push seh.PrevLink
            pop fs:[0]
            invoke UnmapViewOfFile, pMapping
         .else
            invoke MessageBox, 0,\
                               addr FileMappingError,\
                               addr AppName,\
                               MB_OK + MB_ICONERROR
         .endif
         invoke CloseHandle,hMapping
      .else
         invoke MessageBox, 0,\
                            addr FileOpenMappingError,\
                            addr AppName,\
                            MB_OK + MB_ICONERROR
      .endif
      invoke CloseHandle, hFile
   .else
      invoke MessageBox, 0,\
                         addr FileOpenError,\
                         addr AppName,\
                         MB_OK + MB_ICONERROR
   .endif
.endif
invoke ExitProcess, 0
start endp

SEHHandler proc C uses edx pExcept:DWORD, pFrame:DWORD,
pContext:DWORD, pDispatch:DWORD
    mov edx,pFrame
    assume edx:ptr SEH
    mov eax,pContext
    assume eax:ptr CONTEXT
    push [edx].SafeOffset
    pop [eax].regEip
    push [edx].PrevEsp
    pop [eax].regEsp
    push [edx].PrevEbp
    pop [eax].regEbp
    mov ValidPE, FALSE
    mov eax,ExceptionContinueExecution
    ret
SEHHandler endp
end start
```

حال به تجزیه و تحلیل کد های بالا می پردازیم .

برنامه پس از باز کردن فایل ، ابتدا اعتبار DOS Header آنرا مشخص می کند . در صورت معتبر بودن DOS Header ، به سراغ PE Header رفته و اعتبار آنرا نیز بررسی می کنیم و در صورت معتبر بودن می توانیم نتیجه بگیریم که فایل مورد نظر یک فایل اجرایی است . ما در این مثال از سیستم مدیریت ساخت یافته خطاها استفاده کرده ایم که به اختصار به آن SEH (Structured Exception Handling) گفته می شود . در اینصورت دیگر نیازی نیست که کلیه خطاها را بررسی کنیم . در صورتی که خطایی رخ دهد با احتمال قابل قبولی می توانیم نتیجه بگیریم که فایل مورد نظر فایل اجرایی معتبری نیست زیرا به برنامه ما اطلاعات نادرستی را منتقل کرده است .

برنامه ابتدا یک دیالوگ Open File را نمایش داده و به کاربر اجازه می دهد که فایل اجرایی مورد نظر را انتخاب کند . سپس فایل را باز کرده و به حافظه نگاشت می کند . قبل از اینکه به سراغ بخش شناسایی فایل برویم ، سیستم مدیریت خطاها (SEH) را راه اندازی می کنیم .

```
assume fs:nothing
push fs:[0]
pop seh.PrevLink
mov seh.CurrentHandler,offset SEHHandler
mov seh.SafeOffset,offset FinalExit
lea eax,seh
mov fs:[0], eax
mov seh.PrevEsp,esp
mov seh.PrevEbp,ebp
```

ابتدا با دستور اول نحوه استفاده از رجیستر fs را تغییر می دهیم زیرا Masm از این ثبات به منظور مدیریت خطاهای بوجود آمده استفاده می کند . سپس آدرس SEHHandler قبلی را برای استفاده ویندوز در رکورد خود ذخیره کرده و آدرس SEHHandler خود را جایگزین آن می کنیم . در حقیقت با این کار مشخص می کنیم که در صورت بوجود آمدن خطا ، برنامه می تواند با رفتن به آدرس مشخص شده ، کار خود را با موفقیت دنبال کند. در مرحله بعد مقادیر ثباتهای esp و ebp را ذخیره می کنیم تا در موقع بازگشت بتوانیم وضعیت Stack را به حالت قبلی خود باز گردانیم .

```
mov edi, pMapping
assume edi:ptr IMAGE_DOS_HEADER
.if [edi].e_magic==IMAGE_DOS_SIGNATURE
```

پس از راه اندازی SEH ، به سراغ شناسایی فایل می رویم . ابتدا آدرس اولین بایت از فایل مورد نظر را در edi می ریزیم که اولین بایت رکورد DOS MZ Header است . برای آسان تر شدن

عملیات مقایسه ، edi را به عنوان اشاره گری به رکورد IMAGE_DOS_HEADER معرفی می کنیم. سپس دو بایت اول از Dos Header را با رشته " MZ " که در فایل Windows.inc با نام IMAGE_DOS_SIGNITURE ثبت شده است ، مقایسه می کنیم . در صورت برابری به سراغ PH header می رویم و در غیر این صورت به متغییر ValidPEمقدار False می دهیم .

```
add edi, [edi].e_lfanew
assume edi:ptr IMAGE_NT_HEADERS
.if [edi].Signature==IMAGE_NT_SIGNATURE
   mov ValidPE, TRUE
.else
   mov ValidPE, FALSE
.endif
```

برای دستیابی به آدرس PE Header ، به مقدار عضو e_Ifanew از DOS Header نیاز داریم . این مقدار را در edi قرار می دهیم . اینجا همان جایی است که احتمال بروز خطا وجود دارد . در صورتی که فایل مورد نظر ، فایل اجرایی معتبری نباشد ، ممکن است مقدار e_Ifanew با اندازه فایل مطابقت نداشته باشد .

در صورتی که از SEH استفاده نمی کردیم ، مجبور بودیم مقدار e_Ifanew را با اندازه فایل مقایسه کنیم . اگر همه موارد با موفقیت انجام شدند ، دو بایت اول از رکورد PE Header را با رشته " PE " مقایسه می کنیم که در فایل Windows.inc با نام IMAGE_NT_SIGNITURE ثبت شده است . در صورت برابری ، با اطمینان نسبتا" خوبی می توانیم نتیجه بگیریم که فایل مورد نظر یک فایل اجرایی معتبر است .

در صورتیکه مقدار e_Ifanew نامعتبر باشد ، خطایی رخ می دهد که SEHHandler کنترل آنرا به دست گرفته و Stack و رجیستر ها را به وضعیت قبلی خود بر می گرداند . سپس اجرای برنامه توسطSEHHandler به بخش Final Exit منتقل می شود .

```
FinalExit:
   .if ValidPE==TRUE
      invoke MessageBox, 0,\
                         addr FileValidPE,\
                         addr AppName,\
                         MB_OK + MB_ICONINFORMATION
   .else
      invoke MessageBox, 0,\
                         addr FileInValidPE,\
                         addr AppName,\
                         MB_OK + MB_ICONINFORMATION
   .endif
```

کد بالا مقدار متغییر ValidPE را بررسی کرده و پیغام مناسب را به کاربر نمایش می دهد .

```
push seh.PrevLink
pop fs:[0]
```

در آخر هم به علت اینکه دیگر نیازی به SEH نداریم ، آنرا به حالت قبلی خود بر می گردانیم .

ساختار فایلهای اجرایی (بخش ۳) File Header

در این فصل در مورد بخش File Header از PE Header مطالبی خواهیم آموخت .

ابتدا مطالب فصل قبل را مرور می کنیم .

- Dos MZ Header به عنوان رکوردی از نوع IMAGE_DOS_HEADER تعریف شده است که تنها دو عضو آن در اینجا برای ما اهمیت دارند :

۱ : e_magic که شامل رشته " MZ " است .

۲ : e_Ifanew که آدرس PE Header را در خود دارد .

- ابتدا از مقدار e_magic برای بررسی اعتبار Dos Header استفاده می کنیم .
- برای دستیابی به PE Header از مقدار e_Ifanew استفاده می کنیم .
- اولین dword از PE Header باید شامل رشته " PE " به همراه دوصفر باشد. مقدار این dword را با مقدار IMAGE_NT_SIGNITURE مقایسه می کنیم . اگر هر دو مقدار یکسان بودند ، نتیجه می گیریم که فایل مورد نظر یک فایل اجرایی معتبر است .

در این فصل مطالب بیشتری را در مورد PE Header خواهیم آموخت .

همانطور که می دانید PE Header در واقع رکوردی از نوع IMAGE_NT_HEADER است که برای یادآوری ساختار آنرا بار دیگر بیان می کنیم .

```
IMAGE_NT_HEADERS STRUCT
    Signature       dd ?
    FileHeader      IMAGE_FILE_HEADER <>
    OptionalHeader  IMAGE_OPTIONAL_HEADER32 <>
IMAGE_NT_HEADERS ENDS
```

Signature شناسه فایل اجرایی است یعنی رشته " PE " به همراه دو صفر.

File Header رکوردی است که مشخصات فیزیکی فایل اجرایی را مشخص می کند .

Optional Header رکوردی است که حاوی اطلاعاتی درباره ساختار منطقی درون فایل اجرایی می باشد .

مهمترین اطلاعات در بخش Optional Header قرار دارد ولی برخی از اعضای File Header نیز برای ما سودمند خواهند بود . در این فصل مطالبی را در مورد File Header خواهیم آموخت و در فصل بعد به سراغ Optional Header خواهیم رفت .

در زیر ، ساختار File Header را مشاهده می کنید .

```
IMAGE_FILE_HEADER STRUCT
    Machine                 WORD   ?
    NumberOfSections        WORD   ?
    TimeDateStamp           dd     ?
    PointerToSymbolTable    dd     ?
    NumberOfSymbols         dd     ?
    SizeOfOptionalHeader    WORD   ?
    Characteristics         WORD   ?
IMAGE_FILE_HEADER ENDS
```

Machine مدل پرداز شگر مورد نیاز برای اجرای دستورات فایل اجرایی را مشخص می کند . مقدار این عضو برای پردازش گرهای Intel برابر 14Ch است که در فایل Windows.inc با نام IMAGE_FILE_MACHINE_1386 تعریف شده است . در صورت استفاده از مقادیر غیر مجاز ، ویندوز از اجرای فایل جلوگیری می کند که می تواند در مواردی سودمند باشد .

NumberOfSections تعداد Section های موجود در فایل را مشخص می کند . در صورتی که می خواهید یک Section به فایل اضافه و یا کم کنید ، باید مقدار این عضو را تغییر دهید .

TimeDateStamp تاریخ و زمان ایجاد فایل را مشخص می کند .

PointerToSymbolTable از این عضو تنها در زمان دیباگ کردن استفاده می شود .

NumberOfSymbols از این عضو نیز تنها در زمان دیباگ کردن استفاده می شود .

SizeOfOptionalHeader اندازه رکورد OptionalHeader را مشخص می کند که دقیقا" بعد از رکورد File Header قرار دارد .

Characteristics شامل یک فلگ است که نوع فایل اجرایی را مشخص می کند . برای مثال exe یا dll .

توجه داشته باشید که هنگام کار با Section Table باید از مقدار Number Of Sections استفاده کنید . حال به طور خلاصه به بررسی Section Table می پردازیم .

Section Table آرایه ای از رکوردها است که هر رکورد حاوی اطلاعاتی در مورد یک Section است . پس اگر ۳ Section داشته باشیم, آرایه ما سه عضوی خواهد بود . با توجه به مقدار NumberOfSections می توانیم تعداد عناصر آرایه را شناسایی کنیم . اگر از تعداد Section ها مطلع نباشیم ، می توانیم خواندن آنها را ادامه دهیم تا به رکوردی برسیم که تمام اعضای آن صفر باشند . ویندوز از این روش استفاده می کند . برای امتحان می توانید NumberOfSections را بیشتر از مقدار واقعی آن قرار دهید و خواهید دید که ویندوز فایل را بدون هیچ مشکلی اجرا می کند . با این وجود هنوز نمی توانید مقدار NumberOfSections را نادیده بگیرید زیرا برخی از کامپایلرها خاتمه رکوردها به رکورد صفر را رعایت نمی کنند و در اینصورت نادیده گرفتن این مقدار مشکلات زیادی را برای شما ایجاد خواهد کرد .

ساختار فایلهای اجرایی (بخش ۴) Optional Header

در فصلهای گذشته درباره Dos Header و برخی از اعضای PE Header نکاتی را آموختید . اکنون می خواهیم درباره بزرگترین و مهمترین عضو PE Header یعنی Optional Header مطالبی را بیان کنیم .

رکورد Optional Header آخرین عضو IMAGE_NT_HEADERS محسوب می شود که حاوی اطلاعاتی درباره ساختار منطقی فایل اجرایی می باشد. این رکورد شامل ۳۱ عضو است که برخی از آنها از اهمیت بیشتری برخوردار هستند و در این بخش قصد داریم توضیحاتی را درباره این اعضاء بیان کنیم .

در توضیحات به کلمه RVA برخورد خواهید کرد که در ادامه به توضیح آن می پردازیم .

RVA مخفف Relative Virtual Address است که مفهوم ساده ای دارد . در حقیقت RVA میزان فاصله از یک نقطه مرجع از فضای حافظه برنامه را بیان می کند که دقیقا" همانند یک آفست عمل می کند . توجه داشته باشید که RVA به محلی از فضای مجازی حافظه برنامه وابسته است نه به فایل اجرایی . استفاده از RVA باعث ساده تر شدن عملیات بارگذاری فایل می شود زیرا یک ماژول(فایل اجرایی) می تواند در جاهای مختلفی از فضای حافظه برنامه قرار بگیرد و مسلما" برای بارگذار PE بسیار مشکل است اگر بخواهد تمام موارد ارجاع از قبیل پرشها ، فراخوانیها و ..... را در فایل اجرایی تصحیح کند . به بیان دیگر اگر تمام موارد ارجاع از RVA استفاده کنند دیگر نیازی نیست که بارگذار PE چیزی را تصحیح کند و فایل اجرایی به سادگی می تواند به نقاط مختلف از فضای حافظه برنامه منتقل شود .

AddressOfEntryPoint آدرس اولین دستور فایل اجرایی را مشخص می کند . در صورتیکه می خواهید اجرای برنامه را از آدرس دیگری شروع کنید ، باید مقدار این عضو را تغییر دهید .

ImageBase حاوی آدرسی است که ترجیحا" بارگذار PE فایل اجرایی را در آنجا قرار می دهد . برای مثال اگر مقدار این عضو برابر با 400000 h باشد ، بارگذار PE سعی می کند که فایل را در آدرس 400000 h از حافظه برنامه قرار دهد . کلمه ترجیحا" به این معنی است که اگر آدرس مورد نظر قبلا" اشغال شده باشد ، بارگذار PE فایل را در آدرس دیگری از حافظه قرار می دهد .

SectionAlignment آدرس شروع هر Section در حافظه را به مضارب صحیحی از مقدار مورد نظر محدود می کند . به عنوان مثال اگر مقدار این عضو برابر با 4096 (1000 h) باشد ، آدرس شروع هر Section ، از مضرب صحیحی از ٤٠٩٦ شروع می شود . اگر اولین Section از

آدرس 401000 h شروع شود و سایز آن 10 بایت باشد ، Section بعدی باید از آدرس 402000 h شروع شود حتی اگر فضای حافظه بین این دو آدرس بلا استفاده بماند .

FileAlignment این عضو چگونگی قرار گرفتن تک تک Section ها را در فایل مشخص می کند که عملکردی مشابه عضو قبلی دارد .

SizeOfImage اندازه کلی فایل اجرایی در حافظه را مشخص می کند که مجموع Header ها و Section هایی است که با قوانین مشخص شده توسط عضو Section Alignment در کنار هم قرار گرفته اند .

SizeOfHeaders حجم کل Header ها به همراه Section Table را مشخص می کند . به طور خلاصه این مقدار برابر حجم کلی فایل منهای حجم Section ها است. از این مقدار می توانید به عنوان آفست اولین Section در فایل اجرایی استفاده کنید .

Data Directory آرایه ای از رکورد IMAGE_DATA_DIRECTORY است که هر رکورد حاوی RVA مربوط به یک رکورد مهم از فایل اجرایی مانند Import Address Table است .

ساختار فایلهای اجرایی (بخش ۵) Section Table

کدها و فایلهای مربوط به این قسمت در CD ضمیمه موجود می‌باشد

SourceCodes\Asm32\Chapter28

Section Table در واقع آرایه ای از رکوردها است که دقیقا" بعد از PE Header قرار گرفته است . تعداد اعضاء این آرایه توسط Number Of Sections که یکی از اعضای رکورد File Header است ، مشخص می شود . رکوردهای این آرایه از نوع IMAGE_SECTION_HEADER هستند که در زیر ، ساختار آنرا مشاهده می کنید .

```
IMAGE_SIZEOF_SHORT_NAME equ 8
IMAGE_SECTION_HEADER STRUCT
   Name1                   db IMAGE_SIZEOF_SHORT_NAME dup(?)
   union Misc
      PhysicalAddress      dd ?
      VirtualSize          dd ?
   ends
   VirtualAddress          dd ?
   SizeOfRawData           dd ?
   PointerToRawData              dd ?
   PointerToRelocations    dd ?
   PointerToLinenumbers    dd ?
   NumberOfRelocations     dw ?
   NumberOfLinenumbers     dw ?
   Characteristics         dd ?
IMAGE_SECTION_HEADER ENDS
```

حال برخی از اعضای این رکورد را که برای ما از اهمیت بیشتری برخوردارند ، بررسی می کنیم .

Name1 نام اصلی این عضو ، Name است ولی به دلیل اینکه این کلمه از حروف کلیدی MASM می باشد ، به جای آن از کلمه Name1 استفاده می کنیم . این عضو نام Section را معین می کند که حداکثر طول آن ۸ بایت است . این نام چیزی جز یک برچسب نیست و می تواند هر رشته دلخواهی را شامل شود .

Virtual Address RVA مربوط به هر Section را مشخص می کند . بارگذار PE هنگام نگاشت هر Section به حافظه ، از این مقدار استفاده می کند .

SizeOfRawData فضای اشغال شده توسط Section در فایل اجرایی را با در نظر گرفتن مقدار FileAlignment مشخص می کند .

PointerToRawData آدرس شروع Section را در فایل اجرایی مشخص می کند .

Characteristics شامل فلگهایی است که خصوصیات و نوع Section را مشخص می کند .

حال که مطالبی را در مورد رکورد IMAGE_SECTION_HEADER آموختید ، به بررسی عملکرد بارگذار PE می پردازیم .

١- ابتدا تعداد Section ها توسط عضو NumberOfSections از رکورد IMAGE_FILE_HEADER تعیین می شود .

٢- مقدار SizeOfHeaders به عنوان آدرس شروع Section Table در فایل ، خوانده شده و اشاره گر فایل به آن آدرس منتقل می شود .

٣- بر روی آرایه رکوردها حرکت کرده و هر عضو آن بررسی می شود .

٤- برای هر رکورد مقدار PointerToRawData گرفته شده و اشاره گر فایل به آن آفست منتقل می شود. سپس به میزان SizeOfRawData از فایل خوانده شده و برای بدست آوردن آدرس مجازی شروع Section در حافظه ، مقدار VirtualAddress خوانده شده و به مقدار ImageBase اضافه می شود . حال که اطلاعات کافی در مورد Section ها و محل قرارگیری آنها در حافظه برنامه بدست آورده ایم ، می توانیم آنرا به حافظه نگاشت کنیم .

٥- این عمل برای تک تک رکوردهای آرایه انجام می شود .

حال به سراغ مثال این بخش می رویم .

در این مثال یک فایل اجرایی را باز کرده و با حرکت برروی Section Table اطلاعات مربوط به Section ها رادر یک کنترل ListView نمایش می دهد .

```
.386
.model flat,stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
include \masm32\include\comdlg32.inc
include \masm32\include\user32.inc
include \masm32\include\comctl32.inc
includelib \masm32\lib\comctl32.lib
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\comdlg32.lib

IDD_SECTIONTABLE equ 104
IDC_SECTIONLIST equ 1001

SEH struct
dd ? ; the address of the previous seh structure    PrevLink
CurrentHandler dd ? ; the address of the new exception handler
dd ? ; The offset where it's safe to continue execution  SafeOffset
dd ? ; the old value in esp         PrevEsp
dd ? ; The old value in ebp         PrevEbp
SEH ends

.data
db "PE tutorial no.5",0         AppName
OPENFILENAME <>         ofn
FilterString
            db "Executable Files (*.exe, *.dll)",\
            0,\
            "*.exe;*.dll",\
            0,\
            "All Files",\
            0,\
            "*.*",\
            0,\
            0
db "Cannot open the file for reading",0       FileOpenError
FileOpenMappingError db "Cannot open the file for memory mapping",0
db "Cannot map the file into memory",0    FileMappingError
db "This file is not a valid PE",0       FileInValidPE
db "%08lx",0        template
db "Section",0         SectionName
db "V.Size",0         VirtualSize
db "V.Address",0  VirtualAddress
db "Raw Size",0       SizeOfRawData
db "Raw Offset",0       RawOffset
db "Characteristics",0     Characteristics

.data?
dd ?       hInstance
```

```
db 512 dup(?)          buffer
dd ?           hFile
dd ?        hMapping
dd ?        pMapping
dd ?             ValidPE
dd ?    NumberOfSections

.code
start proc
LOCAL seh:SEH
   invoke GetModuleHandle,NULL
   mov hInstance,eax
   mov ofn.lStructSize,SIZEOF ofn
   mov ofn.lpstrFilter, OFFSET FilterString
   mov ofn.lpstrFile, OFFSET buffer
   mov ofn.nMaxFile,512
   mov ofn.Flags, OFN_FILEMUSTEXIST or \
                  OFN_PATHMUSTEXIST or \
                  OFN_LONGNAMES or \
                  OFN_EXPLORER or \
                  OFN_HIDEREADONLY
   invoke GetOpenFileName, ADDR ofn
   .if eax==TRUE
      invoke CreateFile, addr buffer,\
                         GENERIC_READ,\
                         FILE_SHARE_READ,\
                         NULL,\
                         OPEN_EXISTING,\
                         FILE_ATTRIBUTE_NORMAL,\
                         NULL
      .if eax!=INVALID_HANDLE_VALUE
         mov hFile, eax
         invoke CreateFileMapping, hFile, NULL, PAGE_READONLY,0,0,0
         .if eax!=NULL
            mov hMapping, eax
            invoke MapViewOfFile,hMapping,FILE_MAP_READ,0,0,0
            .if eax!=NULL
               mov pMapping,eax
               assume fs:nothing
               push fs:[0]
               pop seh.PrevLink
               mov seh.CurrentHandler,offset SEHHandler
               mov seh.SafeOffset,offset FinalExit
               lea eax,seh
               mov fs:[0], eax
               mov seh.PrevEsp,esp
               mov seh.PrevEbp,ebp
               mov edi, pMapping
               assume edi:ptr IMAGE_DOS_HEADER
               .if [edi].e_magic==IMAGE_DOS_SIGNATURE
                  add edi, [edi].e_lfanew
                  assume edi:ptr IMAGE_NT_HEADERS
```

```
                  .if [edi].Signature==IMAGE_NT_SIGNATURE
                     mov ValidPE, TRUE
                  .else
                     mov ValidPE, FALSE
                  .endif
               .else
                  mov ValidPE,FALSE
               .endif
FinalExit:
               push seh.PrevLink
               pop fs:[0]
               .if ValidPE==TRUE
                  call ShowSectionInfo
               .else
                  invoke MessageBox, 0,\
                                     addr FileInValidPE,\
                                     addr AppName,\
                                     MB_OK + MB_ICONINFORMATION
               .endif
               invoke UnmapViewOfFile, pMapping
            .else
               invoke MessageBox, 0,\
                                  addr FileMappingError,\
                                  addr AppName,\
                                  MB_OK + MB_ICONERROR
            .endif
            invoke CloseHandle,hMapping
         .else
            invoke MessageBox, 0,\
                               addr FileOpenMappingError,\
                               addr AppName,\
                               MB_OK + MB_ICONERROR
         .endif
         invoke CloseHandle, hFile
      .else
         invoke MessageBox, 0,\
                            addr FileOpenError,\
                            addr AppName,\
                            MB_OK + MB_ICONERROR
      .endif
   .endif
   invoke ExitProcess, 0
   invoke InitCommonControls
start endp

SEHHandler proc C uses
pExcept:DWORD,pFrame:DWORD,pContext:DWORD,pDispatch:DWORD
   mov edx,pFrame
   assume edx:ptr SEH
   mov eax,pContext
   assume eax:ptr CONTEXT
   push [edx].SafeOffset
```

```
   pop [eax].regEip
   push [edx].PrevEsp
   pop [eax].regEsp
   push [edx].PrevEbp
   pop [eax].regEbp
   mov ValidPE, FALSE
   mov eax,ExceptionContinueExecution
   ret
SEHHandler endp

DlgProc proc uses edi esi hDlg:DWORD, uMsg:DWORD, wParam:DWORD,
lParam:DWORD
   LOCAL lvc:LV_COLUMN
   LOCAL lvi:LV_ITEM
   .if uMsg==WM_INITDIALOG
      mov esi, lParam
      mov lvc.imask,LVCF_FMT or \
                    LVCF_TEXT or \
                    LVCF_WIDTH or \
                    LVCF_SUBITEM

      mov lvc.fmt,LVCFMT_LEFT
      mov lvc.lx,80
      mov lvc.iSubItem,0
      mov lvc.pszText,offset SectionName
      invoke SendDlgItemMessage, hDlg,\
                                 IDC_SECTIONLIST,\
                                 LVM_INSERTCOLUMN,\
                                 0,\
                                 addr lvc
      inc lvc.iSubItem
      mov lvc.fmt,LVCFMT_RIGHT
      mov lvc.pszText,offset VirtualSize
      invoke SendDlgItemMessage, hDlg,\
                                 IDC_SECTIONLIST,\
                                 LVM_INSERTCOLUMN,\
                                 1,\
                                 addr lvc
      inc lvc.iSubItem
      mov lvc.pszText,offset VirtualAddress
      invoke SendDlgItemMessage, hDlg,\
                                 IDC_SECTIONLIST,\
                                 LVM_INSERTCOLUMN,\
                                 2,\
                                 addr lvc
      inc lvc.iSubItem
      mov lvc.pszText,offset SizeOfRawData
      invoke SendDlgItemMessage, hDlg,\
                                 IDC_SECTIONLIST,\
                                 LVM_INSERTCOLUMN,\
                                 3,\
                                 addr lvc
```

```
inc lvc.iSubItem
mov lvc.pszText,offset RawOffset
invoke SendDlgItemMessage, hDlg,\
                           IDC_SECTIONLIST,\
                           LVM_INSERTCOLUMN,\
                           4,\
                           addr lvc
inc lvc.iSubItem
mov lvc.pszText,offset Characteristics
invoke SendDlgItemMessage, hDlg,\
                           IDC_SECTIONLIST,\
                           LVM_INSERTCOLUMN,\
                           5,\
                           addr lvc
mov ax, NumberOfSections
movzx eax,ax
mov edi,eax
mov lvi.imask,LVIF_TEXT
mov lvi.iItem,0
assume esi:ptr IMAGE_SECTION_HEADER
.while edi>0
   mov lvi.iSubItem,0
   invoke RtlZeroMemory,addr buffer,9
   invoke lstrcpyn,addr buffer,addr [esi].Name1,8
   lea eax,buffer
   mov lvi.pszText,eax
   invoke SendDlgItemMessage, hDlg,\
                              IDC_SECTIONLIST,\
                              LVM_INSERTITEM,\
                              0,\
                              addr lvi
   invoke wsprintf,addr buffer,\
                   addr template,\
                   [esi].Misc.VirtualSize

   lea eax,buffer
   mov lvi.pszText,eax
   inc lvi.iSubItem
   invoke SendDlgItemMessage, hDlg,\
                              IDC_SECTIONLIST,\
                              LVM_SETITEM,\
                              0,\
                              addr lvi
   invoke wsprintf,addr buffer,\
                   addr template,\
                   [esi].VirtualAddress

   lea eax,buffer
   mov lvi.pszText,eax
   inc lvi.iSubItem
   invoke SendDlgItemMessage, hDlg,\
                              IDC_SECTIONLIST,\
```

```
                                LVM_SETITEM,\
                                0,\
                                addr lvi
            invoke wsprintf,addr buffer,\
                            addr template,\
                            [esi].SizeOfRawData
            lea eax,buffer
            mov lvi.pszText,eax
            inc lvi.iSubItem
            invoke SendDlgItemMessage, hDlg,\
                                IDC_SECTIONLIST,\
                                LVM_SETITEM,\
                                0,\
                                addr lvi
            invoke wsprintf,addr buffer,\
                            addr template,\
                            [esi].PointerToRawData

            lea eax,buffer
            mov lvi.pszText,eax
            inc lvi.iSubItem
            invoke SendDlgItemMessage, hDlg,\
                                IDC_SECTIONLIST,\
                                LVM_SETITEM,\
                                0,\
                                addr lvi
            invoke wsprintf,addr buffer,\
                            addr template,\
                            [esi].Characteristics

            lea eax,buffer
            mov lvi.pszText,eax
            inc lvi.iSubItem
            invoke SendDlgItemMessage, hDlg,\
                                IDC_SECTIONLIST,\
                                LVM_SETITEM,\
                                0,\
                                addr lvi
            inc lvi.iItem
            dec edi
            add esi, sizeof IMAGE_SECTION_HEADER
         .endw
      .elseif
         uMsg==WM_CLOSE
            invoke EndDialog,hDlg,NULL
      .else
         mov eax,FALSE
         ret
      .endif
      mov eax,TRUE
      ret
DlgProc endp
```

```
ShowSectionInfo proc uses edi
   mov edi, pMapping
   assume edi:ptr IMAGE_DOS_HEADER
   add edi, [edi].e_lfanew
   assume edi:ptr IMAGE_NT_HEADERS
   mov ax,[edi].FileHeader.NumberOfSections
   movzx eax,ax
   mov NumberOfSections,eax
   add edi,sizeof IMAGE_NT_HEADERS
   invoke DialogBoxParam,  hInstance,\
                           IDD_SECTIONTABLE,\
                           NULL,\
                           addr DlgProc,\
                           edi

   ret
ShowSectionInfo endp
end start
```

این مثال ابتدا از کدهای بخش ۲ برای بررسی اعتبار فایل اجرایی مورد نظر استفاده کرده و سپس تابع ShowSectionInfo را فراخوانی می کند .

```
ShowSectionInfo proc uses edi
   mov edi, pMapping
   assume edi:ptr IMAGE_DOS_HEADER
   add edi, [edi].e_lfanew
   assume edi:ptr IMAGE_NT_HEADERS
```

از edi به عنوان اشاره گر به داده ها در فایل اجرایی استفاده کرده و آنرا با PMapping که در حقیقت آدرس Dos Header است ، مقدار دهی اولیه می کنیم. در مرحله بعد برای دستیابی آدرس PE Header ، مقدار e_Ifanew را با آن جمع می کنیم . حال edi حاوی آدرس PE Header است.

```
   mov ax,[edi].FileHeader.NumberOfSections
   mov NumberOfSections,ax
```

برای حرکت بر روی Section Table ، به تعداد Section ها نیاز داریم که توسط عضو NumberOfSections از رکورد File Header تعیین شده است . فراموش نکنید که این عضو از نوع word است .

```
   add edi,sizeof IMAGE_NT_HEADERS
```

edi حاوی آدرس PE Header است . در نتیجه اضافه کردن اندازه PE Header ، آنرا تبدیل به اشاره گری به Section Table می کند .

```
invoke DialogBoxParam, hInstance,\
                       IDD_SECTIONTABLE,\
                       NULL,\
                       addr DlgProc,\
                       edi
```

تابع DialogBoxParram را برای نمایش دیالوگ باکس برنامه فراخوانی می کنیم . توجه داشته باشید که آدرس Section Table را به عنوان پارامتر آخر ، به این تابع می فرستیم که در هنگام رویداد WM_INITDIALOG ، توسط lParam به پروسیجر دیالوگ منتقل می شود .

در پروسیجر دیالوگ، با دریافت پیغام WM_INITDIALOG، ابتدا مقدار lParam (آدرس Section Table) را در esi و تعداد Section ها را در edi ذخیره کرده و سپس کنترل ListView را پر می کنیم . در مرحله بعد وارد یک حلقه شده و اطلاعات مربوط به هر Section را به لیست اضافه می کنیم .

```
.while edi>0
   mov lvi.iSubItem,0
```

رشته مورد نظر را در اولین سطر لیست قرار می دهیم .

```
invoke RtlZeroMemory,addr buffer,9
invoke lstrcpyn,addr buffer,addr [esi].Name1,8
lea eax,buffer
mov lvi.pszText,eax
```

برای نشان دادن نام Section ، ابتدا آنرا تبدیل به یک رشته ASCII مختوم به صفر می کنیم .

```
invoke SendDlgItemMessag hDlg,\
                         IDC_SECTIONLIST,\
                         LVM_INSERTITEM,\
                         0,\
                         addr lvi
```

رشته مورد نظر را به لیست اضافه کرده و به سراغ Section بعدی می رویم .

```
        dec edi
        add esi, sizeof IMAGE_SECTION_HEADER
    .endw
```

پس از پردازش هر رکورد ، یک واحد از مقدار edi کم کرده و اندازه IMAGE_SECTION_HEADER را به esi اضافه می کنیم . با انجام این کار esi به رکورد IMAGE_SECTION_HEADER بعدی اشاره خواهد کرد .

حال به طور خلاصه مراحل کار را مرور می کنیم .

۱- تشخیص اعتبار فایل اجرایی مورد نظر .

۲- رفتن به ابتدای PE Header .

۳- بدست آوردن تعداد Section ها از عضو NumberOfSections در File Header .

٤- رفتن به ابتدای Section Table با اضافه کردن ImageBase به SizeofHeaders یا با اضافه کردن آدرس PE Header به اندازه PE Header . (Section Table دقیقا" بعد از PE Header قرار دارد).

اگر از نگاشت فایل استفاده نمی کنید و باید اشاره گر فایل را با استفاده از تابع Setfilepointer به Section table منتقل کنید .

٥- پردازش رکوردهای IMAGE_SECTION_HEADER.

ساختار فایلهای اجرایی (بخش ۶) Import Table

کدها و فایلهای مربوط به این قسمت در CD ضمیمه موجود می‌باشد

SourceCodes\Asm32\Chapter29

در این بخش قصد داریم درباره جدول ورودی (Import Table) مطالبی را بیان کنیم.

ابتدا باید بدانید که تابع ورودی چیست . تابع ورودی زیر برنامه ای است که در فایل اجرایی قرار ندارد ولی توسط آن فراخوانی می شود . توابع ورودی در یک یا چند فایل dll قرار می گیرند و فقط اطلاعات مربوط به آنها در فایل اجرایی ذخیره می شود . این اطلاعات عبارتند از نام تابع و فایلی که تابع مورد نظر درآن قرار گرفته است . حال چگونه می توانیم بفهمیم که این اطلاعات در کدام قسمت از فایل اجرایی قرار گرفته است .

برای پاسخ به این سوال به سراغ Data Directory می رویم . حال برای یادآوری نگاهی دوباره به ساختار PE Header می اندازیم .

```
IMAGE_NT_HEADERS STRUCT
   Signature         dd ?
   FileHeader        IMAGE_FILE_HEADER <>
   OptionalHeader    IMAGE_OPTIONAL_HEADER <>
IMAGE_NT_HEADERS ENDS
```

Data Directory آخرین عضو از رکورد Optional Header است . که خود آرایه ای ١٦ تایی از رکوردهای IMAGE_DATA_DIRECTORY می باشد .

```
IMAGE_OPTIONAL_HEADER32 STRUCT
   ....
   LoaderFlags                                    dd ?
   NumberOfRvaAndSizes                            dd ?
   DataDirectory
    IMAGE_DATA_DIRECTORY 16 dup(<>)
IMAGE_OPTIONAL_HEADER32 ENDS
```

هر رکورد IMAGE_DATA_DIRECTORY حاوی اطلاعات مهمی درباره یک رکورد داده در فایل اجرایی می باشد .

Member	Info inside
0	Export symbols
1	Import symbols
2	Resources
3	Exception
4	Security
5	Base relocation
6	Debug
7	Copyright string
8	Unknown
9	Thread local storage (TLS)
10	Load configuration
11	Bound Import
12	Import Address Table
13	Delay Import
14	COM descriptor

حال به بررسی جزئی اعضای این رکورد می پردازیم .

هر یک از اعضای Data Directory رکوردی از نوع IMAGE_DATA_DIRECTORY است که در زیر ، ساختار آنرا مشاهده می کنید .

```
IMAGE_DATA_DIRECTORY STRUCT
  VirtualAddress    dd ?
  isize             dd ?
IMAGE_DATA_DIRECTORY ENDS
```

VirtualAddress در واقع RVA رکورد داده است . به عنوان مثال اگر این رکورد مربوط به ImportSymbols باشد ، این فیلد حاوی RVA آرایه IMAGE_IMPORT_DESCRIPTOR است .

isize اندازه رکورد داده را مشخص می کند .

حال روش کلی پیداکردن رکوردهای داده مهم در فایل اجرایی را با هم بررسی می کنیم .

١- توسط Dos Header به PE Header می رویم .

٢- آدرس شروع Data Directory را از Optional Header بدست می آوریم .

٣- اندازه رکورد IMAGE_DATA_DIRECTORY را در اندیس عضو مورد نظر ضرب می کنیم . به عنوان مثال در صورتیکه می خواهید از محل قرار گرفتن Import Symbols مطلع شوید ، اندازه رکورد IMAGE_DATA_DIRECTORY (که ٨ است) را در یک ضرب می کنیم .

٤- حاصل را با آدرس شروع Data Directory جمع می کنیم .

حال شما آدرس رکورد IMAGE_DATA_DIRECTORY مربوط به عضو مورد نظر را در اختیار دارید که حاوی اطلاعاتی در مورد رکورد داده مورد نظر شما است .

حال به سراغ Import Table می رویم . آدرس Import Table در بخش Virtual Address از عضو دوم آرایه Data Directory قرار دارد . Import Table در واقع آرایه ای از رکورد های IMAGE_IMPORT_DESCRIPTOR است که هر یک حاوی اطلاعاتی درباره یک dll است که فایل اجرایی از آن استفاده می کند . به عنوان مثال اگر یک فایل اجرایی از ١٠ dll مختلف استفاده کند ، در این آرایه ١٠ عضو وجود خواهد داشت . انتهای آرایه نیز با یک رکورد صفر مشخص می شود .

حال به بررسی جزئی این رکورد می پردازیم .

```
IMAGE_IMPORT_DESCRIPTOR STRUCT
  union
    Characteristics              dd ?
    OriginalFirstThunk           dd ?
  ends
  TimeDateStamp                  dd ?
  ForwarderChain                 dd ?
  Name1                          dd ?
  FirstThunk                     dd ?
IMAGE_IMPORT_DESCRIPTOR ENDS
```

اولین عضو این رکورد یک union است که در حقیقت یک نام مستعار برای OriginalFirstThunk تعیین می کند . در نتیجه می توانید آنرا Characteristics بنامید . این عضو حاوی RVA آرایه ای از رکورد IMAGE_THUNK_DATA است . IMAGE_THUNK_DATA یک union باندازه dword است که معمولا" آنرا به عنوان اشاره گری به یک رکورد IMAGE_IMPORT_BY_NAME در نظر می گیریم . توجه داشته باشید که تعداد زیادی کورد IMAGE_IMPORT_BY_NAME وجود دارد که RVA آنها (IMAGE_THUNK_DATA) را در آرایه دیگری ذخیره کرده و با قرار دادن یک رکورد صفر ، خاتمه آنرا مشخص می کنیم .

حال RVA آرایه ایجاد شده را در OriginalFirstThunk قرار می دهیم . هر رکورد IMAGE_IMPORT_BY_NAME حاوی اطلاعاتی درباره یک تابع ورودی است . حال به بررسی این رکورد می پردازیم .

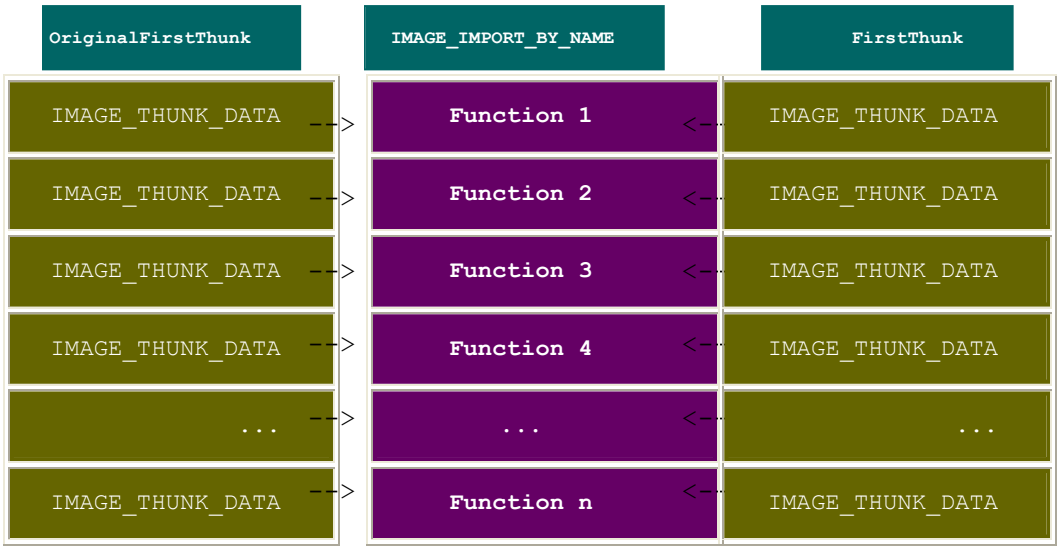
```
IMAGE_IMPORT_BY_NAME STRUCT
  Hint      dw ?
  Name1     db ?
IMAGE_IMPORT_BY_NAME ENDS
```

Hint : حاوی اندیسی از جدول خروجی (export table) dll ای است که تابع در آن قرار دارد . این مقدار اهمیت زیادی ندارد و برخی از Linker ها مقدار آنرا صفر قرار می دهند .

Namel : نام تابع ورودی را بصورت یک رشته اسکی مختوم به صفر مشخص می کند .

First Thunk : دقیقا" شبیه OriginalFirstThunk می باشد که حاوی RVA آرایه رکوردهای IMAGE_THUNK_DATA است . به بیان دیگر First Thunk یک کپی از Original First Thunk است که قبلا" آنرا بررسی کرده ایم .

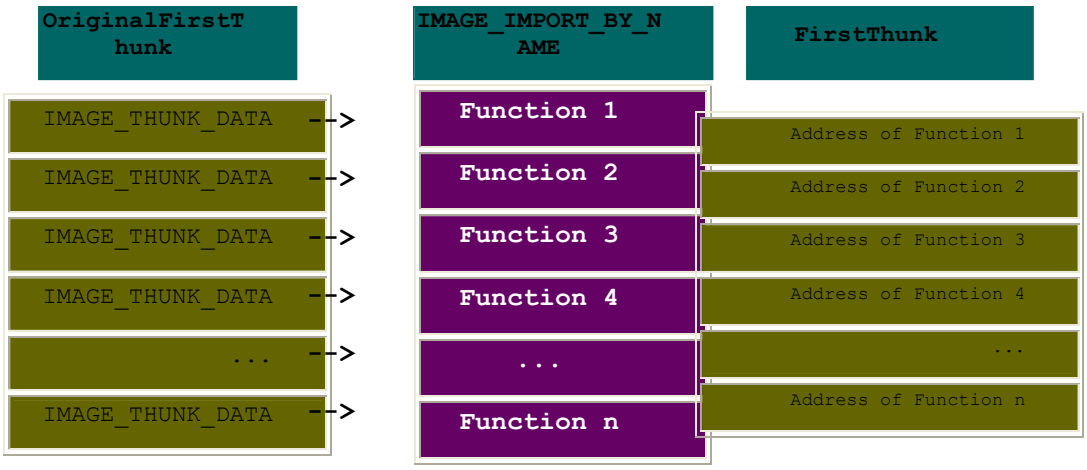
با نگاهی به شکل زیر می توانید تصور بهتری از مطالب گفته شده پیدا کنید .



تعداد اعضای Original First Thunk و First Thunk بستگی به تعداد توابعی دارد که فایل اجرایی از یک فایل dll وارد می کند . به عنوان مثال اگر یک فایل اجرایی از ۱۰ تابع درون Kernel32.dll استفاده کند ، عضو Name1 از رکورد IMAGE_IMPORT_DESCRIPTOR حاوی RVA رشته Kernel32. dll بوده و هر آرایه شامل ۱۰ عضو از نوع IMAGE_THUNK_DATA خواهد بود .

سوال بعدی اینست که چرا باید از دو آرایه استفاده کنیم ؟

در پاسخ به این سوال باید بدانید که هنگام اجرای یک فایل ، ابتدا بارگذار PE با استفاده ازآرایه های IMAGE_IMPORT_BY_NAME و IMAGE_THUNK_DATA آدرس توابع مورد نیاز برنامه را مشخص کرده و آنها را در آرایه First Thunk قرار می دهد . درنتیجه با اجرای فایل شکل قبل بصورت زیر در می آید .



با کمی توجه به شکل بالا در می یابید که OriginalFirstThunk تغییری نکرده است و در صورت نیاز برای تعیین توابع ورودی می توان از آن استفاده کرد. در این طرح یک مشکل کوچک وجود دارد و آن اینست که برخی توابع تنها توسط شماره شان شناسایی می شوند . این بدان معنی است که به جای استفاده از نام تابع ، از موقعیت آن استفاده می شود . در این حالت دیگر رکوردی با نام IMAGE_IMPORT_BY_NAME وجود نخواهد داشت و به جای آن IMAGE_THUNK_DATA در قسمت Low word حاوی شماره تابع بوده و بیت پر ارزش آن (MSB) مقدار ۱ خواهد داشت .

به‌عنوان مثال برای مشخص کردن یک تابع با شماره 1234h ، مقدار IMAGE_THUNK_DATA برای آن تابع 80001234 h خواهد بود . برای تست کردن بیت پر ارزش از مقدار 80000000 h استفاده می کنیم که در فایل Windows.inc با نام IMAGE_ORDINAL_FLAG32 تعریف شده است .

حال مراحل تعیین توابع ورودی یک فایل اجرایی را بصورت مرحله به مرحله مرور می کنیم .

۱- تشخیص اعتبار فایل اجرایی .

۲- رفتن از Dos Header به PE Header .

۳- بدست آوردن آدرس Data Directory از Optional Header .

٤- رفتن به دومین عضو Data Directory و تعیین مقدار Virtual Address .

٥- استفاده‌ازمقدار فوق برای دستیابی به‌اولین رکوردIMAGE_IMPORT_DESCRIPTOR.

٦- چک کردن مقدار OriginulFirstThunk. در صورتیکه این مقدار صفر نبود ، RVA ها را در OriginulFirstThunk دنبال می کنیم ولی اگر مقدار آن صفر بود ، به جای آن از مقدار First Thunk استفاده می کنیم . توجه داشته باشید که برخی از Linker ها فایلهای اجرایی را با مقدار صفر در OriginalFirstThunk ایجاد می کنند .

۷- برای هر عضو آرایه مقدار بیت پر ارزش را چک می کنیم . در صورتی که این مقدار برابر یک باشد می توانیم نتیجه بگیریم که تابع توسط شماره اش شناسایی می شود . در این صورت شماره تابع در قسمتLow word از آن عضو قرار دارد .

۸- در صورتی که مقدار بیت پر ارزش برابر صفر باشد ، از مقدار عضو مورد نظر به عنوان اشاره گری به رکورد IMAGE_IMPORT_BY_NAME استفاده کرده و نام تابع را توسط عضو Name1 از این رکورد بدست می آوریم .

۹- عملیات بالا را برای تمام عناصر آرایه تا رسیدن به رکورد صفر ادامه داده و در پایان به سراغ فایل dll بعدی می رویم .

۱۰- مراحل ٦ تا ۹ را برای رکوردهای IMAGE_IMPORT_DESCRIPTOR تکرار کرده و این کار را تا رسیدن به انتهای آرایه رکوردها که با رکورد صفر مشخص می شود ادامه می دهیم .

حال به سراغ مثال این بخش می رویم .

این مثال یک فایل اجرایی را باز کرده و لیستی از کلیه توابع ورودی را به همراه مقادیر اعضای رکوردهای IMAGE_IMPORT_DESCRIPTOR ، در یک کنترل Edit نمایش می دهد .

```
.386
.model flat,stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
include \masm32\include\comdlg32.inc
include \masm32\include\user32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\comdlg32.lib

equ 101     IDD_MAINDLG
equ 1000    IDC_EDIT
equ 40001    IDM_OPEN
equ 40003    IDM_EXIT

proto :DWORD,:DWORD,:DWORD,:DWORD             DlgProc
proto :DWORD        ShowImportFunctions
proto :DWORD,:DWORD    ShowTheFunctions
proto :DWORD,:DWORD      AppendText

SEH struct
dd ? ; the address of the previous seh structure    PrevLink
CurrentHandler dd ? ; the address of the new exception handler
dd ? ; The offset where it's safe to continue execution  SafeOffset
dd ? ; the old value in esp         PrevEsp
dd ? ; The old value in ebp         PrevEbp
SEH ends

.data
AppName db "PE tutorial no.6",0
ofn OPENFILENAME <>
FilterString db "Executable Files (*.exe, *.dll)",\
             0,\
             "*.exe;*.dll",\
             0,\
             "All Files",\
             0,\
             "*.*",\
             0,\
             0
db "Cannot open the file for reading",0       FileOpenError
FileOpenMappingError db "Cannot open the file for memory mapping",0
db "Cannot map the file into memory",0    FileMappingError
db "This file is not a valid PE",0      NotValidPE
db 0Dh,0Ah,0            CRLF
db 0Dh,0Ah,"===[ IMAGE_IMPORT_DESCRIPTOR ]===",0    ImportDescriptor

db "OriginalFirstThunk = %lX",0Dh,0Ah      IDTemplate
db "TimeDateStamp = %lX",0Dh,0Ah
db "ForwarderChain = %lX",0Dh,0Ah
```

```
db "Name = %s",0Dh,0Ah
db "FirstThunk = %lX",0
db 0Dh,0Ah,"Hint Function",0Dh,0Ah      NameHeader
db "-------------------------------",0
db "%u %s",0         NameTemplate
db "%u (ord.)",0     OrdinalTemplate

.data?
db 512 dup(?)  buffer
dd ?   hFile
hMapping dd ?
pMapping dd ?
dd ?     ValidPE

.code
start:
invoke GetModuleHandle,NULL
invoke DialogBoxParam, eax, IDD_MAINDLG,NULL,addr DlgProc, 0
invoke ExitProcess, 0

DlgProc proc hDlg:DWORD, uMsg:DWORD, wParam:DWORD, lParam:DWORD
.if uMsg==WM_INITDIALOG
  invoke SendDlgItemMessage,hDlg,IDC_EDIT,EM_SETLIMITTEXT,0,0
.elseif uMsg==WM_CLOSE
  invoke EndDialog,hDlg,0
.elseif uMsg==WM_COMMAND
  .if lParam==0
    mov eax,wParam
    .if ax==IDM_OPEN
      invoke ShowImportFunctions,hDlg
    .else ; IDM_EXIT
      invoke SendMessage,hDlg,WM_CLOSE,0,0
    .endif
  .endif
.else
  mov eax,FALSE
  ret
.endif
mov eax,TRUE
ret
DlgProc endp

SEHHandler proc C pExcept:DWORD, pFrame:DWORD, pContext:DWORD,
pDispatch:DWORD
  mov edx,pFrame
  assume edx:ptr SEH
  mov eax,pContext
  assume eax:ptr CONTEXT
  push [edx].SafeOffset
  pop [eax].regEip
  push [edx].PrevEsp
  pop [eax].regEsp
```

```
  push [edx].PrevEbp
  pop [eax].regEbp
  mov ValidPE, FALSE
  mov eax,ExceptionContinueExecution
  ret
SEHHandler endp

ShowImportFunctions proc uses edi hDlg:DWORD
  LOCAL seh:SEH
  mov ofn.lStructSize,SIZEOF
  ofn mov ofn.lpstrFilter, OFFSET FilterString
  mov ofn.lpstrFile, OFFSET buffer
  mov ofn.nMaxFile,512
  mov ofn.Flags,OFN_FILEMUSTEXIST or \
                OFN_PATHMUSTEXIST or \
                OFN_LONGNAMES or \
                OFN_EXPLORER or \
                OFN_HIDEREADONLY
  invoke GetOpenFileName, ADDR ofn
  .if eax==TRUE
    invoke CreateFile, addr buffer,\
                       GENERIC_READ,\
                       FILE_SHARE_READ,\
                       NULL,\
                       OPEN_EXISTING,\
                       FILE_ATTRIBUTE_NORMAL,\
                       NULL
    .if eax!=INVALID_HANDLE_VALUE
      mov hFile, eax
      invoke CreateFileMapping, hFile, NULL, PAGE_READONLY,0,0,0
      .if eax!=NULL
        mov hMapping, eax
        invoke MapViewOfFile,hMapping,FILE_MAP_READ,0,0,0
        .if eax!=NULL
          mov pMapping,eax
          assume fs:nothing
          push fs:[0]
          pop seh.PrevLink
          mov seh.CurrentHandler,offset SEHHandler
          mov seh.SafeOffset,offset FinalExit
          lea eax,seh
          mov fs:[0], eax
          mov seh.PrevEsp,esp
          mov seh.PrevEbp,ebp
          mov edi, pMapping
          assume edi:ptr IMAGE_DOS_HEADER
          .if [edi].e_magic==IMAGE_DOS_SIGNATURE
            add edi, [edi].e_lfanew
            assume edi:ptr IMAGE_NT_HEADERS
            .if [edi].Signature==IMAGE_NT_SIGNATURE
              mov ValidPE, TRUE
            .else
```

```
              mov ValidPE, FALSE
            .endif
          .else
            mov ValidPE,FALSE
          .endif
FinalExit:
          push seh.PrevLink
          pop fs:[0]
          .if ValidPE==TRUE
            invoke ShowTheFunctions, hDlg, edi
          .else
            invoke MessageBox,0,\
                              addr NotValidPE,\
                              addr AppName,\
                              MB_OK + MB_ICONERROR
          .endif
          invoke UnmapViewOfFile, pMapping
      .else
          invoke MessageBox, 0,\
                              addr FileMappingError,\
                              addr AppName,\
                              MB_OK + MB_ICONERROR
      .endif
      invoke CloseHandle,hMapping
    .else
      invoke MessageBox, 0,\
                          addr FileOpenMappingError,\
                          addr AppName,\
                          MB_OK + MB_ICONERROR
    .endif
    invoke CloseHandle, hFile
   .else
   invoke MessageBox, 0,\
                      addr FileOpenError,\
                      addr AppName,\
                      MB_OK + MB_ICONERROR
   .endif
 .endif
 ret
ShowImportFunctions endp

AppendText proc hDlg:DWORD,pText:DWORD
   invoke SendDlgItemMessage,hDlg,IDC_EDIT,EM_REPLACESEL,0,pText
   invoke SendDlgItemMessage,hDlg,IDC_EDIT,EM_REPLACESEL,0,addr CRLF

   invoke SendDlgItemMessage,hDlg,IDC_EDIT,EM_SETSEL,-1,0
   ret
AppendText endp

RVAToOffset PROC uses edi esi edx ecx pFileMap:DWORD,RVA:DWORD
   mov esi,pFileMap
   assume esi:ptr IMAGE_DOS_HEADER
```

```
   add esi,[esi].e_lfanew
   assume esi:ptr IMAGE_NT_HEADERS
   mov edi,RVA ; edi == RVA
   mov edx,esi
   add edx,sizeof IMAGE_NT_HEADERS
   mov cx,[esi].FileHeader.NumberOfSections
   movzx ecx,cx
   assume edx:ptr IMAGE_SECTION_HEADER
   .while ecx>0 ; check all sections
     .if edi>=[edx].VirtualAddress
       mov eax,[edx].VirtualAddress
       add eax,[edx].SizeOfRawData
       .if edi<eax ; The address is in this section
         mov eax,[edx].VirtualAddress
         sub edi,eax
         mov eax,[edx].PointerToRawData
         add eax,edi ; eax == file offset
         ret
       .endif
     .endif
     add edx,sizeof IMAGE_SECTION_HEADER
     dec ecx
   .endw
   assume edx:nothing
   assume esi:nothing
   mov eax,edi
   ret
RVAToOffset endp

ShowTheFunctions proc uses esi ecx ebx hDlg:DWORD, pNTHdr:DWORD
   LOCAL temp[512]:BYTE
   invoke SetDlgItemText,hDlg,IDC_EDIT,0
   invoke AppendText,hDlg,addr buffer
   mov edi,pNTHdr
   assume edi:ptr IMAGE_NT_HEADERS
   mov edi, [edi].OptionalHeader.DataDirectory[sizeof
IMAGE_DATA_DIRECTORY].VirtualAddress
   invoke RVAToOffset,pMapping,edi
   mov edi,eax
   add edi,pMapping
   assume edi:ptr IMAGE_IMPORT_DESCRIPTOR
   .while !([edi].OriginalFirstThunk==0 && \
            [edi].TimeDateStamp==0 && \
            [edi].ForwarderChain==0 && \
            [edi].Name1==0 && \
            [edi].FirstThunk==0)
     invoke AppendText,hDlg,addr ImportDescriptor
     invoke RVAToOffset,pMapping, [edi].Name1
     mov edx,eax
     add edx,pMapping
     invoke wsprintf, addr temp,\
                      addr IDTemplate,\
```

```
                        [edi].OriginalFirstThunk,\
                        [edi].TimeDateStamp,\
                        [edi].ForwarderChain,edx,[edi].FirstThunk

  invoke AppendText,hDlg,addr temp
     .if [edi].OriginalFirstThunk==0
        mov esi,[edi].FirstThunk
     .else
        mov esi,[edi].OriginalFirstThunk
     .endif
     invoke RVAToOffset,pMapping,esi
     add eax,pMapping
     mov esi,eax
     invoke AppendText,hDlg,addr NameHeader
     .while dword ptr [esi]!=0
       test dword ptr [esi],IMAGE_ORDINAL_FLAG32
       jnz ImportByOrdinal
       invoke RVAToOffset,pMapping,dword ptr [esi]
       mov edx,eax
       add edx,pMapping
       assume edx:ptr IMAGE_IMPORT_BY_NAME
       mov cx, [edx].Hint
       movzx ecx,cx
       invoke wsprintf,addr temp,\
                       addr NameTemplate,\
                       ecx,\
                       addr [edx].Name1
       jmp ShowTheText
ImportByOrdinal:
       mov edx,dword ptr [esi]
       and edx,0FFFFh
       invoke wsprintf,addr temp,addr OrdinalTemplate,edx
ShowTheText:
       invoke AppendText,hDlg,addr temp
       add esi,4
    .endw
    add edi,sizeof IMAGE_IMPORT_DESCRIPTOR
  .endw
  ret
ShowTheFunctions endp
end start
```

حال به بررسی برنامه می پردازیم .

برنامه در ابتدا توسط یک دیالوگ Open File ، فایل اجرایی مورد نظر کاربر را مشخص کرده ، و سپس اعتبار آنرا مورد بررسی قرار می دهد . در مرحله بعد تابع ShowTheFunctions فراخوانی می شود که وظیفه آن دریافت و نمایش اطلاعات مربوط به توابع ورودی فایل اجرایی مورد نظر است .

```
ShowTheFunctions proc uses esi ecx ebx hDlg:DWORD, pNTHdr:DWORD
   LOCAL temp[512]:BYTE
```

512 بایت از پشته را برای عملیات کار با رشته ها اختصاص می دهیم .

```
   invoke SetDlgItemText,hDlg,IDC_EDIT,0
```

متن موجود در کنترل Edit را پاک می کنیم .

```
   invoke AppendText,hDlg,addr buffer
```

نام فایل اجرایی را در کنترل Edit قرار می دهیم .تابع AppendText از پیغام EM_REPLACE برای اضافه کردن متن به کنترل Edit استفاده می کند . توجه داشته باشید که این تابع از پیغام EM_SETSEL با وضعیت wParam = 1 و lParam = 0 برای انتقال کر سر متن به انتهای آن استفاده می کند .

```
   mov edi,pNTHdr
   assume edi:ptr IMAGE_NT_HEADERS
   mov edi, [edi].OptionalHeader.DataDirectory[sizeof
IMAGE_DATA_DIRECTORY].VirtualAddress
```

در این مرحله ابتدا RVA قسمت ImportSymbols از Data Directory را گرفته و سپس مقدار عضو VirtualAddress آنرا بدست می آوریم .

```
   invoke RVAToOffset,pMapping,edi
   mov edi,eax
   add edi,pMapping
```

اکثر آدرس ها در فایلهای اجرایی بصورت RVA هستند و RVA تنها در شرایطی با معنی است که فایل اجرایی توسط بارگذار PE به حافظه بارگذاری شده باشد . در این مثال ، فایل را به حافظه

نگاشت می کنیم ولی نحوه نگاشت ، با نگاشت بارگذار PE کاملا" متفاوت است . در نتیجه برای استفاده از RVA ها ابتدا باید آنها را به File Offset تبدیل کنیم .

به این منظور تابعی را با نام RVAToOffset ایجاد کرده ایم . جزئیات این تابع را بررسی نخواهیم کرد ولی به طور خلاصه این تابع با توجه به RVA شروع و خاتمه Section ها و مقدار عضو PointerToRawData از رکورد IMAGE_SECTION_HEADER ، RVA را به File Offset تبدیل می کند . این تابع دو مقدار ورودی دارد که عبارتند از اشاره گر به فایل نگاشت شده و RVA مورد نظر . خروجی تابع ، File Offset را مشخص خواهد کرد که در eax قرار می‌گیرد.

```
assume edi:ptr IMAGE_IMPORT_DESCRIPTOR
.while !([edi].OriginalFirstThunk==0 && \
         [edi].TimeDateStamp==0 && \
         [edi].ForwarderChain==0 && \
         [edi].Name1==0 && \
         [edi].FirstThunk==0)
```

اکنون edi به اولین رکورد IMAGE_IMPORT_DESCRIPTOR اشاره می کند . حال عناصر آرایه را تا رسیدن به رکورد صفر بررسی می کنیم .

```
invoke AppendText,hDlg,addr ImportDescriptor
invoke RVAToOffset,pMapping, [edi].Name1
mov edx,eax
add edx,pMapping
```

می خواهیم مقادیر رکورد IMAGE_IMPORT_DESCRIPTOR را در کنترل Edit نمایش دهیم . Name1 با بقیه اعضاء فرق دارد زیرا حاوی RVA نام فایل dll است . پس باید آنرا نیز به File Offset تبدیل کنیم .

```
invoke wsprintf, addr temp,\
                 addr IDTemplate, [edi].OriginalFirstThunk,\
                 [edi].TimeDateStamp,\
                 [edi].ForwarderChain,\
                 edx,[edi].FirstThunk
invoke AppendText,hDlg,addr temp
```

مقدار فعلی IMAGE_IMPORT_DESCRIPTOR را نمایش می دهیم .

```
.if [edi].OriginalFirstThunk==0
   mov esi,[edi].FirstThunk
.else
   mov esi,[edi].OriginalFirstThunk
.endif
```

حال برای حرکت بر روی آرایه IMAGE_THUNK_DATA آماده می شویم . به طور معمول باید از آرایه OriginalFirstThunk استفاده کنیم ولی به دلیل اینکه برخی از Linker ها از مقدار صفر برای OriginalFirstThunk استفاده می کنند ، ابتدا باید مقدار آنرا بررسی کرده و در صورتی که مخالف صفر باشد از آن استفاده می کنیم . در غیر اینصورت باید از آرایه First Thunk استفاده کنیم .

```
invoke RVAToOffset,pMapping,esi
add eax,pMapping
mov esi,eax
```

همان‌طور که می دانید مقادیر OriginalFirstThunk و First Thunk به صورت RVA هستند و باید آنها را به File Offset تبدیل کنیم .

```
invoke AppendText,hDlg,addr NameHeader
.while dword ptr [esi]!=0
```

حال برای بررسی آرایه IMAGE_THUNK_DATA آماده هستیم و می توانیم عناصر این آرایه را بررسی کرده و لیستی از فایل های dll مورد استفاده را تهیه کنیم .

```
test dword ptr [esi],IMAGE_ORDINAL_FLAG32
jnz ImportByOrdinal
```

اولین کاری که باید انجام دهیم اینست که مقدار IMAGE_THUNK_DATA را با IMAGE_ORDINAL_FLAG32 مقایسه کنیم . این تست در حقیقت مقدار بیت پر ارزش IMAGE_THUNK_DATA را چک می کند . در صورتیکه مقدار این بیت برابر با یک باشد ، نتیجه می گیریم که تابع مورد نظر توسط شماره اش شناسایی می شود و می توانیم این شماره را از قسمت Low word بدست آوریم .

```
invoke RVAToOffset,pMapping,dword ptr [esi]
mov edx,eax
```

```
        add edx,pMapping
        assume edx:ptr IMAGE_IMPORT_BY_NAME
```

در صورتی که مقدار بیت پر ارزش IMAGE_THUNK_DATA برابر صفر باشد ، این مقدار حاوی RVA مربوط به رکورد IMAGE_IMPORT_BY_NAME است که باید آنرا به File Offset تبدیل کنیم .

```
        mov cx, [edx].Hint
        movzx ecx,cx
        invoke wsprintf,addr temp,\
                        addr NameTemplate,\
                        ecx,\
                        addr [edx].Name1

        jmp ShowTheText
```

Hint از نوع Word است که برای فرستادن آن به تابع wsprintf ، باید آنرا به نوع dword تبدیل کنیم . سپس Hint و نام تابع را در کنترل Edit چاپ می کنیم .

```
ImportByOrdinal:
        mov edx,dword ptr [esi]
        and edx,0FFFFh
        invoke wsprintf,addr temp,addr OrdinalTemplate,edx
```

در حالتی که تابع توسط شماره اش شناسایی می شود ، قسمت High word را صفر کرده و شماره را نمایش می دهیم .

```
ShowTheText:
        invoke AppendText,hDlg,addr temp
        add esi,4
```

پس از وارد کردن شماره و یا نام تابع در کنترل Edit ، به سراغ IMAGE_THUNK_DATA بعدی می رویم.

```
    .endw
    add edi,sizeof IMAGE_IMPORT_DESCRIPTOR
```

وقتی که تمام IMAGE_THUNK_DATA های آرایه را بررسی کردیم، به سراغ IMAGE_IMPORT_DESCRIPTOR بعدی رفته و توابع ورودی از dll بعدی را مورد بررسی قرار می دهیم. توجه داشته باشید که معمولا" فایلهای اجرایی از چندین dll استفاده می کنند .

ساختار فایلهای اجرایی (بخش۷) Export Table

در فصل قبل مطالبی را در رابطه با Import Table آموختیم . حال جدول توابع صادر شده از dllها (Export Table) را مورد بررسی قرار خواهیم داد.

همانطور که ذکر شد ، هنگامی‌که بارگذار PE برنامه‌ای را اجرا می‌کند dll های مورد نیاز آن برنامه را نیز باگذاری کرده و در فضای آدرس آن قرار می‌دهد. سپس اطلاعات مربوط به توابع ورودی را از برنامه اصلی بازخوانی کرده و با استفاده از آنها به جستجوی آدرس توابع مورد نیاز برنامه در فایل dll می‌پردازد. محلی که بارگذار PE برای جستجوی توابع به آنجا رجوع می‌کند، Export Table نام دارد. این بخش جزئی ازساختار کنترلی فایل‌های PE محسوب می‌شود. یک فایل PE می‌تواند توابع داخلی خود را برای استفاده سایر فایل‌های PE به دو روش در جدول توابع خروجی (Export Table) قرار دهد. با استفاده از نام و یا شماره .

این شماره در حقیقت یک عدد ١٦ بیتی است که به صورت یکتا یک تابع را در یک فایل dll خاص مشخص می‌کند. استفاده از این روش غیر قابل اطمینان و خطرناک است زیرا در نگهداری dll ها مشکلاتی را به وجود خواهد آورد. در صورت تغییر این فایل dll و تغییر شماره‌های توابع ، برنامه نویسانی که ازاین فایل استفاده می‌کنند نمی‌توانند شماره توابعی را که در برنامه‌های خود از انها استفاده کرده‌اند عوض کنند و برنامه‌های آنان با مشکل مواجهه خواهد شد.

حال می‌خواهیم به بررسی رکورد Export بپردازیم . همانند جدول ورودی ، با نگاهی به Data Directory می‌توانید به محل قرار گرفتن Export Table پی ببرید. Export Table اولین عضو Data Directory می باشد. رکورد Export با نام IMAGE_EXPORT_DIRECTORY شناخته می‌شود. این رکورد ١١ عضو داد اما فقط بعضی از آنها برای ما مهم هستند.

nName : نام واقعی ماژول است . این فیلد بسیار ضروری است چون نام فایل می‌تواند تغییر کند و در این حالت بارگذار PE از این نام داخلی استفاده خواهد کرد.

NumberOfFunctions : تعداد کل توابعی که توسط این فایل صادر می‌شوند.

NumberOfNames : تعداد توابعی که با استفاده از نام صادر می‌شوند. این مقدار تعداد کل توابع موجود در فایل نیست. برای پی بردن به تعداد کل توابع موجود باید مقدار NumberOfFunctions را چک کنند. توجه داشته باشید که در صورتی که هیچ تابعی برای صدور مشخص نشده باشد، آدرس مربوط به Export Table در Data Directory صفر خواهد بود.

AddressOfFunctions : یک RVA است که به آدرس شروع آرایه‌ای از RVA های توابع داخلی صادر شده از طرف فایل PE اشاره می‌کند.

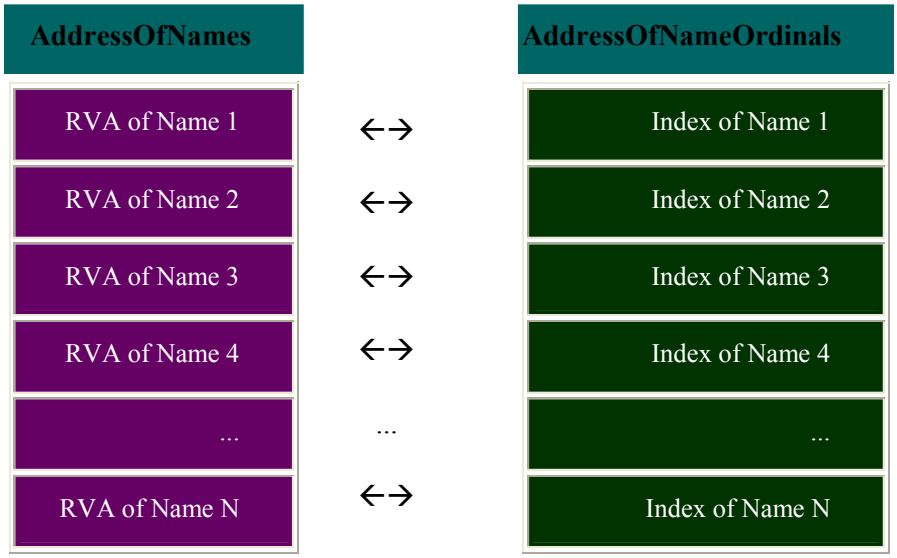
AddressOfNames : اشاره‌گر به آرایه‌ای است که حاوی آدرس رشته‌های کاراکتری مربوط به نام توابع صادر شده می‌باشد.

AddressOfNameOrdinals : اشاره‌گر به آرایه‌ای است که حاوی شماره‌های مربوط به نام توابع در آرایه بالا است .

خواندن مطالب بالا ممکن است نتواند تصویر واقعی Export Table را در ذهن شما ایجاد کند. توضیحات ساده شده زیر مفهوم را روشن‌تر خواهد نمود.

Export Table برای استفاده توسط بارگذار PE ایجاد شده است .یک ماژول باید آدرس توابع صادراتی خود را در جایی نگه‌داری کند تا بارگذار PE بتواند از آن به سادگی استفاده کند. این عمل به وسیله آرایه‌ای انجام می‌شود که فیلد AddressOfFunctions به آن اشاره می‌کند. تعداد اعضای این آرایه در عضو NumberOfFunctions نگه‌داری می‌شود. پس اگر یک ماژول ٤٠ تابع خود را صادر کند باید در آرایه‌ای که توسط AddressOfFunctions به آن اشاره می‌شود، ٤٠ عضو وجود داشته باشد. NumberOfFunctions حاوی مقدار ٤٠ باشد. حال اگر بعضی از توابع با نام خود فرستاده می‌شوند ، فایل PE باید نام آنها را نیز نگه‌داری کند. فایل PE ، RVA نام آنها را در یک آرایه نگه‌داری می‌کند و بارگذار PE به راحتی از آن آرایه استفاده می‌کند. همان طور که ذکر شد ، فیلد AddressOfNames حاوی آدرس شروع این آرایه بوده و تعداد آنها در NumberOfNames نگه‌داری می‌شود. حال به نحوه عملکرد بارگذار PE فکر کنید. بارگذار PE تا اینجا نام توابع را دارد و باید به طریقی آدرس این توابع را به دست آورد. فایل PE دارای دو آرایه می‌باشد یکی برای نام‌ها و دیگری برای آدرس‌ها، اما هیچ ارتباطی بین این دو آرایه وجود ندارد. پس به چیزی نیاز داریم که ارتباطی بین نام توابع و آدرس آنها را ایجاد کند. باگذار PE به این منظور از آرایه اندیس‌ها استفاده می‌کند که آدرس شروع آن در AddressOfNameOrdinals قراردارد. به عنوان مثال اگر نام یک تابع صادراتی با نام Test دراندیس ۱۳ از آرایه نام‌ها (AddressOfNames) واقع شده باشد ، آنگاه عضو سوم از آرایه AddressOfOrdinals حاوی اندیس آدرس تابع در آرایه AddressOf Names خواهد بود.

توجه داشته باشید که هر نام فقط و فقط یک آدرس می‌تواند داشته باشد اما عکس این مطلب درست نیست و یک آدرس می‌تواند با چند نام صادر شود. به بیان دیگر برای ارجاع به یک تابع می‌توانیم نام‌های مستعار داشته باشیم.



اگر نام یک تابع صادراتی را داشته باشیم باید بتوانیم آدرس آنرا نیز پیدا کنیم. برای این کار گام‌های زیر را باید به ترتیب انجام دهیم .

۱- رفتن به PE Header.

۲- خواندن Virtual Address مربوط به Export Table از Data Directory.

۳- رفتن به Export Table و به دست آوردن NumberOfNames.

٤- حرکت موازی بر روی دو آرایه AddressOf Names و AddressOfNamesOrdinals. اگر نام مورد نظر پیدا شد، باید مقدار عنصر متناظر با این عنصر را نیز از آرایه AddressOfNameOrdinals بدست آورید. برای مثال اگر RVA نام مورد نظر را در عضو هفتاد و هفتم آرایه AddressOfNames به دست آوردید ، باید مقدار موجود در عنصر هفتاد و هفتم آرایه AddressOf Ordinals را نیز بخوانید.

٥- از مقدار موجود در آرایه AddressOfNameOrdinals به عنوان یک اندیس به آرایه AddressOfFunctions استفاده کنید. برای مثال اگر مقدار موجود آن عنصر از AddressOfNameOrdinals برابر ٥ بود، باید پنجمین عنصر از آریه AddressOfFunctions را بخوانید که حاوی RVA تابع مورد نظر شما می‌باشد.

فصل نهم

برنامه‌نویسی و ایجاد درایورها در ویندوزهای خانواده NT

فصل نهم

برنامه‌نویسی و ایجاد درایورها در ویندوزهای خانواده NT

مروری بر معماری ویندوز

اجزای اصلی سیستم

همان‌طور که می‌دانید، حافظهٔ ٤ گیگا بایتی هر Process در معماری ۳۲ بیتی به دو بخش مساوی تقسیم می‌شود. دو گیگا بایت پایین حافظه فضای آدرس User است یعنی از آدرس‌های 00000000 تا 7FFFFFF. دو گیگا بایت دیگر برای اجزاء سیستم، ساختمان داده‌ها، ساختارهای کنترلی و.... به کار می‌رود.

فرآیندهای سطح User به چند دسته تقسیم‌بندی می‌شوند که در ادامه به طور خلاصه به بررسی آنها خواهیم پرداخت.

۱- **System Support Processes**: برای مثال فرآیند Logon که در شاخهٔ System 32 و به نام Winlogon.exe می‌باشد.

۲- **Service Process**: برای مثال Spool Service که در شاخهٔ System32 در شاخهٔ Windows و با نام فایل Spoolsv.exe می‌باشد.

۳- **User Application**: می‌تواند یکی از ٥ نوع زیر باشد:

۱) Win 32 ۲) Win 3.1 ۳) MS-Dos ٤) Posix ٥) OS/2 باشد.

۴- **Environment Subsystems** : ویندوز با این سه Environment Subsystems حرکت می‌کند. هر سه اینها در شاخهٔ System32 می‌باشند. برای Win 32 با نام Csrss.exe برای

Posix با نام Psxss.exe و برای OS/2 با نام OS2ss.exe می‌باشند. در ویندوز XP و بعد از آن Posix و OS/2 برداشته شده است.

اجراء Kernel-mode در زیر آمده:

Executive : مدیریت حافظه، مدیریت فرآیندها و Thread ها، امنیت و .... را برعهده دارد.

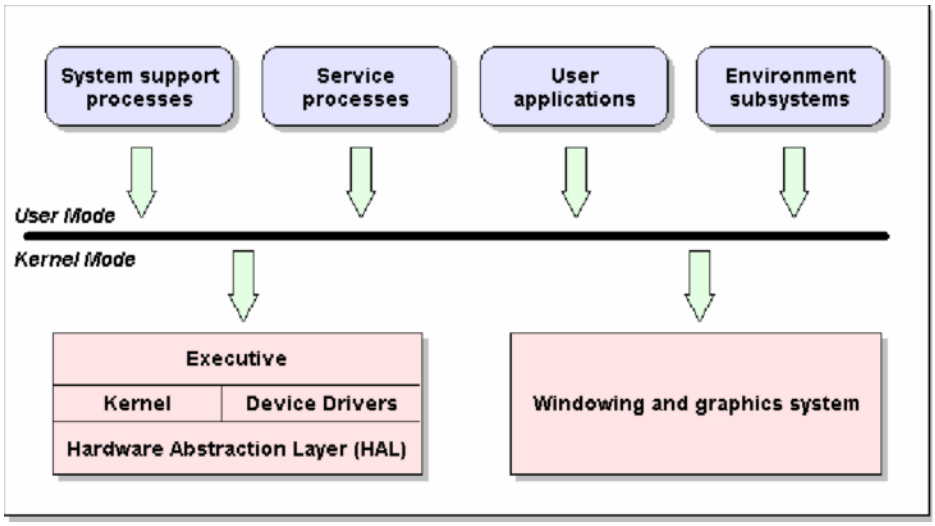
Kernel : زمان بندی Thread ، وقفه‌ها و فرستادن خطاها را برعهده دارد. این فایل در شاخهٔ System32 با نام Ntoskrnl.exe قرار دارد.

Device Driver : درایورهای قطعات سخت‌افزاری، درایورهای شبکه و سیستم‌های فایل می‌باشد.

(HAL) Hardware Abstraction Layer : این بخش، سه بخش قبلی را از نوع سخت‌افزار مجزا کرده و در شاخهٔ System32 با نام hal.dll قرار دارد.

Windows and Graphic System : کنترل‌های واسط کاربر گرافیکی را ارائه می‌کند. این فایل در شاخهٔ System32 و با نام Win32k.sys قرار دارد.

کلیه مطالب بیان شده را در شکل زیر مشاهده می‌کنید.



مقایسهٔ مد کاربر یا مد هسته

معماری پردازنده‌های اینتل X86 ، ٤ سطح دسترسی را تعریف می‌کند. ویندوز از سطح صفر (Ring 0) برای مد هسته و از سطح سوم آن برای مد کاربر استفاده می‌کند. علت اینکه ویندوز از دو سطح استفاده می‌کند این است که بعضی از سخت‌افزارهایی که پشتیبانی می‌شدند مثل Compaq Alpha دو سطح دسترسی را در نظر می‌گرفتند. هر فرآیند که در مد کاربر قرار دارد فضای آدرس مخصوص به خود دارد. صف‌های دستورات مربوط به این فرآیندها در پایین‌ترین سطح دسترسی اجرا می‌شوند (Ring 3) و نمی‌توانند دستورات ممتاز CPU را اجرا کنند. این صف‌ها، دسترسی محدود و غیرمستقیم به داده‌ها و فضای آدرس‌دهی سیستم دارند. آنها نمی‌توانند دسترسی مستقیم به تجهیزات سخت‌افزاری داشته باشند. برای مثال اگر یک فرآیند قصد دسترسی به یک آدرس از نیمه بالای فضای ٤ گیگابایتی حافظه خود را داشته باشد، سیستم سریعاً آن را متوقف خواهد کرد.

فرآیندهای مد کاربر به منظور پایداری سیستم بسیار خطرناک درنظر گرفته شده‌اند و حقوق آنها کاملاً محدود است و هر تلاشی که برای شکستن این محدودیت‌ها صورت بگیرد، به سرعت متوقف شده و فرآیند مربوطه خاتمه داده می‌شود.

فرآیندهای مد هسته فضای حافظهٔ محافظت شدهٔ مد هسته را به اشتراک می‌گذراند و در سطح دسترسی ممتاز اجرا می‌شوند که (Ring 0) هم نامیده می‌شود. این فرآیندها توانایی اجرای تمامی دستورات CPU را دارند. همچنین دسترسی نامحدود به داده‌ها و ساختارهای سیستمی و کدها و منابع سخت‌افزاری دارند.

کدهایی که در فضای آدرس سیستم اجرا می‌شوند به‌صورت مطمئن در نظر گرفته و بارگذاری شده و به‌عنوان یک بخش از هستهٔ سیستم نظر گرفته می‌شوند. Driver ها به عنوان یک ابزار مطمئن از Kernel یا هسته در نظر گرفته می‌شود و قدرت نامحدودی برای اجرای تمام کارهایی که می‌خواهند انجام دهند دارند. دسترسی نامحدود درایورهای مد هسته می‌تواند برای انجام کارهای غیرممکن در برنامه‌های مد کاربر مورد استفاده قرار بگیرد. پس اگر طرحی برای دسترسی به توابع داخلی سیستم یا ساختمان داده‌های سیستم دارید، تنها راه این است که یک درایور مد هسته در فضای آدرس سیستم بارگذاری کنید. این کار نسبتاً ساده و قابل اطمینان است و توسط سیستم عامل پشتیبانی می‌شود.

Device Driver ها در ویندوز های خانواده NT

ویندوز NT سطح گسترده‌ای از انواع درایورها را پشتیبانی می‌کند که آنها را می‌توان به گروه‌های زیر تقسیم کرد:

درایورهای سطح کاربر

Device Drivers های مجازی (VDD)

یک ابزار مد کاربر است که برای شبیه‌سازی برنامه‌های ١٦ بیتی MS-Dos مورد استفاده قرار می‌گیرد. اگر چه این نوع از درایورها از پسوند VXD استفاده می‌کنند اما ساختاری کاملاً متفاوت با درایورهای Windows95/98 دارند.

درایورهای چاپگر:

درخواست‌های مستقل از ابزار گرافیکی را به دستورات Printer ترجمه می‌کنند.

درایور های مد هسته

File System Driver

استانداردهای گوناگون سیستم‌های فایل ارائه می‌کنند.

Legacy Drives

یک قطعه سخت‌افزاری را کنترل می‌کنند. این Driverها برای نسخه‌های اولیه Windows NT نوشته شده بوند اما اکنون نیز بدون تغییر در Windows NT/2000/2003 قابل اجرا هستند.

Streaming Drivers

برای پشتیبانی قطعات چندرسانه‌ای مثل کارت صدا می‌باشد.

از نام Device Driver معلوم است که کدی می‌باشد که می‌خواهد یک قطعه را کنترل کند. البته اجباری نیست که قطعه موردنظر یک قطعه سخت‌افزاری باشد و می‌تواند به‌صورت مجازی ایجاد شده و مدیریت شود. از نظر ساختاری یک Device Driver چیزی جز یک فایل با فرمت PE نیست، دقیقاً مثل سایر فایل‌های exe یا dll . درایورهای قطعات (Device Driver) می‌توانند با پسوند sys در مد هسته بارگذاری شوند و تنها تفاوت آنها این است که مدیریت و کنترل آنها با سایر فایل‌های اجرایی کاملاً متفاوت است.

البته تفاوت در اینجاست که ما نمی‌توانیم دسترسی مستقیم به Device Driver و اطلاعات آن را داشته باشیم. تنها راه ممکن به منظور ارتباط با آنها، I/O manager است. I/O manager یک محیط اولیه برای مدیریت درایورها فراهم می‌آورد.

سطوح درخواست وقفه Interrupt Request Level (IRQL)

وقفه‌ها یکی از اجزای اساسی هر سیستم عامل هستند. آنها پردازنده را متوجه یک جریان خارجی می‌کنند. وقفه‌ها به دو نوع سخت‌افزاری و نرم‌افزاری تقسیم‌بندی می‌شوند. برای وقفه‌ها اولویت‌های خاصی در نظر گرفته می‌شود به عنوان مثال یک وقفه با اولویت بالاتر حق به خدمت گرفتن یک وقفه با اولویت پایین‌تر را دارد.

ویندوز طرح اولویت وقفه‌ها را با نام Interrupt Request Level می‌شناسد هسته اولویت وقفه‌ها را از صفر تا ۳۱ نشان می‌دهد. اعداد بزرگتر نشان‌دهندهٔ وقفه با اولویت بالاتر هستند. این اعداد به اختصار IRQL نامیده می‌شوند.

ابزارهای مورد نیاز برای ایجاد درایورها در Macro Assembler

Kmdkit شامل تمام ابزارهای موردنیاز برای برنامه‌نویسی درایورهای مد هسته با استفاده از Masm است.

این ابزار در CD ضمیمه موجود می‌باشد

Tools\Kmdkit

سرویس‌ها (Services)

ممکن است تعجب کنید که چگونه سرویس‌های مد کاربر به درایورهای مد هسته بستگی دارند. قبل از اینکه بتوانیم با یک Device Driver ارتباط برقرار کنیم ، ابتدا باید آنرا نصب کنیم و سپس آن را اجرا کنیم. در این قسمت به بررسی سرویس‌ها و چگونگی ارتباطات آنها می‌پردازیم.

Services

سرویس‌ها برنامه‌هایی هستند که به منظور ارائه خدمات به نرم‌افزارهای دیگر در نظر گرفته شده‌اند. اکثر Serviceها دارای واسط کاربر (User Interface) می‌باشند. سرویس‌ها هم می‌توانند به‌صورت دستی (Manual) و یا در هنگام شروع (Startup) اجرا شوند.

از این نظر Device Driverها بسیار شبیه سرویس‌ها هستند. مایکروسافت به دلایل نامعلومی سرویس‌های مد کاربر و درایورهای مد هسته را به روش یکسانی مدیریت می‌کنند. در این مطلب عناوین Service و Driver را معادل در نظر می‌گیریم. در این بخش توابع زیادی مورد بررسی قرار می‌گیرد که هم برای سرویس‌ها در نظر گرفته می‌شود و هم برای Device Driver ها. همان‌طور که ذکر شد در این قسمت فقط تأکید روی Device Driver ها است. سه نوع ابزار به‌منظور به کار انداختن سرویس‌های Windows NT وجود دارد.

١- **Service Control Manager (SCM)**: SCM مسئول اجرای سرویس و برقراری ارتباط با آن است.

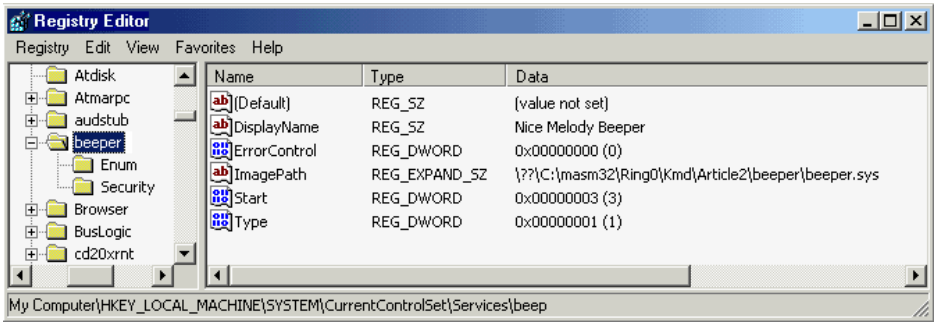
٢- **Service Control Program** : SCP برای برقراری ارتباط با SCM به کار می‌رود و اینکه اعلام کند چه زمانی سرویس شروع شده و چه زمانی پایان یابد.

٣- **Service Program** : شامل کدهای اجرایی است.

(SCM) Service Control Manager

SCM در آدرس \%Systemroot%\System32\Services.exe قرار دارد که فرآیند Win logon در هنگام بوت شدن سیستم، آنرا اجرا می‌کند. بعد محتویات بیشتری که در قسمت HKLM\SYSTEM\CurrentControlSet\Services\ قرار دارد را اسکن می‌کند و بازای هر کلید داده‌های مربوطه را در پایگاه داده سرویس‌ها قرار می‌دهد. یک ورودی در پایگاه داده مربوطه به سرویس‌ها شامل تمامی پارامترهای مربوط به یک سرویس می‌باشد. برای بدست آوردن اطلاعات بیشتر درباره آن می‌توانیدRegistry Editor را باز کرده (%SystemRoot%\regedit.exe) و به قسمت زیر مراجعه کنید HKLM\SYSTEM\CurrentControlSet\Services\. محتویات این قسمت را باز کنید. برای مشاهده لیست سرویس‌های نصب شده قسمت Administrative Tools از Control Panel را انتخاب کنید و از آنجا وارد بخش Services شوید.

حال می‌خواهیم کمترین پارامترهای لازم برای نصب یک Device Driver را مورد بررسی قرار دهیم. همان‌طور که در مثال می‌بینید یک درایور با نام beeper.sys را در نظر گرفته‌ایم.



DisplayName : نام سرویسی که توسط برنامهٔ واسط کاربر استفاده می‌شود. اگر نامی اختصاصی داده نشود همان نام سرویس که در کلید رجیستری قرار دارد. به عنوان نام آن در نظر گرفته خواهد شد.

Error Control : اگر یک درایور در مراحل بارگذاری و شروع با مشکل مواجه شود، SCM باتوجه به مقادیر این قسمت، نسبت به خطای بوجود آمده عکس‌العمل نشان خواهد داد. دو مقدار برای ما مورد نظر است:

- SERVICE_ERROR_IGNORE (0) : مدیر I/O خطایی اتفاق افتاده را نادیده گرفته و روند اجرایی خود را ادامه دهد.
- SERVICE_ERROR_NORMAL (1) : اگر درایور برای بارشدن یا معرفی به مشکلی برخورد یا با شکست مواجه شود، SCM باید یک اخطار را به کاربر نمایش داده و یک رویداد خطا در ثبت‌کننده رویدادهای سیستم (System Event Log) بنویسد..

می‌توانید توضیحات مربوط به یک رویداد را با انتخاب AdministrativeTools>EventViewer و Double-Click کردن بر روی رویداد مورد نظر مشاهده کنید. به عنوان مثال درایور Beeper تمام عملیات مورد نیاز خود را در مرحله ابتدایی انجام داده و سپس یک شماره خطا برای از بین رفتن از حافظه برمی‌گرداند چون نمی‌تواند کار خاصی را انجام دهد. مقدار Error Control برای درایور مربوط به Beeper برابر SERVICE_ERROR_IGNORE است پس هیچ ثبتی انجام نمی‌شود.

Image Path : مسیر مناسب برای فایل درایور را مشخص می‌کند وقتی که این مقدار مشخص نشده باشدI/O Manager در مسیر %SystemRoot%\Driver\ به دنبال درایور می‌گردد.

Start : مشخص می‌کند که چه زمانی عملیات شروع و بارگذاری انجام شود. دو مقدار برای این پارامتر وجود دارد که بررسی می‌کنیم.

SERVICE_AUTO_START (2): درایور در طول مراحل Boot اجرا می‌شود. چنین درایورهایی سرویس‌های Auto_Start نامیده می‌شوند. اگر یک درایور به درایورهای دیگری وابسته باشد, SCM آن درایور را نیز اجرا خواهد گرد.

(3) SERVICE_DEMAND_STAR۴ T: درایور توسط SCM و در پاسخ به دستور کاربر اجرا می‌شود.

Type : نوع سرویس را مشخص می‌کند چون ما فقط با درایورهای ابزار سروکار داریم، تنها مقداری که ما می‌توانیم استفاده کنیم (1) SERVICE_KERNEL_DRIVER است.

اگر بخواهیم درایوری را شروع کنیم که در پایگاه داده SCM تعریف نشده است، در هر زمان می‌توانیم با کمک SCP (Service Control Program) این کار را انجام دهیم.

Service Control Program

همان‌طور که از نام آن پیداست، از این قسمت به‌منظور کنترل و مدیریت سرویس‌ها و درایورها استفاده می‌شود. این قسمت کلیه عملیات خود را تحت نظارت SCM انجام داده و توابع مناسب را فراخوانی خواهد کرد. این قسمت کد مربوط به SCP است که درایور Beeper.sys را کنترل خواهد کرد.

کدها و فایل‌های مربوط به این قسمت در CD ضمیمه موجود می‌باشد

SourceCodes\NTDriver

```
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;

;  Service Control Program for beeper driver

;

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

.386
.model flat, stdcall
option casemap:none

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;                              I N C L U D E   F I L E S

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

include \masm32\include\windows.inc
```

```
include \masm32\include\kernel32.inc
include \masm32\include\user32.inc
include \masm32\include\advapi32.inc

includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\user32.lib
includelib \masm32\lib\advapi32.lib

include \masm32\Macros\StRings.mac

;=====================================================================
;                                    C O D E

;=====================================================================

.code

start proc

local hSCManager:HANDLE
local hService:HANDLE
local acDriverPath[MAX_PATH]:CHAR

    invoke OpenSCManager, NULL, NULL, SC_MANAGER_CREATE_SERVICE
    .if eax != NULL
        mov hSCManager, eax

        push eax
        invoke GetFullPathName, $CTA0("beeper.sys"),\
                                sizeof acDriverPath,\
                                addr acDriverPath,\
                                esp
        pop eax

        invoke CreateService, hSCManager,\
                              $CTA0("beeper"),\
                              $CTA0("Nice Melody Beeper"), \
                              SERVICE_START + DELETE,\
                              SERVICE_KERNEL_DRIVER,\
                              SERVICE_DEMAND_START, \
                              SERVICE_ERROR_IGNORE,\
                              addr acDriverPath,\
                              NULL,\
                              NULL,\
                              NULL,\
                              NULL,\
                              NULL
        .if eax != NULL
            mov hService, eax
            invoke StartService, hService, 0, NULL
            invoke DeleteService, hService
            invoke CloseServiceHandle, hService
```

```
        .else
            invoke MessageBox, NULL,\
 $CTA0("Can't register driver."),\
 NULL,\
 MB_ICONSTOP
        .endif
        invoke CloseServiceHandle, hSCManager
    .else
        invoke MessageBox, NULL,\
 $CTA0("Can't connect to Service Control Manager."),\
 NULL,\
 MB_ICONSTOP
    .endif

    invoke ExitProcess, 0

start endp

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
:
;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
:

end start
```

برقراری ارتباط با SCM

اولین کاری که باید انجام دهیم این است که برای برقراری ارتباط با SCM تابع OpenSCManager را به منظور باز کردن سرویس مورد نظر بر روی کامپیوتر فعلی فراخوانی کنیم.

```
OpenSCManager proto lpMachineName:LPSTR,\
                    lpDatabaseName:LPSTR,\
                    dwDesiredAccess:DWORD
```

lpMachineName : به یک رشتهٔ مختوم به صفر اشاره می‌کند که نام کامپیوتر مقصد یا هدف را مشخص می‌کند. اگر این مقدار برابر Null باشد، یا به یک رشتهٔ خالی اشاره کند، تابع با کامپیوتر محلی ارتباط برقرار خواهد کرد.

lpDatabaseName : به رشتهٔ مختوم به صفری اشاره می‌کند که حاوی نام پایگاه داده SCM است. اگر این رشته Null باشد، پایگاه داده ServiceActive به‌صورت پیش‌فرض باز می‌شود.

```
.const
szActiveDatabase db "ServicesActive", 0
SERVICES_ACTIVE_DATABASE equ offset szActiveDatabase
```

dwDesiredAccess : حقوق دسترسی به SCM را مشخص می‌کند. این پارامتر می‌تواند مقادیر زیر را اختیار کند.

SC_MANAGER_CONNECT : برقراری ارتباط را ممکن می‌سازد. این نوع دسترسی به سادگی با فرستادن مقدار صفر مشخص می‌شود.

SC_MANAGER_CREATE_SERVICE : امکان فراخوانی تابع CreateService برای ایجاد یک شی Service و اضافه شدن آن به پایگاه داده را فراهم می‌سازد.

SC_MANAGER_ALL_ACCESS : دسترسی کامل به پایگاه داده را ایجاد می‌کند.

حال بدین صورت با SCM ارتباط برقرار می‌کنیم.

```
    invoke OpenSCManager, NULL, NULL, SC_MANAGER_CREATE_SERVICE
    .if eax != NULL
        mov hSCManager, eax
```

اگر تابع OpenSCManager عملیات خود را با موفقیت انجام دهد، مقدار بازگشتی ، شماره دسترسی به پایگاه داده SCM می‌باشد. با فرستادن این مقدار به توابع دیگر می‌توانیم در پایگاه داده SCM تغییراتی را ایجاد کنیم.

نصب یک درایور جدید نیازمند دسترسی Administrator است. کاربران معمولی نمی‌توانند کدهای مربوطه را اجرا کنند. در این جا فرض برا این است که شما دسترسی مناسب را دارید.

نصب درایور جدید

وقتی SCM باز می‌شود، می‌توانیم Driver خود را به پایگاه داده آن اضافه کنیم. این کار را توسط تابع CreateService انجام می‌دهیم. در این جا پارامترهای آن را بررسی می‌کنیم. این تابع ۱۳ پارامتر دارد که تقریباً ساده هستند.

hSCManager : شماره دسترسی پایگاه داده SCM.

lpServiceName : به یک رشتهٔ مختوم به صفر اشاره می‌کند. این رشته حاوی اسم Service است که قرار است نصب شود. بیشترین طول ممکن ٢٥٦ کاراکتر است. کاراکترهای (/) و (\) کاراکترهای نامعتبر در اسم Service می‌باشند. این رشته مربوط به نام سرویس در زیر کلید رجیستری می‌باشد.

lpDisplayName : به یک رشتهٔ مختوم به صفر اشاره می‌کند که توسط برنامهٔ واسط کاربر برای مشخص کردن سرویس استفاده می‌شود. ماکزیمم طول این رشته ٢٥٦ کاراکتر است. این رشته مربوط به مقدار DisplayValue در زیر کلید رجیستری می‌باشد.

dwDesiredAccess : نوع دسترسی به سرویس را مشخص می‌کند. مقادیر مجاز به این صورت هستند.

SERVICE_ALL_ACCESS : دسترسی کامل به سرویس.

SERVICE_START : توانایی فراخوانی تابع Start Service برای شروع سرویس.

SERVICE_STOP : توانایی فراخوانی تابع Control Service برای توقف سرویس.

DELETE : توانایی فراخوانی تابع Delete Service برای حذف سرویس.

ما فقط به دو چیز نیاز داریم. اول اینکه سرویس را شروع کنیم و دوم اینکه آن را از پایگاه داده SCM حذف کنیم. بنابراین از دو مقدار SERVICE_START و DELETE در این پارامتر استفاده می‌کنیم.

dwServiceType :نوع‌سرویس‌رامشخص می‌کند. ما فقط از SERVICE_KERNEL_DRIVER استفاده می‌کنیم. این مقدار نیز مربوط به مقدار Type در زیر کلید رجیستری می‌باشد.

dwStartType : مشخص می‌کند که چه زمانی باید سرویس شروع شود. اگر می‌خواهیم خودمان سرویس را شروع کنیم مقدار SERVICE_DEMAND_START را می‌فرستیم. اگر Driver باید دقیقاً بعد از راه‌اندازی سیستم، قبل از ظهور Log On Prompt شروع شود، از مقدار SERVICE_AUTO_START استفاده می‌کنیم. این مقدار نیز مربوط به مقدار Start در زیر کلید رجیستری است.

dwErrorControl : نحوهٔ بررسی خطا را در زمان شروع سرویس مشخص می‌کند. به‌منظور نادیده گرفتن خطاها از مقدار SERVICE_ERROR_IGNORE استفاده می‌کنیم و برای ثبت خطاهای ممکن، از مقدار SERVICE_ERROR_NORMAL استفاده می‌کنیم. این مقدار نیز مربوط به مقدار Error Control در زیر کلید رجیستری می‌باشد.

lpBinaryPathName : به یک رشتهٔ مختوم به صفر اشاره می‌کند که حاوی مسیر فایل باینری درایور است. این مقدار هم مربوط به مقدار زیر کلید Image Path در رجیستری می‌باشد.

lpLoadOrderGroup : به یک رشتهٔ مختوم به صفر اشاره می‌کند. این رشته نام گروهی که سرویس در آن عضو است را مشخص می‌کند. درایورها به هیچ گروه خاصی اختصاص ندارد. به این دلیل مقدار NULL را به آن اختصاص می‌دهیم.

lpdwTagId : به یک متغیر ۳۲ بیتی اشاره می‌کند. این متغیر شناسهٔ یکتایی است که به سرویس در گروه اختصاص می‌یابد. درنتیجه این پارامتر را نیز Null قرار می‌دهیم.

lpDependencies : این پارامتر برای درایورها معنای خاصی ندارد و آن را همیشه Null قرار می‌دهیم.

lpServiceStartName : به یک رشتهٔ مختوم به صفر اشاره می‌کند که حاوی نام Account است که سرویس باید در آن اجرا شود. اگر نوع سرویس SERVICE_KERNEL_DRIVER باشد ، نام مربوطه، همان نام شیی درایور است که سیستم برای بارگذاری آن استفاده می‌کند. در این جا مقدار Null را به آن اختصاص می‌دهیم تا درایور ما از نام شی پیش‌فرض که توسط زیر سیستم I/O ایجاد می‌شود استفاده کند.

lpPassword : Password برای درایورها نادیده گرفته می‌شود و همیشه باید Null باشد.

به‌منظور روشن‌تر شدن مطالب به جدول زیر توجه کنید:

CreateService	Registry
lpServiceName	Registry subkey name
lpDisplayName	DisplayName
dwServiceType	Type
dwStartType	Start
dwErrorControl	ErrorControl
lpBinaryPathName	ImagePath

```
push eax
invoke GetFullPathName, $CTA0("beeper.sys"),\
                        sizeof acDriverPath,\
                        addr acDriverPath,\
                        esp
pop eax

invoke CreateService, hSCManager,\
                      $CTA0("beeper"),\
                      $CTA0("Nice Melody Beeper"),\
                      SERVICE_START + DELETE,\
                      SERVICE_KERNEL_DRIVER,\
                      SERVICE_DEMAND_START,\
                      SERVICE_ERROR_IGNORE,\
                      addr acDriverPath,\
                      NULL,\
                      NULL,\
                      NULL,\
                      NULL,\
                      NULL
.if eax != NULL
    mov hService, eax
```

ابتدا تابع GetFullPathName را به‌منظور یافتن مسیر کامل فایل درایور و فرستادن آن به تابع CreateService فراخوانی می‌کنیم. تابع CreateService درایور ما را به پایگاه داده SCM اضافه می‌کند و زیر کلیدهای مناسب را در رجیستری ایجاد می‌نماید (به جدول قبل نگاه کنید).

فکر نکنید با استفاده از توابع API عمومی Regxxx می‌توانید با دستکاری رجیستری به این نتایج برسید. شاید بتوانید داده‌هایی را به رجیستری اضافه کنید، اما نمی‌توانید به پایگاه داده SCM چیزی را اضافه کنید.

اگر Device Driver موردنظر قبلاً در داده پایگاه SCM ایجاد شده باشد فراخوانی تابع Create Service با شکست مواجه خواهد شد. با فراخوانی تابع GetLastError خواهید دید که مقدار بازگشتی ERROR_SERVICE_EXISTS خواهد بود. اگر تابع Create Service بتواند کار خود را با موفقیت انجام دهد، مقدار بازگشتی شماره دسترسی به‌درایور خواهد بود. این شماره دسترسی برای دیگر توابع به‌منظور دسترسی به درایور مورد نیاز است.

شروع یک درایور

تابع بعدی StartService است که پیش تعریف آن به‌صورت زیر است:

```
StartService proto hService:HANDLE,\
                   dwNumServiceArgs:DWORD,\
                   lpServiceArgVectors:LPSTR
```

hService : که همان شمارهٔ دسترسی به درایور است.

dwNumServiceArgs : برای Device Driver ها این پارامترها صفر است

```
invoke StartService, hService, 0, NULL
```

با فراخوانی تابع StartService تصویر فایل درایور به فضای آدرس سیستم نگاشته می‌شود. معمولاً درایورها به آدرس مورد نظر خود در حافظه نگاشت می‌شوند و سیستم همانند فایل‌های PE عملیات بارگذاری و تصحیح آدرس‌ها و جدول ورودی را برای آنها انجام داده و سپس مدخل درایور را فراخوانی می‌کند. این مدخل در روال DriverEntry قرار دارد.

اگر معرفی درایور با موفقیت انجام پذیرد، روال DriverEntry مقدار STATUS_SUCCESS را باز می‌گرداند و تابع StartService مقدار غیرصفر باز خواهد گرداند. سپس StartService را فراخوانی می‌کنیم و درایور ملودی خود را پخش خواهد کرد.

حذف یک درایور

```
        invoke DeleteService, hService
        invoke CloseServiceHandle, hService
    .else
        invoke MessageBox, NULL,\
                           $CTA0("Can't register driver."),\
                           NULL,\
                           MB_ICONSTOP
    .endif
    invoke CloseServiceHandle, hSCManager
```

حال باید با فراخوانی تابع DeleteService ، درایور را از پایگاه داده SCM پاک کنیم. پیش تعریف این تابع به‌صورت زیر است:

```
DeleteService proto hService:HANDLE
```

hService : سرویسی که باید از بین برود را مشخص می‌کند البته باید اجازهٔ دسترسی مناسب برای انجام این کار را داشته باشید.

این تابع در واقع این سرویس را پاک نمی‌کند بلکه آنرا برای پاک شدن علامت‌گذاری می‌کند. SCM فقط زمانی آنرا پاک می‌کند که اجرای سرویس متوقف شده و تمام شماره‌های دسترسی به سرویس از بین بروند.

حال که نیازی به برقراری ارتباط با درایور نداریم، شماره دسترسی آنرا توسط تابع CloseServiceHandle از بین می‌بریم.

```
CloseServiceHandle proto hSCObject:HANDLE
```

hSCObject : شماره دسترسی به درایور یا پایگاه داده SCM . حال که هیچ شماره دسترسی به درایور وجود ندارد، از پایگاه داده SCM پاک خواهد شد. فراخوانی دوم یعنی فراخوانی تابع CloseServiceHandle شماره دسترسی به پایگاه داده SCM را از بین می‌برد.

ماکروهای رشته

$CTA0 یک ماکرو است که به شما اجازه تعریف رشته‌های مختوم به صفر را در بخش Read-Only داده‌ها می‌دهد.

ساخت چند درایور ساده

نحوه کامپایل و ساخت درایورهای Ring 0

به منظور سهولت عملیات کامپایل، معمولاً کدهای درایور را در یک فایل batch قرار می‌دهیم. این فایل مخلوطی از فایل‌های bat.* و asm.* است. مراحل زیر در هنگام اجرای فایل bat اتفاق می‌افتد.

```
;@echo off
;goto make


.386                        ; driver's code start

  ;:::::::::::::::::::::::::::::::
  ; the rest of the driver's code ;
  ;:::::::::::::::::::::::::::::::

end DriverEntry              ; driver's code end


:make

set drv=drvname

\masm32\bin\ml /nologo /c /coff %drv%.bat
\masm32\bin\link /nologo /driver /base:0x10000 /align:32
/out:%drv%.sys
  /subsystem:native %drv%.obj

del %drv%.obj

echo.
pause
```

دو دستور اول توسط کامپایلر Masm درنظر گرفته نمی‌شوند. اما به عنوان دستورات batch پذیرفته می‌شود. کنترل برنامه به make پرش می‌کند در این جا دستوراتی برای کامپایلر و لینکر وجود دارد. دستورات بین gotomake و make برای کامپایلر مفهوم هستند نه برای پردازندهٔ batch ولی بقیه دستورات برعکس فوق هستند یعنی برای پردازنده batch قابل فهم هستند ولی

کامپایلر آنها را در نظر نمی‌گیرد. این متد یک روش بسیار عالی است چون چگونگی کامپایل و لینک شدن در خود کد قرار دارد.

```
set drv=drvname
```

در این قسمت یک متغیر محیطی تعریف می‌کنیم که جانشینی برای نام فایل باشد.

گزینه‌های لینکر به شرح زیر می‌باشد:

/driver: به لینکر می‌گوید که باید یک درایور ایجاد کند.

/base:0x10000: آدرس پایه درایور را معادل 10000h قرار می‌دهد.

/out:%dvr%.sys: به لینکر اعلام می‌کند که به جای ساختن exe یا dll یک فایل sys ایجاد کند.

/subsystem:native: در header هر فایل PE قسمتی وجود دارد که به بارگذاری می‌گوید که چه زیر سیستمی مورد نیاز است: (Win 32 , Posix یا OS/2).

بسیار مهم است که محیط اجرایی مناسب فایل را تعیین کنیم. وقتی که یک فایل اجرایی را کامپایل می‌کنیم در این بخش زیر سیستم مورد نیاز خود را Win32 معرفی می‌کنیم اما درایورهای هسته نیازی به هیچ زیر سیستمی ندارند. آنها در محیط محلی (Native) اجرا می‌شوند.

یک درایور ساده

در این قسمت کد مربوط به ساده‌ترین Device Driver را مشاهده می‌کنید.

```
;@echo off
;goto make

;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;

; simplest - Simplest possible kernel-mode driver

;

;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

.386
.model flat, stdcall
option casemap:none
```

```
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;                                 I N C L U D E   F I L E S

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

include \masm32\include\w2k\ntstatus.inc
include \masm32\include\w2k\ntddk.inc

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;                                           C O D E

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

.code

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;                                         DriverEntry

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

DriverEntry proc pDriverObject:PDRIVER_OBJECT,
pusRegistryPath:PUNICODE_STRING

    mov eax, STATUS_DEVICE_CONFIGURATION_ERROR
    ret

DriverEntry endp

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

end DriverEntry

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;                               B U I L D I N G   D R I V E R

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:make

set drv=simplest

\masm32\bin\ml /nologo /c /coff %drv%.bat
\masm32\bin\link /nologo /driver /base:0x10000 /align:32
/out:%drv%.sys
  /subsystem:native %drv%.obj

del %drv%.obj
```

```
echo.
Pause
```

Driver Entry Routine

همانند تمامی ماژول‌های قابل اجرا، هر درایور نیز نیازمند یک روال ورودی (Entry Point) است. تا زمانیکه در حافظه بارگذاری می‌شود فراخوانی شود. روال ورودی درایورها روال Driver Entry می‌باشد. معمولاً این روال وظیفهٔ انجام عملیات آماده‌سازی و سایر عملیات ابتدایی را برعهده دارد. در زیر، پیش تعریف این تابع را مشاهده می‌کنید.

```
DriverEntry proto DriverObject:PDRIVER_OBJECT,
RegistryPath:PUNICODE_STRING
```

نوع داده PDRIVER_OBJECT و PUNICODE_STRING در فایل \include\w2k\ntddk.inc و \include\w2k\ntdef.inc تعریف شده‌اند.

```
PDRIVER_OBJECT  typedef PTR DRIVER_OBJECT
PUNICODE_STRING typedef PTR UNICODE_STRING
```

وقتی که I/O Manager روال Driver Entry را صدا می‌زند، دو پارامتر به آن می‌فرستد که توضیح آنها به شرح زیر است:

PDriverObject : اشاره‌گر به شیء معرفی شده درایور است. ویندوز NT یک سیستم عامل شی‌گرا است. پس درایورها نیز باید به‌صورت اشیاء به نمایش درآیند. هنگامی که یک درایور در حافظه بارگذاری می‌شود، سیستم شیء مربوط به آنرا ایجاد می‌کند. شیء درایور چیزی جز یک ساختمان داده DRIVER_OBJECT نیست و اشاره‌گر DriverObject امکان دسترسی به آن ساختمان داده را به شما می‌دهد. در حال حاضر کاری با آن ساختمان داده نداریم.

pusRegistryPath : یک اشاره‌گر به یک رشتهٔ Unicode که مسیر زیر کلید درایور را در رجیستری مشخص می‌کند.

در قسمت قبل درباره زیر کلیدهای درایور در رجیستری بحث کردیم. درایور می‌تواند از این اشاره‌گر به منظور ذخیره‌سازی اطلاعات مورد نیاز خود استفاده کند. در صورتی که درایور پس از اتمام روال Driver Entry به مسیر فوق نیاز داشته باشد، باید یک کپی از آن تهیه کند چون در

خارج از روال Driver Entry این مقدار معتبر نخواهد بود. رشتهٔ Unicode مذکور یک نمونه از ساختمان داده UNICODE_STRING است. برخلاف مد کاربر در مد هسته با رشته‌هایی با فرمت UNICODE_STRING کار می‌کنیم.

```
UNICODE_STRING STRUCT
    _Length         WORD    ?
    MaximumLength   WORD    ?
    Buffer          PWSTR   ?
UNICODE_STRING ENDS
```

Length : طول رشته به بایت، بدون شمارش آخرین کاراکتر صفر.

Maximum Length : طول بافر که توسط عضو بافر به آن اشاره می‌شود (بر حسب بایت).

Buffer : اشاره‌گر به رشته Unicode. توجه داشته باشید که همیشه این رشته مختوم به صفر نیست.

یکی از مهمترین امتیازات این روش این است که طول رشتهٔ فعلی و بیشترین طول آن معلوم است. این امر می‌تواند از محاسبات اضافی جلوگیری کند. در این قسمت درایور Simplest.sys را مورد بررسی قرار خواهیم داد. تنها کاری که این درایور انجام می‌دهد این است که اجازهٔ بارگذاری خود را می‌دهد. از آنجایی که این درایور کار خاصی انجام نمی‌دهد، خطای STATUS_DEVICE_CONFIGURATION_ERROR را برمی‌گرداند. اگر مقدار STATUS_SUCCESS را بازگردانید ، درایور در حافظه باقی خواهد ماند و نمی‌توانید آن را unload کنید. توسط برنامه KmdManager می‌توانید هر درایوری را رجیستر و بارگذاری کنید.

درایوری برای استفاده از بلندگوی داخلی

```
;@echo off
;goto make

;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;

;  beeper - Kernel Mode Drive

;  Makes beep thorough computer speaker

;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

.386
.model flat, stdcall
option casemap:none

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                              I N C L U D E   F I L E S

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

include \masm32\include\w2k\ntstatus.inc
include \masm32\include\w2k\ntddk.inc
include \masm32\include\w2k\hal.inc

includelib \masm32\lib\w2k\hal.lib

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                     E Q U A T E S

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

TIMER_FREQUENCY        equ 1193167                  ; 1,193,167 Hz
OCTAVE                 equ 2                        ; octave
multiplier

PITCH_C                equ 523                      ; C        -
523,25 Hz
PITCH_Cs               equ 554                      ; C#       -
554,37 Hz
PITCH_D                equ 587                      ; D        -
587,33 Hz
PITCH_Ds               equ 622                      ; D#       -
622,25 Hz
PITCH_E                equ 659                      ; E        -
659,25 Hz
PITCH_F                equ 698                      ; F        -
698,46 Hz
PITCH_Fs               equ 740                      ; F#       -
739,99 Hz
PITCH_G                equ 784                      ; G        -
783,99 Hz
PITCH_Gs               equ 831                      ; G#       -
830,61 Hz
PITCH_A                equ 880                      ; A        -
880,00 Hz
PITCH_As               equ 988                      ; B        -
987,77 Hz
PITCH_H                equ 1047                     ; H        -
1046,50 Hz

; We are going to play c-major chord
```

```
TONE_1                  equ TIMER_FREQUENCY/(PITCH_C*OCTAVE)
TONE_2                  equ TIMER_FREQUENCY/(PITCH_E*OCTAVE)
TONE_3                  equ (PITCH_G*OCTAVE)           ; for
HalMakeBeep

DELAY                   equ 1800000h                   ; for my
~800mHz box

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:
;                                         M A C R O S

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:

DO_DELAY MACRO
    mov eax, DELAY
    .while eax
        dec eax
    .endw
ENDM

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:
;                                            C O D E

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:

.code

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:
;                                            MakeBeep1

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:

MakeBeep1 proc dwPitch:DWORD

    ; Direct hardware access

    cli

    mov al, 10110110y
    out 43h, al

    mov eax, dwPitch
    out 42h, al

    mov al, ah
```

```
    out 42h, al

    ; Turn speaker ON

    in al, 61h
    or  al, 11y
    out 61h, al

    sti

    DO_DELAY

    cli

    ; Turn speaker OFF

    in al, 61h
    and al, 11111100y
    out 61h, al

    sti

    ret

MakeBeep1 endp

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:
;                                            MakeBeep2

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:

MakeBeep2 proc dwPitch:DWORD

    ; Hardware access using WRITE_PORT_UCHAR and READ_PORT_UCHAR
    ; functions from hal.dll

    cli

    invoke WRITE_PORT_UCHAR, 43h, 10110110y

    mov eax, dwPitch
    invoke WRITE_PORT_UCHAR, 42h, al
    mov eax, dwPitch
    invoke WRITE_PORT_UCHAR, 42h, ah

    ; Turn speaker ON

    invoke READ_PORT_UCHAR, 61h
    or  al, 11y
    invoke WRITE_PORT_UCHAR, 61h, al
```

```
    sti

        DO_DELAY

    cli

    ; Turn speaker OFF

    invoke READ_PORT_UCHAR, 61h
    and al, 11111100y
    invoke WRITE_PORT_UCHAR, 61h, al

    sti

    ret

MakeBeep2 endp

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:
;                                       DriverEntry

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:

DriverEntry proc pDriverObject:PDRIVER_OBJECT,
pusRegistryPath:PUNICODE_STRING

    invoke MakeBeep1, TONE_1
    invoke MakeBeep2, TONE_2

    ; Hardware access using hal.dll HalMakeBeep function

    invoke HalMakeBeep, TONE_3
    DO_DELAY
    invoke HalMakeBeep, 0

    mov eax, STATUS_DEVICE_CONFIGURATION_ERROR
    ret

DriverEntry endp

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:
;

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:

end DriverEntry
```

```
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:
;

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:

:make

set drv=beeper

\masm32\bin\ml /nologo /c /coff %drv%.bat
\masm32\bin\link /nologo /driver /base:0x10000 /align:32
/out:%drv%.sys
  /subsystem:native %drv%.obj

del %drv%.obj

echo.
pause
```

این درایور با استفاده از بلندگوی داخلی Mother Board، اصواتی را تولید می‌کند. به این منظور این درایور از دستورات In و Out مربوط به CPU استفاده می‌کند. دسترسی به پورت‌های I/O به عنوان یکی از منابع مهم، توسط ویندوزNT محافظت می‌شود و تلاش برای اجرای دستورات In یا Out در مد کاربر به منزلهٔ ختم فرآیند خواهد بود. اما راهی برای انجام این کار وجود دارد که در آینده به آن خواهیم پرداخت.

کنترل تایمر داخلی سیستم

سه تایمر در داخل کامپیوتر وجود دارند که به عنوان تایمرهای 0 و 1 و 2 شناخته می‌شوند و در Programmable Internal Timer (PIT) قرار دارند. از تایمر ۲ برای تولید صدا استفاده می‌شود. فرکانسی که توسط تایمر ۲ ایجاد می‌شود، توسط یک مقدار آغازی و ابتدایی مشخص می‌شود. تایمر از این مقدار تا صفر می‌شمرد و زمانی که به صفر رسید ، نوسان را ایجاد می‌کند. تایمر به مقدار پیش‌فرض برگشته و فرآیند دوباره انجام می‌شود. شمارش معکوس به سمت پایین توسط اسیلاتور اصلی سیستم کنترل می‌شود. این اسیلاتور با فرکانس 1,193,180 Hz کار می‌کند. این فرکانس در تمام خانواده‌های کامپیوترهای شخصی یکسان است. برای تغییر فرکانس‌های ایجاد شده فقط کافی است که یک مقدار پیش‌فرض جدید بدهیم در این جا یک نکتهٔ ظریف وجود دارد و آن این است که QueryPerformanceFrequency از Kernell32.dll مقدار واقعی 1193180 را برمی‌گرداند اما در hall.dll تابع halmakebeep مقدار متفاوت 1193167 را استفاده می‌کند و ما نیز از این مقدار استفاده می‌کنیم. در این جا اولین صدا را با استفاده از روال makebeep1 تولید می‌کنیم.

```
    mov al, 10110110y
    out 43h, al
```

ابتدا باید ثبات کنترل تایمر را ست کنیم. برای این کار مقدار باینری 10110110 را در پورت 43h می‌نویسیم.

```
    mov eax, dwPitch
    out 42h, al

    mov al, ah
    out 42h, al
```

حال با استفاده از دو دستور متوالی مقدار پایین و بالای بایت مقدار پیش‌فرض را در پورت 42h می‌نویسیم.

```
    in al, 61h
    or  al, 11y
    out 61h, al
```

حال با ست کردن بیت‌ها با صفر و یک در پورت 61h ، بلندگو را روشن می‌کنیم. حال بلندگو صدا را ایجاد می‌کند.

```
DO_DELAY MACRO
    mov eax, DELAY
    .while eax
        dec eax
    .endw
ENDM
```

حال با استفاده از ماکروی DO_DELAY به بلندگو مجوز پخش در بعضی زمان‌ها داده می‌شود.

```
    in al, 61h
    and al, 11111100y
    out 61h, al
```

برای خاموش کردن بلندگو نیاز داریم که بیت‌های یک و دو را در پورت 61h صفر کنیم.

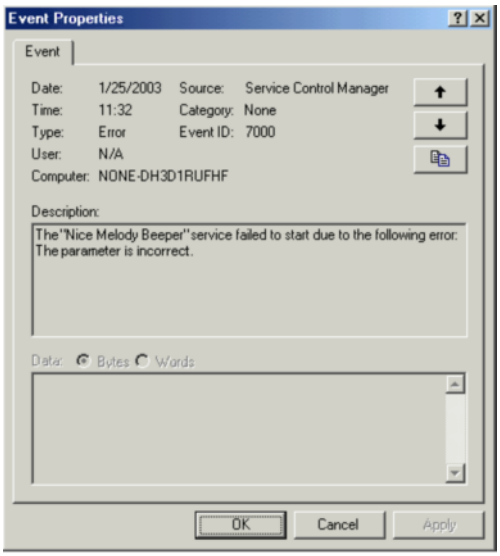
فراموش نکنید که تایمر یکی از منابع سراسری است پس باید وقفه‌های سخت‌افزاری قابل پوشش (maskeable) را از کار بیندازیم. با استفاده از روال makebeep2 دومین صدا را ایجاد می‌کنیم. نت

اولی که اجرا شد «دو» بود و این نت دوم «می» می‌باشد. در تابع دوم بجای استفاده از دستورالعملهای in/out از دو تابع READ_PORT_UCHAR و WRITE_PORT_UCHAR در hal.dll استفاده می‌کنیم. HAL وابستگی‌های سخت‌افزاری را پنهان می‌کند و کار با I/O را مستقل از ماشین می‌کند. سومین صدایی که ایجاد می‌کنیم نت «سل» است که آن را با استفاده از تابع HalMakeBeep ایجاد می‌کنیم. این تابع از توابع فایل hal.dll بوده و نیازی به مقدار پیش‌فرض اولیه ندارد.

در ابتدای فایل beeper.bat ، ۱۲ نت وجود دارد که ما از سه تای آنها استفاده کردیم، برای خاموش کردن بلندگو باید یک بار دیگر تابع HalMakeBeep را فراخوانی کرده و مقدار صفر را به عنوان آرگومان به آن بفرستیم.

شروع خودکار درایور

می‌دانیم که روش‌های مختلفی برای شروع درایور وجود دارد. حال می‌خواهیم سیستم را مجبور کنیم که درایور ما را به طور اتوماتیک راه‌اندازی کند. این کار به روش‌های مختلفی امکان‌پذیر است ساده‌ترین روش این است که در فراخوانی تابع DeleteService مقدار SERVICE_DEMAND_START را به SERVICE_AUTO_START تغییر دهیم و SERVICE_ERROR_IGNORE را به SERVICE_ERROR_NORMAL تغییر داده و فایل جدید SCP.asm را دوباره کامپایل کرده و اجرا کنیم. بعد از اینکه اجرای فایل SCP.exe به پایان رسید، رجیستری حاوی یک سرویس کاملاً جدید خواهد بود. بعد از بوت شدن سیستم درایور beep.sys عملیات خود را انجام خواهد داد. همان‌طور که در شکل زیر مشاهده می‌کنید در بخش Event Log می‌توانید اطلاعاتی را دربارهٔ درایورها بیابید (Start Menu← Programs ← Administrative Tools←Event Viewer).



فراموش نکنید که اطلاعات و داده‌های مربوط به درایور را از رجیستری پاک کنید در غیراینصورت با هر بار بوت شدن سیستم، این ملودی را خواهید شنید.

دسترسی به CMOS

در مادربورد کامپیوترها یک میکروچیپ وجود دارد که از آن برای ذخیره اطلاعات پیکربندی حافظه، دیسک‌های سخت، زمان و ساعت استفاده می‌شود. انرژی لازم برای این میکروچیپ توسط باتری تأمین می‌گردد و اطلاعات آن را می‌توان با دسترسی به پورت 70h و 71h , I/O بدست آورد.

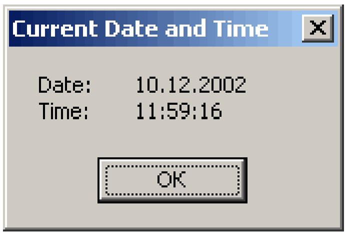
```
mov al, 0Bh
out 70h, al
in al, 71h

push eax
and al, 11111011y
or al, 010y
out 71h, al
```

ابتدا فرمت مناسب اطلاعات را با استفاده از رجیستر B تعیین می‌کنیم. با استفاده از ماکروهای CMOS می‌توانیم اطلاعات مورد نیاز را بدست آوریم و در همان زمان آنها را فرمت‌بندی کنیم.

```
invoke wsprintf, addr acOut,\
```

```
 $CTA0("Date:\t%s\nTime:\t%s"),\
 addr acDate,\
 addr acTime
    invoke MessageBox, NULL,\
 addr acOut,\
 $CTA0("Current Date and Time"),\
 MB_OK
```

حال اطلاعات را به نمایش می‌گذاریم و شما چیزی شبیه به شکل زیر را مشاهده خواهید کرد:



ما توانسته‌ایم به حافظه SMOS دسترسی پیدا کنیم بدون اینکه سیستم، ما را متوقف کند. همان‌طور که در قبل متذکر شدیم، دسترسی به پورت‌های I/O در ویندوز NT محافظت شده است و اجرای دستورات In یا Out در مد کاربر باعث ختم اجرای فرآیند خواهد شد.

درایوری‌برای تغییراجازه‌های دسترسی به‌پورت‌های سخت‌افزاری

```
;@echo off
;goto make

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:
;

;  giveio - Kernel Mode Driver

;



:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:

.386
.model flat, stdcall
option casemap:none
```

```
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:
;                              I N C L U D E   F I L E S

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:

include \masm32\include\w2k\ntstatus.inc
include \masm32\include\w2k\ntddk.inc
include \masm32\include\w2k\ntoskrnl.inc

includelib \masm32\lib\w2k\ntoskrnl.lib

include \masm32\Macros\Strings.mac

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:
;                                     E Q U A T E S

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:

IOPM_SIZE equ 2000h     ; sizeof I/O permission map

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:
;                                        C O D E

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:

.code

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:
;                                       DriverEntry

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:

DriverEntry proc pDriverObject:PDRIVER_OBJECT,
pusRegistryPath:PUNICODE_STRING

local status:NTSTATUS
local oa:OBJECT_ATTRIBUTES
local hKey:HANDLE
local kvpi:KEY_VALUE_PARTIAL_INFORMATION
local pIopm:PVOID
local pProcess:LPVOID

    invoke DbgPrint, $CTA0("giveio: Entering DriverEntry")
```

```
    mov status, STATUS_DEVICE_CONFIGURATION_ERROR

    lea ecx, oa
    InitializeObjectAttributes ecx, pusRegistryPath, 0, NULL, NULL

    invoke ZwOpenKey, addr hKey, KEY_READ, ecx
    .if eax == STATUS_SUCCESS

        push eax
        invoke ZwQueryValueKey,hKey,\
                            $CCOUNTED_UNICODE_STRING("ProcessId",
                           4),\
                            KeyValuePartialInformation,\
                            addr kvpi,\
                            sizeof kvpi,\
                            esp
        pop ecx

        .if ( eax != STATUS_OBJECT_NAME_NOT_FOUND ) && ( ecx != 0 )

            invoke DbgPrint,\
           $CTA0("giveio: Process ID: %X"),\
         dword ptr (KEY_VALUE_PARTIAL_INFORMATION PTR [kvpi]).Data

            ; Allocate a buffer for the I/O permission map

            invoke MmAllocateNonCachedMemory, IOPM_SIZE
            .if eax != NULL
                mov pIopm, eax

                lea ecx, kvpi
                invoke PsLookupProcessByProcessId,\
                       word ptr (KEY_VALUE_PARTIAL_INFORMATION PTR
[ecx]).Data,\
 addr pProcess
                .if eax == STATUS_SUCCESS

                    invoke DbgPrint, $CTA0("giveio: PTR KPROCESS:
%08X"),pProcess

                    invoke Ke386QueryIoAccessMap, 0, pIopm
                    .if al != 0

                        ; I/O access for 70h port

                        mov ecx, pIopm
                        add ecx, 70h / 8
                        mov eax, [ecx]
                        btr eax, 70h MOD 8
                        mov [ecx], eax
```

```
                        ; I/O access for 71h port

                         mov ecx, pIopm
                         add ecx, 71h / 8
                         mov eax, [ecx]
                         btr eax, 71h MOD 8
                         mov [ecx], eax

                         invoke Ke386SetIoAccessMap, 1, pIopm
                         .if al != 0
                             invoke Ke386IoSetAccessProcess, \
                                                             pProcess,\
                                                             1
                             .if al != 0
                                 invoke DbgPrint, $CTA0("giveio: I/O
permission is successfully given")
                             .else
                                 invoke DbgPrint, $CTA0("giveio: I/O
permission is failed")
                                 mov status,\
                                      STATUS_IO_PRIVILEGE_FAILED

                             .endif
                         .else
                             mov status, STATUS_IO_PRIVILEGE_FAILED
                         .endif
                     .else
                         mov status, STATUS_IO_PRIVILEGE_FAILED
                     .endif
                     invoke ObDereferenceObject, pProcess
                 .else
                     mov status, STATUS_OBJECT_TYPE_MISMATCH
                 .endif
                 invoke MmFreeNonCachedMemory, pIopm, IOPM_SIZE
             .else
                 invoke DbgPrint, $CTA0("giveio: Call to
MmAllocateNonCachedMemory failed")
                 mov status, STATUS_INSUFFICIENT_RESOURCES
             .endif
         .endif
         invoke ZwClose, hKey
     .endif

     invoke DbgPrint, $CTA0("giveio: Leaving DriverEntry")

     mov eax, status
     ret

DriverEntry endp

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:
```

```
;

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:

end DriverEntry

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:
;

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:

:make

set drv=giveio

\masm32\bin\ml /nologo /c /coff %drv%.bat
\masm32\bin\link /nologo /driver /base:0x10000 /align:32
/out:%drv%.sys
 /subsystem:native %drv%.obj

del %drv%.obj

echo.
pause
```

تغییر در اجازه‌های دسترسی به پورتهای I/O

در این بخش درایور ما نقشهٔ بیتی دسترسی‌های مجاز به پورت‌های I/O را برای یک Process خاص تغییر می‌دهد. هر فرآیند یک نقشهٔ بیتی دسترسی مجاز به پورت‌های I/O دارد. هر بیت از این نقشه مربوط به یک بایت از پورت‌های I/O می‌باشد اگر این بیت 1 باشد، دسترسی ممنوع است و اگر این بیت صفر باشد، فرآیند می‌تواند به آن پورت دسترسی داشته باشد. فضای آدرس‌های I/O حاوی 64000 پورت I/O هشت بیتی قابل آدرس‌دهی هستند، حداکثر اندازه نقشه IOPM ، 2000h است.

دو توابع مکتوب نشده و بدون مستندات در فایل ntoskrnl.exe برای دستکاری IOPM وجود دارد ke386QueryIoAccessMap و ke386SetIoAccessMap.

```
Ke386QueryIoAccessMap proto stdcall dwFlag:DWORD, pIopm:PVOID
```

تابع ke386QueryIoAccessMap ، IOPM کنونی که اندازهٔ آن 2000h بایت است را از TSS در فضای حافظه‌ای که PIOPM به آن اشاره می‌کند، کپی می‌کند.

dwflag:

0 : فضای حافظه را با مقدار 0FF h پر می‌کند.

1 : IOPM کنونی را از TSS در حافظه کپی می‌کند

pIopm : به فضایی از حافظه اشاره می کند که IOPM کنونی در آن کپی خواهد شد. اندازهٔ این فضای حافظه نباید کمتر از 2000h بایت باشد.

در صورتیکه تابع فوق عملیات خود را با موفقیت انجام دهد ، در ثبات al مقدار غیر صفر قرار می‌دهد اما اگر در انجام عملیات با شکست مواجه شود مقدار al را صفر قرار می‌دهد.

```
Ke386SetIoAccessMap proto stdcall dwFlag:DWORD, pIopm:PVOID
```

تابع ke386SetIoAccessMap، IOPM مشخص شده با اندازه 2000h بایت را از حافظه‌ای که PIOPM بدان اشاره می‌کند، در TSS کپی می‌کند.

dwflag: مقدار این پارامتر فقط می‌تواند 1 باشد و یک به معنی اجازهٔ کپی است.

pIopm : به فضایی از حافظه اشاره می‌کند که شامل Iopm است. اندازه این فضای حافظه نباید کمتر از 2000h بایت باشد. اگر این تابع عملیات خود را با موفقیت انجام دهد، در ثبات al مقدار غیرصفر قرار می‌گیرد اما اگر در اجرای عملیات خود با شکست مواجه شود، ثبات al را صفر خواهد کرد.

پس از اینکه Iopm در TSS کپی شد، باید اشاره‌گر افست Iopm، به Iopm جدید اشاره کند. این کار توسط تابع ke386IoSetAccessProcess انجام می‌شود. این تابع نیز یکی از توابع بسیار مفید و مکتوب نشده از فایل ntoskrnl.exe است.

```
Ke386IoSetAccessProcess proto stdcall pProcess:PTR KPROCESS,
dwFlag:DWORD
```

تابع ke386IoSet Access Process اجازه یا عدم اجازه استفاده از Iopm را به فرآیند می‌دهد.

pProcess: به رکورد KPROCESS اشاره می‌کند که بعداً درباره آن توضیح خواهیم داد.

dw flag:

0 : اجازه دسترسی به پورت‌های I/O را نمی‌دهد.

1 : اجازه دسترسی به پورت‌های I/O را می‌دهد.

این تابع نیز اگر عملیات خود را با موفقیت انجام دهد ، مقدار غیرصفر در رجیستر al قرار می‌دهد اما اگر در انجام عملیات خود با شکست مواجه شد، مقدار al را صفر خواهد کرد.

***نکته** : تقریباً تمام توابع فایل Ntoskrnl دارای پیشوند هستند. معمولاً اولین حرف پیشوند آنها i است که به معنی internal است و بعد از آن حرف p است که به معنای private یا f که منظور fastcall است. ke به معنی kernel است. PsP توابع internal process support است و Mm توابع Memory Management است.*

اولین پارامتر تابع ke386IoSet Access Process یک اشاره‌گر به رکورد kProcess است. پیش تعریف این رکورد در فایل \include\w2k\w2kundoc.inc قرار دارد. علت قرار دادن این فایل در شاخهٔ w2k این است که توابع مکتوب نشده در نسخه‌های مختلف Windows NT با یکدیگر تفاوت دارند. برای مثال استفاده از آن برای درایور Windows XP مناسب نمی‌باشد. تابع ke386IoSetAccessProcess عضو IopmOffset از رکورد KPROCESS را به مقدار مناسب تغییر می‌دهد.

خواندن اطلاعات از Registry

به‌منظور فراخوانی تابع ke386IoSetAccessProcess به اشاره‌گر شی Process نیاز داریم که از راه‌های مختلفی می‌توان به آن دست پیدا کرد. در این جا از ساده‌ترین روش استفاده خواهیم کرد یعنی استفاده از شناسهٔ Process . برای این کار ابتدا در فایل Datatime.exe، شناسه فرآیند کنونی را بدست می‌آوریم و آنرا در رجیستری قرار می‌دهیم. در این حالت از رجیستری برای فرستادن پارامترهای مورد نیاز، از کدهای کاربر به درایور استفاده می‌کنیم. هنگامی که روال DriveEntry در System Process Context اجرا می‌شود هیچ راهی وجود ندارد که دریابیم چه فرآیندی درایور را شروع کرده است.

پارامتر دوم روال DriverEntry , pusRegistryPath است که یک اشاره‌گر به زیر کلید درایور در رجیستری است و ما از آن برای یافتن شناسه Process از رجیستری استفاده می‌کنیم. حال ببینیم این کارها چگونه انجام می‌شود.

```
    lea ecx, oa
    InitializeObjectAttributes ecx, pusRegistryPath, 0, NULL, NULL
```

قبل از اینکه تابع ZwOpenKey را فراخوانی کنیم ، باید رکورد OBJECT_ATTRIBUTES را معرفی کنیم . برای این کار از ماکروی InitializeObjectAttributes استفاده کردیم. اما بهتر است که این کار را خودتان انجام دهید زیرا ماکروی InitializeObjectAttributes همیشه آنگونه که شما انتظار دارید رفتار نمی‌کند. این کار را می‌توانید همانند مثال زیر انجام دهید.

```
lea ecx, oa
xor eax, eax
assume ecx:ptr OBJECT_ATTRIBUTES
mov [ecx].dwLength, sizeof OBJECT_ATTRIBUTES
mov [ecx].RootDirectory, eax                        ; NULL
push pusRegistryPath
pop [ecx].ObjectName
mov [ecx].Attributes, eax                           ; 0
mov [ecx].SecurityDescriptor, eax                   ; NULL
mov [ecx].SecurityQualityOfService, eax             ; NULL
assume ecx:nothing
```

تابع ZwOpenKey شماره دسترسی به کلید رجیستری را در hkey برمی‌گرداند. پارامتر دوم سطح دسترسی مورد نیاز را تعیین می‌کند. باید بدانید که ثبات ecx حاوی اشاره‌گر به Object Attribute کلید باز شده است.

```
    invoke ZwOpenKey, addr hKey, KEY_READ, ecx
    .if eax == STATUS_SUCCESS

        push eax
        invoke ZwQueryValueKey, hKey,\
                        $CCOUNTED_UNICODE_STRING("ProcessId", 4),\
                        KeyValuePartialInformation,\
                        addr kvpi,\
                        sizeof kvpi,\
                        esp
        pop ecx
```

ZwQueryValueKey مقدار کلید باز شده رجیستری را برمی‌گرداند و ما از این تابع به‌منظور یافتن شناسهٔ Process از رجیستری استفاده می‌کنیم.

پارامتر دوم اشاره‌گر به نامی است که برای مقدار موردنظر در نظر گرفته شده است.

در اینجا از ماکروی $CCOUNTED_UNICODE_STRING برای تعریف رکورد UNICODE_STRING استفاده کردیم. اگر علاقه‌ای به استفاده از ماکروها ندارید، می‌توانید از روش زیر استفاده کنید.

```
usz dw 'U', 'n', 'i', 'c', 'o', 'd', 'e', ' ', 's', 't', 'r', 'i',
'n', 'g', 0
 us UNICODE_STRING {sizeof usz - 2, sizeof usz, offset usz}
```

روش بالا مناسب نیست و به این دلیل ماکروهای ,COUNTED_UNICODE_STRING $COUNTED_UNICODE_STRING, CCOUNTED_UNICODE_STRING, $CCOUNTED_UNICODE_STRING (\Macros\Strings.mac) را نوشتیم و از آنها استفاده می‌کنیم.

پارامتر سوم نوع دادهٔ مورد نیاز را مشخص می‌کند. KeyValuePartialInformation یک ثابت سمبلیک است که در (\include\w2k\ntddk.inc) تعریف شده است. پارامترهای چهارم و پنجم اشاره‌گر به رکورد KEY_VALUE_PARTIAL_INFORMATION و اندازهٔ آن هستند. از عضو Data از این رکورد می‌توان شناسه Process موردنظر را یافت. آخرین پارامتر هم اشاره‌گر به تعداد بایت‌های بازگشتی است. باید بدانیم که قبل از فراخوانی تابع ZwQueryValueKey باید فضایی را روی Stack برای آن رزرو کنیم.

اعطای اجازهٔ دسترسی مستقیم به پورت‌های Process های کاربر

```
        .if ( eax != STATUS_OBJECT_NAME_NOT_FOUND ) && ( ecx != 0 )
            invoke MmAllocateNonCachedMemory, IOPM_SIZE
            .if eax != NULL
                mov pIopm, eax
```

اگر تابع ZwQueryValueKey عملیات خود را با موفقیت انجام داد، با فراخوانی تابع MmAllocateNonCachedMemory حافظهٔ مجازی مورد نیاز را به Iopm اختصاص دهیم.

```
                lea ecx, kvpi
                invoke PsLookupProcessByProcessId, \
                        dword ptr (KEY_VALUE_PARTIAL_INFORMATION PTR
[ecx]).Data, addr pProcess
                .if eax == STATUS_SUCCESS
                    invoke Ke386QueryIoAccessMap, 0, pIopm
```

با فرستادن ProcessIdentifier به تابع PsLookupProcessByProcessId، اشاره‌گر به ProcessObject را وارد PProcess می‌کنیم. تابع Ke386QueryIoAccessMap، IOPM را در فضای حافظه پر می‌کند.

```
.if al != 0

    mov ecx, pIopm
    add ecx, 70h / 8
    mov eax, [ecx]
    btr eax, 70h MOD 8
    mov [ecx], eax

    mov ecx, pIopm
    add ecx, 71h / 8
    mov eax, [ecx]
    btr eax, 71h MOD 8
    mov [ecx], eax

    invoke Ke386SetIoAccessMap, 1, pIopm
    .if al != 0
        invoke Ke386IoSetAccessProcess, \
                    pProcess, 1

        .if al != 0
        .else
            mov status, \
                    STATUS_IO_PRIVILEGE_FAILED
        .endif
    .else
        mov status, STATUS_IO_PRIVILEGE_FAILED
    .endif
.else
    mov status, STATUS_IO_PRIVILEGE_FAILED
.endif
```

حال ما بیت‌های متناظر با پورت‌های 71-70 را پاک می‌کنیم و IOPM تغییر یافته را دوباره بازنویسی کرده و تابع ke386SetAccessProcess را برای برقراری دسترسی به I/O فراخوانی می‌کنیم.

```
    invoke ObDereferenceObject, pProcess
.else
    mov status, STATUS_OBJECT_TYPE_MISMATCH
.endif
```

فراخوانی قبلی تابع PsLookupProcessByProcessId شمارش‌گر ارجاعات به Process Object را افزایش می‌دهد. برای کاهش شمارش‌گر ارجاعات ، تابع ObDereferenceObject را فراخوانی می‌کنیم.

```
invoke MmFreeNonCachedMemory, pIopm, IOPM_SIZE
.else
invoke DbgPrint,\
$CTA0("giveio: Call to MmAllocateNonCachedMemory failed")
mov status, STATUS_INSUFFICIENT_RESOURCES
.endif
.endif
invoke ZwClose, hKey
.endif
```

با فراخوانی تابع MmFreeNonCachedMemory فضای حافظه را آزاد کرده و با فراخوانی ZwClose شماره دسترسی به رجیستری را می‌بندیم.

عملیات موردنظر انجام شده و دیگر نیازی به درایور نیست هنگامی که درایور یک کد خطا باز می‌گرداند، سیستم آنرا از حافظه از بین می‌برد. اما اکنون فرآیند مد کاربر دسترسی مستقیم به دو پورت I/O دارد. با این تکه کوچک از کد حتی می‌توانید به خودتان اجازهٔ دسترسی به تمام ۶۵۵۳۵ پورت را اعطا کنید.

```
invoke MmAllocateNonCachedMemory, IOPM_SIZE
.if eax != NULL
    mov pIopm, eax
    invoke RtlZeroMemory, pIopm, IOPM_SIZE
    lea ecx, kvpi
    invoke PsLookupProcessByProcessId, \
                dword ptr (KEY_VALUE_PARTIAL_INFORMATION PTR
[ecx]).Data,\
 addr pProcess
    .if eax == STATUS_SUCCESS
        invoke Ke386SetIoAccessMap, 1, pIopm
        .if al != 0
            invoke Ke386IoSetAccessProcess, pProcess, 1
        .endif
        invoke ObDereferenceObject, pProcess
    .endif
    invoke MmFreeNonCachedMemory, pIopm, IOPM_SIZE
.else
    mov status, STATUS_INSUFFICIENT_RESOURCES
.endif
```

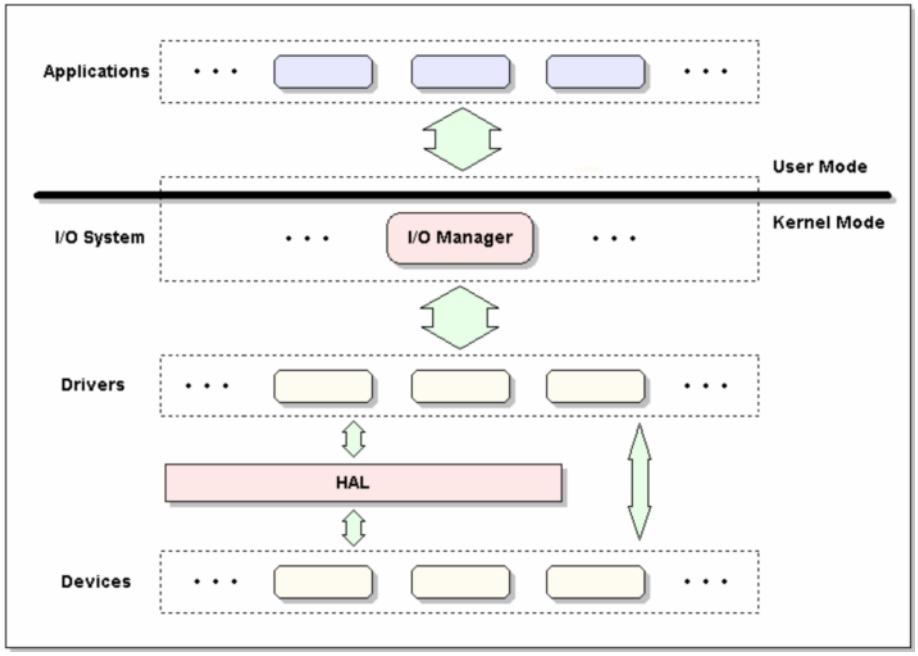
همیشه در ذهن داشته باشید که پخش کردن اصوات توسط Speaker سیستم و خواندن از حافظهٔ CMOS به اندازه کافی خطرناک است. اما دسترسی به پورت‌های I/O می‌تواند خطرناک‌تر باشد چون اساساً نمی‌توانید آنها را با مد کاربر هماهنگ کنید.

زیر سیستم I/O

I/O manager

در مد کاربر به سادگی توابع را با استفاده از آدرس آنها، از فایل‌های dll فراخوانی کرده و از آنها استفاده می‌کردیم. در مد هسته چنین کاری از نقطه نظر پایداری سیستم بسیار خطرناک است. بنابراین سیستم عامل واسطی را فراهم آورده است که توسط آن با مد هسته ارتباط برقرار می‌شود. یکی از این واسطها I/O Manager است. این واسط یکی از ابزارهای **زیر سیستم** I/O محسوب می‌شود. I/O Manager ارتباط بین برنامه‌ها و ابزارهای سیستم را با درایورهای سیستمی برقرار می‌کند و در حقیقت زیربنایی برای پشتیبانی از Device Driver ها تعریف می‌کند.

شکل بسیار ساده‌ای که چگونگی کارکرد متقابل I/O Manager با برنامه‌های مد کاربر و Device Driver ها را در زیر مشاهده می‌کنید.



با توجه به شکل درخواهیم یافت که مطمئناً تمام فراخوانی‌ها از برنامه‌های مد کاربر به مد هسته تحت کنترل I/O Manager خواهد بود. کدهای مد کاربر مجبور هستند که عملیات مربوط به I/O را توسط یک درایور انجام دهند و درایور باید حداقل یک Device ایجاد کند. در اینجا منظور یک

Device مجازی است. این را باید بدانیم که ایجاد Device به این معنی نیست که یک Device واقعی ایجاد خواهد شد منظور این است که یک شی جدید در حافظه ایجاد می‌شود که خود را به عنوان نماینده یک Device منطقی یا فیزیکی معرفی می‌کند و خصوصیات آنرا بیان می‌کند. با ثبت Device جدید و معرفی آن، I/O Manager درخواست‌های I/O مربوطه را به آن Device ارسال کرده و نتایج را به درخواست‌کننده باز می‌گرداند. تنها مسئولیت I/O Manager این است که درخواست‌های I/O را به Device Driver مناسب بفرستد زیرا کد کاربر نمی‌داند که چه درایوری به یک Device خاص سرویس می‌دهد.

برنامه کنترلی برای درایور (Virtophys)

کد این قسمت دربارۀ Service Control Program است که مسئول ثبت و شروع درایور و برنامه Client ای است که باید با این درایور ارتباط برقرار کند.

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:
;

; VirtToPhys.asm - Driver Control Program for VirtToPhys driver

;

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:

.386
.model flat, stdcall
option casemap:none

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:
;                              I N C L U D E   F I L E S

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:

include \masm32\include\windows.inc

include \masm32\include\kernel32.inc
include \masm32\include\user32.inc
include \masm32\include\advapi32.inc

includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\user32.lib
includelib \masm32\lib\advapi32.lib
```

```
include \masm32\include\winioctl.inc

include \masm32\Macros\Strings.mac

include common.inc

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:
;                                       C O D E

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:

.code

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:
;                                     BigNumToString

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:

BigNumToString proc uNum:UINT, pszBuf:LPSTR

; This function accepts a number and converts it to a
; string, inserting commas where appropriate.

local acNum[32]:CHAR
local nf:NUMBERFMT

    invoke wsprintf, addr acNum, $CTA0("%u"), uNum

    and nf.NumDigits, 0
    and nf.LeadingZero, FALSE
    mov nf.Grouping, 3
    mov nf.lpDecimalSep, $CTA0(".")
    mov nf.lpThousandSep, $CTA0(" ")
    and nf.NegativeOrder, 0
    invoke GetNumberFormat, LOCALE_USER_DEFAULT,\
                            0,\
                            addr acNum,\
                            addr nf,\
                            pszBuf,\
                            32
    ret

BigNumToString endp

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:
```

```
;                                       start

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:

start proc uses esi edi

local hSCManager:HANDLE
local hService:HANDLE
local acModulePath[MAX_PATH]:CHAR
local _ss:SERVICE_STATUS
local hDevice:HANDLE

local adwInBuffer[NUM_DATA_ENTRY]:DWORD
local adwOutBuffer[NUM_DATA_ENTRY]:DWORD
local dwBytesReturned:DWORD

local acBuffer[256+64]:CHAR
local acThis[64]:CHAR
local acKernel[64]:CHAR
local acUser[64]:CHAR
local acAdvapi[64]:CHAR

local acNumber[32]:CHAR

    invoke OpenSCManager, NULL, NULL, SC_MANAGER_ALL_ACCESS
    .if eax != NULL
        mov hSCManager, eax

        push eax
        invoke GetFullPathName, $CTA0("VirtToPhys.sys"), \
                                sizeof acModulePath,\
  addr acModulePath,\
  esp
        pop eax

        invoke CreateService, hSCManager,\
                              $CTA0("VirtToPhys"),\
                              $CTA0("Virtual To Physical Address
                              Converter"),\
                              SERVICE_START + \
                              SERVICE_STOP + DELETE,\
                              SERVICE_KERNEL_DRIVER,\
                              SERVICE_DEMAND_START,\
                              SERVICE_ERROR_IGNORE,\
                              addr acModulePath,\
                              NULL,\
                              NULL,\
                              NULL,\
                              NULL,\
                              NULL
        .if eax != NULL
```

```
            mov hService, eax

            ; Driver's DriverEntry procedure will be called
            invoke StartService, hService, 0, NULL
            .if eax != 0

                ; Driver will receive I/O request packet (IRP) of type IRP_MJ_CREATE
                invoke CreateFile, $CTA0("\\\\.\\slVirtToPhys"),\
                                   GENERIC_READ + GENERIC_WRITE, \
                                   0,\
                                   NULL,\
                                   OPEN_EXISTING,\
                                   0,\
                                   NULL
                .if eax != INVALID_HANDLE_VALUE
                    mov hDevice, eax

                    lea esi, adwInBuffer
                    assume esi:ptr DWORD
                    invoke GetModuleHandle, NULL
                    mov [esi][0*(sizeof DWORD)], eax
                    invoke GetModuleHandle, $CTA0("kernel32.dll",
szKernel32)
                    mov [esi][1*(sizeof DWORD)], eax
                    invoke GetModuleHandle, $CTA0("user32.dll",
szUser32)
                    mov [esi][2*(sizeof DWORD)], eax
                    invoke GetModuleHandle, $CTA0("advapi32.dll",
szAdvapi32)
                    mov [esi][3*(sizeof DWORD)], eax

                    lea edi, adwOutBuffer
                    assume edi:ptr DWORD
                    ; Driver will receive IRP of type
IRP_MJ_DEVICE_CONTROL
                    invoke DeviceIoControl, hDevice,\
                                            IOCTL_GET_PHYS_ADDRESS,\
                                            esi,\
                                            sizeof adwInBuffer,\
                                            edi,\
                                            sizeof adwOutBuffer,\
                                            addr dwBytesReturned,\
                                            NULL
                    .if ( eax != 0 ) && ( dwBytesReturned != 0 )

                        invoke GetModuleFileName, [esi][0*(sizeof
DWORD)],\
addr acModulePath,\
sizeof acModulePath

                        lea ecx, acModulePath[eax-5]
                        .repeat
                            dec ecx
                            mov al, [ecx]
```

```
                        .until al == '\'
                        inc ecx
                        push ecx

                        CTA0 "%s \t%08Xh\t%08Xh   ( %s )\n",
szFmtMod

                        invoke BigNumToString, [edi][0*(sizeof
DWORD)],\
 addr acNumber
                        pop ecx
                        invoke wsprintf, addr acThis,\
                                         addr szFmtMod,\
                                         ecx,\
                                         [esi][0*(sizeof DWORD)],\
                                         [edi][0*(sizeof DWORD)],\
                                         addr acNumber
                        invoke BigNumToString, [edi][1*(sizeof
DWORD)],\
 addr acNumber
                        invoke wsprintf, addr acKernel,\
                                         addr szFmtMod,\
                                         addr szKernel32,\
                                         [esi][1*(sizeof DWORD)],\
                                         [edi][1*(sizeof DWORD)],\
                                         addr acNumber
                        invoke BigNumToString, [edi][2*(sizeof
DWORD)],\
 addr acNumber
                        invoke wsprintf, addr acUser,\
                                         addr szFmtMod,\
                                         addr szUser32,\
                                         [esi][2*(sizeof DWORD)],\
                                         [edi][2*(sizeof DWORD)],\
                                         addr acNumber
                        invoke BigNumToString, [edi][3*(sizeof
DWORD)],\
 addr acNumber
                        invoke wsprintf, addr acAdvapi,\
                                         addr szFmtMod,\
                                         addr szAdvapi32,\
                                         [esi][3*(sizeof DWORD)],\
                                         [edi][3*(sizeof DWORD)],\
                                         addr acNumber
                        invoke wsprintf, addr acBuffer, \
$CTA0("Module:\t\tVirtual:\t\tPhysical:\n\n%s\n%s%s%s"), \
                                        addr acThis,\
 addr acKernel,\
 addr acUser,\
 addr acAdvapi

                        assume esi:nothing
```

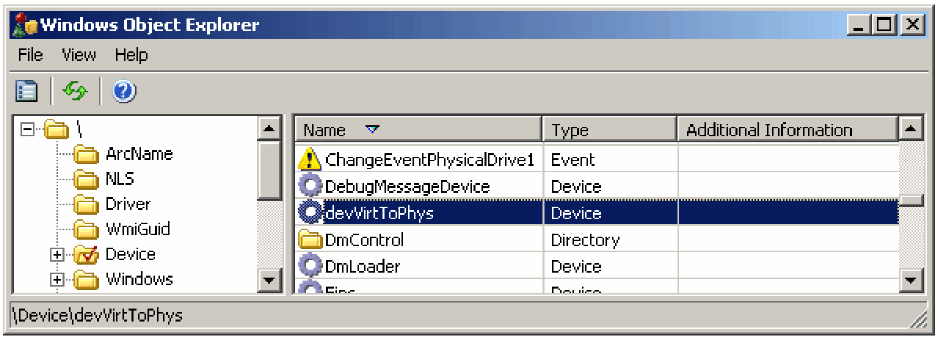
```
                        assume edi:nothing
                        invoke MessageBox, NULL,\
                                addr acBuffer,\
                                $CTA0("Modules Base Address"),\
                                MB_OK + MB_ICONINFORMATION
                    .else
                        invoke MessageBox, NULL,\
 $CTA0("Can't send control code to device."),\
 NULL, \
 MB_OK + MB_ICONSTOP
                    .endif
                    ; Driver will receive IRP of type IRP_MJ_CLOSE
                    invoke CloseHandle, hDevice
                .else
                    invoke MessageBox, NULL,\
                            $CTA0("Device is not present."),\
                            NULL,\
                            MB_ICONSTOP
                .endif
                ; DriverUnload proc in our driver will be called
                invoke ControlService, hService,
SERVICE_CONTROL_STOP, addr _ss
            .else
                invoke MessageBox, NULL,\
                                $CTA0("Can't start driver."),\
                                NULL,\
                                MB_OK + MB_ICONSTOP
            .endif
            invoke DeleteService, hService
            invoke CloseServiceHandle, hService
        .else
            invoke MessageBox, NULL,\
                            $CTA0("Can't register driver."),\
                            NULL,\
                            MB_OK + MB_ICONSTOP
        .endif
        invoke CloseServiceHandle, hSCManager
    .else
        invoke MessageBox, NULL,\
 $CTA0("Can't connect to Service Control Manager."),\
 NULL, \
 MB_OK + MB_ICONSTOP
    .endif

    invoke ExitProcess, 0

start endp
```

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:
;

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::;
:

end start
```

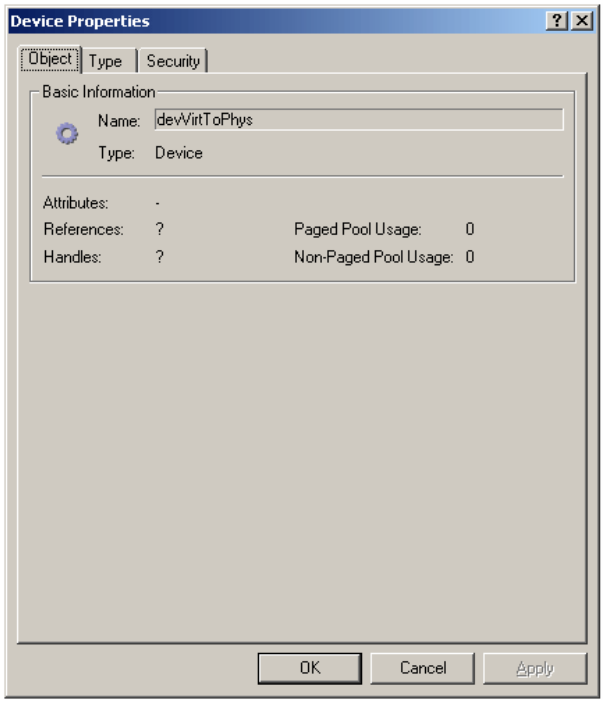
غیر از عملیات گفته شده در کد بالا، سه نکتهٔ جدید در آن وجود دارد و آن هم فراخوانی سه تابع CreateFile , DeviceIoControl و CloseHandle است. هر سه تابع ذکر شده شماره دسترسی Device (Handle) را به عنوان آرگومان خود می‌پذیرند.

شی ابزار

بعد از بازگذاری درایور VirToPhys، یک Device با نام "devVirToPhys" ایجاد می‌شود. نام Device در فضای نام Object Manager قرار می‌گیرد. Object Manager یکی از ابزارهای سیستم است که مسئول ایجاد، پاک کردن، محافظت و دنبال کردن اشیاء است . برای مشاهدهٔ فضای نامی که توسط Object Manager نگهداری می‌شود می‌توانید از یکی از دو برنامه Windows Object Explorer یا Object Viewer استفاده کنید. برای مشاهدهٔ اشیاء ایجاد شده توسط VirToPhys در کامپیوتر خود VirToPhys را اجرا کنید اما Dialog Box برنامه را نبندید.



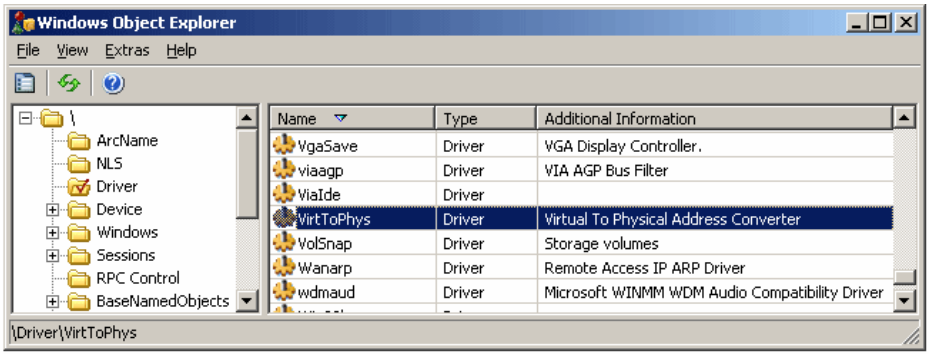
اشیاء devVirToPhysdevice در فضای نام Object Manager



خصوصیات شی *devVirtToPhys device*

شی درایور

شی درایور VirtToPhys در مسیر \Driver قرار گرفته است.



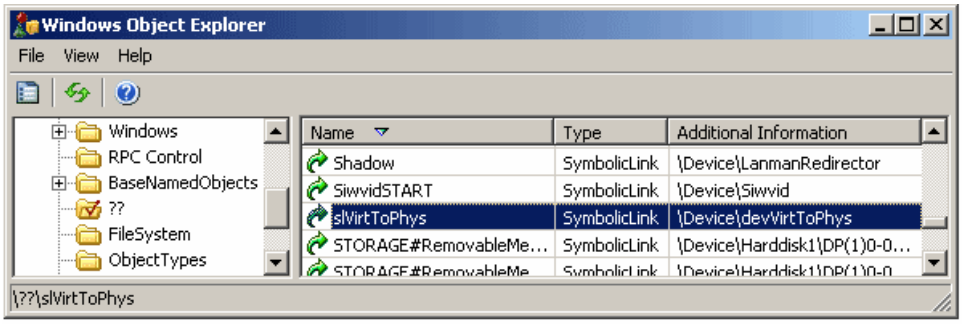
شی درایور *VirToPhys* در فضای نام *Object manager*

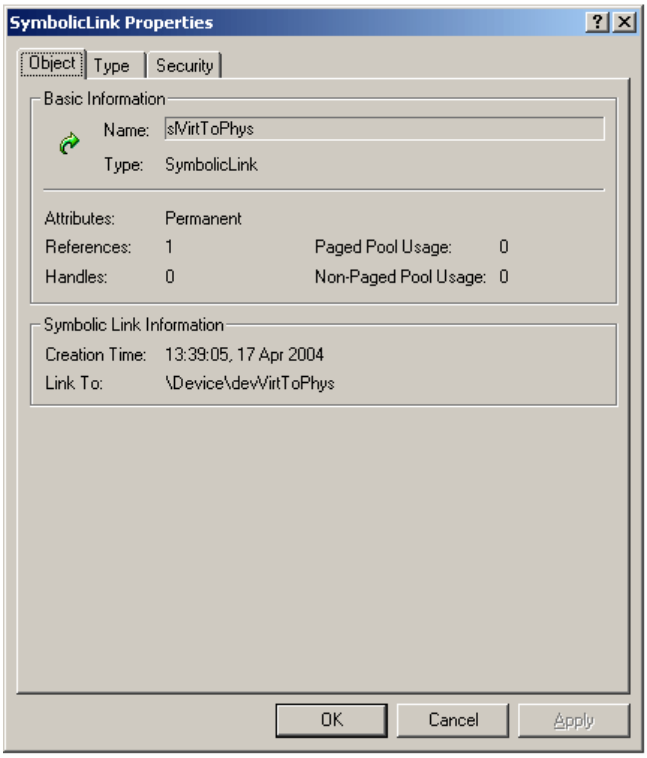
Symbolic Link Object

نام داخلی Device برای برنامه‌های Win32 قابل استفاده نیست. همهٔ شاخه‌ها به جز "\BaseNamedObjects" و"\??" از دید برنامه‌های کاربر پنهان هستند. در عوض نام Device باید در یک شاخهٔ خاص در فضای Object Manager قرار گیرد. شاخهٔ "\??" حاوی یک Symbolic Link به نام واقعی داخلی Device می‌باشد.

درایورها مسئول ایجاد لینک‌ها در این شاخه هستند تا Device های آنها برای برنامه‌های Win32 قابل دسترسی باشد. پس درایورها برای اینکه دسترسی کدهای مد کاربر را به Device Object ممکن سازند، باید یک لینک در شاخهٔ "\??" ایجاد کنند که به Device Object در شاخه \Device اشاره می‌کند. از این پس اگر متقاضی ، Device Handle خواست کند، I/O Manager می‌تواند آن را به سادگی پیدا کند. البته با استفاده از دو تابع QueryDosDevice و DefineDosDevice می‌توانید این لینک‌ها را از مد کاربر جستجو یا تغییر دهید. قبلاً در Windows NT نام این شاخه "\DosDevices" بود که به علت مسائل اجرایی به "\??" تغییر نام پیدا کرد تا در ترتیب الفبا در ابتدای لیست قرار گیرد. برای پشتیبانی گذشته در فضای نام Object Manager یک شاخهٔ "\DosDevices" وجود دارد که به همان مسیر "\??" می‌رود.

درایور VirtToPhys یک Symbolic Link با نام "slVirtToPhys " به " devVirtToPhys" Device در شاخهٔ "\??" ایجاد می‌کند. مقدار آن رشتهٔ "\Device\devVirtToPhys " خواهد بود.





پس از خارج شدن از تابع Start Service سه شی جدید داریم:

1- driver ← "\Driver\VirtToPhys"

2- Device ← " \Device\devVirtToPhys"

3- Symbolic Link ← " \??\slVirtToPhys"

شیء فایل

حال به کد فایل خود باز می‌گردیم بعد از اینکه درایور شروع شد می‌خواهیم روال‌های مورد نظر خود را از آن فراخوانی کنیم برای این کار به یک File Handle (شماره دسترسی فایل) نیاز داریم که با استفاده از تابع CreateFile ایجاد می‌کنیم. توضیح این تابع بسیار طولانی است و در اینجا به توضیح قسمت‌های مورد نیاز اکتفا می‌کنیم.

```
CreateFile proto stdcall  lpFileName:LPCSTR,
                          dwDesiredAccess:DWORD,\
                          dwShareMode:DWORD,\
```

```
lpSecurityAttributes:LPVOID,\
dwCreationDistribution:DWORD,\
dwFlagsAndAttributes:DWORD,\
hTemplateFile:HANDLE
```

این تابع هم اشیاء را ایجاد می‌کند و هم اشیایی که قبلاً ایجاد شده‌اند را باز می‌کند و تنها برروی فایل کار نمی‌کند Device ها نیز می‌تواند به عنوان به یک شی در نظر گرفته شوند.

lpFileName : به یک رشتهٔ مختوم به صفر اشاره می‌کند. این رشته نام Device یا فایلی را که قرار باز شود مشخص می‌کند. Symbolic Link به Device Object اشاره می‌کند.

dwDesiredAccess : نوع دسترسی به Device را مشخص می‌کند. در این قسمت به دو مقدار نیاز داریم:

GENERIC_READ : دسترسی خواندن را اختصاص می‌دهد یعنی داده از Device موردنظر می‌تواند خوانده شود.

GENERIC_WRITE : دسترسی نوشتن را اختصاص می‌دهد یعنی داده می‌تواند در Device مورد نظر نوشته شود. این مقادیر می‌توانند با یکدیگر ترکیب شوند.

dwShareMode : چگونگی به اشتراک گذاشتن Device را مشخص می‌کند.

0 : Device نمی‌تواند به اشتراک گذاشته شود و عملیات باز کردن‌های متوالی برروی Device با شکست مواجه خواهد شد. اما اگر نیاز به اشتراک گذاشتن Device خود دارید می‌توانید از مقادیر زیر استفاده کنید:

FILE_SHARE_READ : اگر درخواست خواندن داشته باشیم عملیات بازکردن متوالی در این حالت با موفقیت انجام خواهد شد.

FILE_SHARE_WRITE : اگر درخواست نوشتن بر روی Device داشته باشیم عملیات بازکردن متوالی با موفقیت انجام خواهد شد.

lpSecurityAttributes : اشاره‌گر به رکورد SECURITY_ATTRIBUTES می‌باشد. هنگامی که حفاظت خاصی نیاز نداریم به سادگی این مقدار را Null قرار می‌دهیم.

dwCreationDistribution : نوع عملیاتی را مشخص می‌کند که در صورت وجود یا عدم وجود فایل باید انجام شود. در مورد Device ها این مقدار باید همیشه OPEN_EXISTING باشد.

dwFlagsAndAttributes : پرچم‌ها و خصوصیات را مشخص می‌کند. این مقدار 0 خواهد بود.

hTemplateFile : یک شماره دسترسی به فایل الگو اختصاص می‌دهد. برای Device ها این مقدار باید همیشه Null باشد.

در صورتیکه تابع CreateFile عملیات خود را با موفقیت انجام دهد، شماره دسترسی به Device موردنظر را باز می‌گرداند در غیر این صورت مقدار INVALID_HANDLE_VALUE را باز خواهد گرداند. تابع CreateFile را به صورت زیر فراخوانی می‌کنیم.

```
invoke CreateFile, $CTA0("\\\\.\\slVirtToPhys"),\
                   GENERIC_READ + GENERIC_WRITE,\
                   0,\
                   NULL,\
                   OPEN_EXISTING,\
                   0,\
                   NULL
```

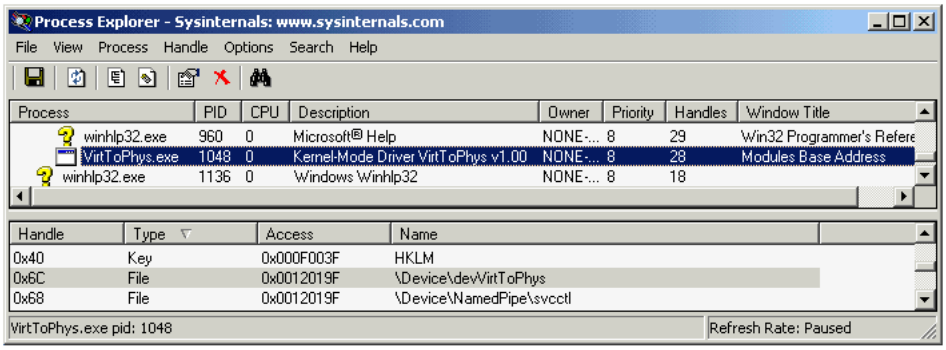
همه چیز در ۵ پارامتر آخر روشن است فقط پارامتر دوم ترکیب GENERIC_READ و GENERIC_WRITE است چون نیاز داریم که همه اطلاعات را به Device بفرستیم و هم اینکه اطلاعاتی را از Device دریافت کنیم. پارامتر اول یک اشاره‌گر به نام Symbolic Link به شکل "\\.\slVirtToPhys" است. در Win32 "\\.\" یک نام مستعار برای کامپیوترهای محلی است.

System Service یک محل ورودی به هستهٔ سیستم عامل از طرف زیر سیستم‌ها مانند Win32 یا Posix محسوب می‌شود. روال کنترلی یک سیستم سرویس پس از اجرای دستورالعمل وقفه 2EH در ویندوز NT و 2000 و یا Sysenter در ویندوز XP و 2003 فراخوانی می‌شود. توجه داشته باشید که این دستورالعمل‌ها به عنوان دستورالعمل‌های ممتاز CPU های 80x86 محسوب می‌شوند. اجرای این دستورالعمل‌ها باعث انتقال کنترل روند اجرایی در سطح کاربر به روند کنترلی System Service در سطح هسته می‌شود.

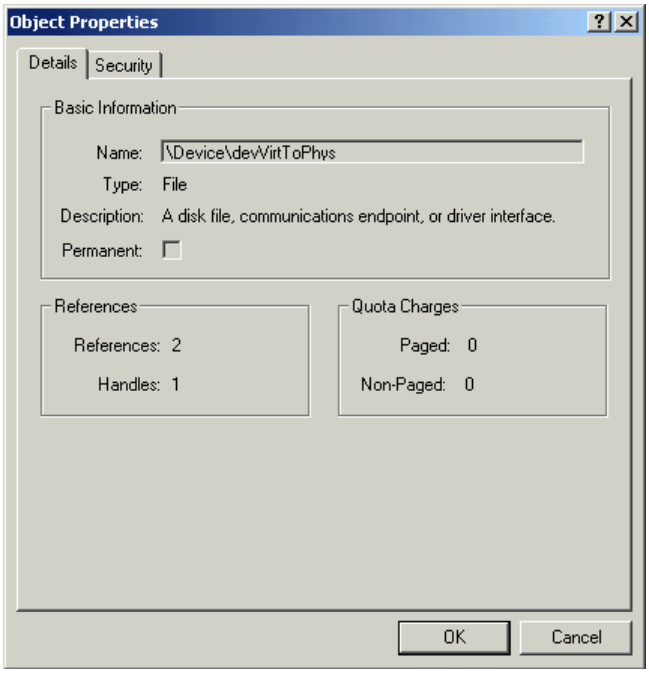
سیستم عامل تمام درخواست‌های I/O را به عنوان عملیات برروی فایل مجازی در نظر می‌گیرد. در اینجا درایور درخواست‌هایی را که از طرف درخواست‌کننده می‌آید به‌صورت درخواست عملیات بر روی فایل مجازی تبدیل می‌کند. این خلاصه‌سازی باعث عمومی شدن واسط برنامه‌ها با Device ها می‌شود. کلیه داده‌هایی که خوانده یا نوشته می‌شود به‌صورت رشته‌های ساده‌ای از بایت‌ها برروی فایل مجازی می‌باشد. قبل از اینکه تابع CreateFile مقداری را بازگرداند I/O Manager یک IRP از نوع IRP_MJ_CREATE ایجاد کرده و آنرا جهت پردازش به درایور می‌فرستد.

در صورتی‌که روال تعریف شده درایور با موفقیت عملیات خود را انجام دهد، Object Manager یک شماره دسترسی (Handle) برای شیء فایل در جدول شماره‌های دسترسی فرآیندها ایجاد

می‌کند و این شماره دسترسی با زنجیر فراخوانی‌ها به عقب باز می‌گردد تا به عنوان مقدار بازگشتی تابع CreateFile بازگردانده شود. شی جدید ایجاد شده از نوع اشیاء اجرایی است و در فضای نام Object Manager قرار نمی‌گیرد. برای دیدن چنین اشیایی می‌توانید از برنامهٔ Process Explorer استفاده کنید.



File Object



File object properties

برقراری ارتباط با Device ها

```
.if eax != INVALID_HANDLE_VALUE
    mov hDevice, eax
```

اگر تابع CreateFile شماره دسترسی معتبری را بازگرداند، آنرا در متغیر hDevice ذخیره می‌کنیم. حال می‌توانیم با استفاده از توابع ReadFile، WriteFile و DeviceIoControl با این Device ارتباط برقرار کنیم. تابع DeviceIoControl یکی از توابع عمومی برقراری ارتباط با Device ها است. در زیر، پیش تعریف آنرا مشاهده می‌کنید.

```
DeviceIoControl proto stdcall hDevice          :HANDLE,\
                              dwIoControlCode  :DWORD,\
                              lpInBuffer       :LPVOID,\
                              nInBufferSize    :DWORD,\
                              lpOutBuffer      :LPVOID,\
                              nOutBufferSize   :DWORD,\
                              lpBytesReturned  :LPVOID,\
                              lpOverlapped     :LPVOID
```

تابع DeviceIoControl پارامترهای بیشتری از CreateFile می‌پذیرد که در ادامه آنها را مورد بررسی قرار خواهیم داد.

hDevice : شماره دسترسی به Device .

dwIoControlCode : کد کنترلی که نشان می‌دهد چه عملیاتی باید انجام شود.

lPInBuffer : اشاره‌گر به بافری که محتوی اطلاعات مورد نیاز برای انجام عملیات است. این پارامتر زمانی می‌تواند Null باشد که عملیات مشخص شده توسط پارامتر dwIoControlCode نیاز به هیچ دادهٔ ورودی نداشته باشد.

nInBuffer Size : اندازه بافر را به بایت مشخص می‌کند که توسط lpInBuffer به آن اشاره می‌شود.

lpOutBuffer : اشاره‌گربه بافری که داده‌های خروجی عملیات‌مشخص شده dwIoControlCode را دریافت می‌کند. این پارامتر زمانی می‌تواند Null باشد که عملیات مشخص شده توسط dwIoControlCode هیچ دادهٔ خروجی تولید نکند.

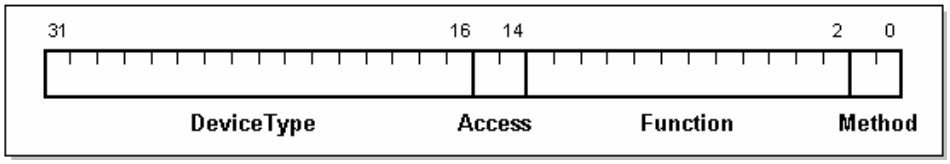
nOutBufferSize : اندازه بافر lpoutBuffer را بر حسب بایت مشخص می‌کند.

lpBytesReturned : اشاره‌گر به متغیری است که اندازه داده‌هایی را که در بافر ذخیره می‌شود به بایت دریافت می‌کند.

IpOerverlapped : اشاره‌گر به رکورد Overlapped . این رکورد برای کمک به کنترل یک عملیات آسنکرون موردنیاز است. این مقدار را Null قرار می‌دهیم. درنتیجه تابع DeviceIoControl تا زمانی که روال درایور به پایان نرسد باز نخواهد گشت.

کدهای کنترلی I/O

Device Driver را می‌توان به عنوان یک بسته از توابع مد هسته در نظر گرفت. کدهای کنترلی I/O معین می‌کنند که چه تابعی صدا زده خواهد شد. آرگومان dwIoControlCode از تابع DeviceIoControl برای این منظور به کار گرفته شده است. این آرگومان عملیات کنترلی مورد نظر و چگونگی اجرای آن را مشخص می‌کند. کد کنترلی موردنظر یک عدد ۳۲ بیتی است که توسط ماکروی CTL_CODE تعریف می‌شود این ماکرو در فایل‌های Winioctl.inc و Ntddk.inc قرار دارد.



فرمت خروجی I/O Control Code

Device Type : ۱۶ بیت نهایی مربوط به نوع Device است . این ۱۶ بیت نوع Device ای که این عملیات کنترلی را پیاده‌سازی می‌کند مشخص می‌کند. مقادیر 0-7FFFH توسط شرکت ماکروسافت رزرو شده است. مقادیر 8000H-0FFFH آماده برای توسعه‌دهندگان انواع جدید درایورهای مد هسته است. در فایل \include\w2k\ntddk.inc می‌توانید یک سری از ثابت‌های نمادین FILE_DEVICE_XXX را بیابید. این‌ها همان مقادیر رزرو شده ماکروسافت هستند. در این جا ما از مقدار FILE_DEVICE_UNKNOWN استفاده می‌کنیم. شما هم می‌توانید یک FILE_DEVICE_XXX دیگر تعریف کنید.

Access : این بخش دو بیت است. این دو بیت سطوح دسترسی مورد نیاز برنامهٔ استفاده‌کننده را تعیین می‌کند. از آنجایی که این بخش از دو بیت تشکیل شده است. چهار حالت ممکن داریم که به بیان آنها می‌پردازیم.

FILE_ANY_ACCESS : بیشترین سطح دسترسی. در این حالت درایور عملیات درخواست شده توسط هر فراخواننده‌ای که شماره دسترسی به Device آنرا دارد را انجام می‌دهد.

FILE_READ_ACCESS : سطح دسترسی خواندنی. با این سطح دسترسی درایور Device اطلاعات را از Device به بافر حافظه منتقل می‌کند.

FILE_WRITE_ACCESS : سطح دسترسی نوشتنی. با این سطح دسترسی درایور اطلاعات را از بافر حافظه به Device منتقل می‌کند.

FILE-READ_ACCESS or FILE_WRITE-ACCESS : سطح دسترسی خواندنی و نوشتنی. با این سطح دسترسی درایور Device اطلاعات را بین بافر حافظه و Device جابه‌جا می‌کند.

Function : این بخش ۱۲ بیت است و مشخص می‌کند که دقیقاً چه عملیاتی باید انجام شود. مقادیر بین 800H-0FFH برای کدهای کنترلی خصوصی I/O هستند و مقادیر بین 0-7FFH توسط مایکروسافت برای کدهای کنترلی عمومی I/O رزرو شده اند.

Method : این بخش ۲ بیت است. این دو بیت چگونگی رفتار I/O manager با بافری را که برنامه تدارک دیده مشخص می‌کند. چون این دو بخش دو بیت است پس ٤ حالت ممکن برای ما به وجود می‌آورد که برخی از آنها را مورد بررسی قرار می‌دهیم.

METHOD_BUFFERED (0) : I/O بافر شده.

METHOD_IN_DIRECT (1) : I/O مستقیم.

روش بافر شده کاملاً امن است زیرا خود سیستم درباره سرباره عملیات کپی در حافظه و چگونگی رفتار کردن با بافر تصمیم‌گیری می‌کند. اما درایورها معمولاً زمانی که درخواست فراخوانندهٔ آنها کوچکتر از یک صفحه (4kB) باشد از این روش استفاده می‌کنند زیرا سرباره سیستم در زمانی که بافر کوچک باشد کمتر از روش I/O مستقیم است. در درایور VirToPhys از روش بافر شده استفاده می‌کنیم. شما می‌توانید کد کنترلی I/O را به‌صورت دستی تشکیل دهید اما راحت‌تر آن است که از ماکروی CTL_CODE استفاده کنید. در زیر، روش استفاده از آنرا مشاهده می‌کنید.

```
CTL_CODE MACRO DeviceType:=<0>,\
 Function:=<0>,\
 Method:=<0>,\
```

```
 Access:=<0>
    EXITM %(((DeviceType) SHL 16) OR \
((Access) SHL 14) OR \
((Function) SHL 2) OR (Method))
ENDM
```

همان‌طور که قبلاً گفتیم ماکروی CTL_CODE در فایل .Winioctl قرار داده شده است.

```
NUM_DATA_ENTRY         equ 4
DATA_SIZE              equ (sizeof DWORD) * NUM_DATA_ENTRY
IOCTL_GET_PHYS_ADDRESS equ CTL_CODE(FILE_DEVICE_UNKNOWN,\
 800h, METHOD_BUFFERED,\
 FILE_READ_ACCESS + FILE_WRITE_ACCESS)
```

تبادل داده‌ها

حال به کد درایور باز می‌گردیم.

```
                    lea esi, adwInBuffer
                    assume esi:ptr DWORD
                    invoke GetModuleHandle, NULL
                    mov [esi][0*(sizeof DWORD)], eax
                    invoke GetModuleHandle, $CTA0("kernel32.dll",
szKernel32)
                    mov [esi][1*(sizeof DWORD)], eax
                    invoke GetModuleHandle, $CTA0("user32.dll",
szUser32)
                    mov [esi][2*(sizeof DWORD)], eax
                    invoke GetModuleHandle, $CTA0("advapi32.dll",
szAdvapi32)
                    mov [esi][3*(sizeof DWORD)], eax
```

در این جا بافر adwInBuffer را با آدرس مجازی که تبدیل خواهد شد پر می‌کنیم.

```
lea edi, adwOutBuffer
assume edi:ptr DWORD
invoke DeviceIoControl, hDevice,\
IOCTL_GET_PHYS_ADDRESS,\
esi,\
sizeof adwInBuffer,\
edi,\
```

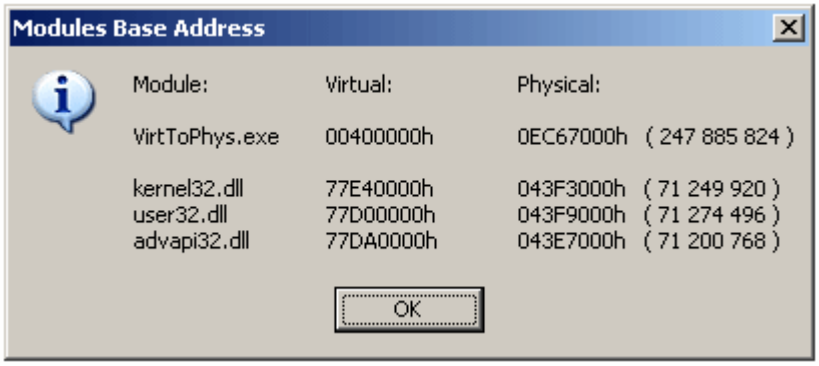
```
sizeof adwOutBuffer,\
addr dwBytesReturned,\
NULL
```

با فراخوانی تابع DeviceIoControl ، بافر را به درایور می‌فرستیم. درایور هر آدرس مجازی را به آدرس فیزیکی تبدیل می‌کند.

```
                    .if ( eax != 0 ) && ( dwBytesReturned != 0 )

                        invoke GetModuleFileName, [esi][0*(sizeof
DWORD)],\
addr acModulePath,\
sizeof acModulePath
                        lea ecx, acModulePath[eax-5]
                        .repeat
                            dec ecx
                            mov al, [ecx]
                        .until al == '\'
                        inc ecx
                        push ecx

                        CTA0 "%s \t%08Xh\t%08Xh   ( %s )\n",
szFmtMod

                        invoke BigNumToString, [edi][0*(sizeof
DWORD)],\
 addr acNumber
                        pop ecx
                        invoke wsprintf, addr acThis,\
                                         addr szFmtMod,\
                                         ecx,\
                                         [esi][0*(sizeof DWORD)], \
                                         [edi][0*(sizeof DWORD)],\
                                         addr acNumber

                        invoke BigNumToString, [edi][1*(sizeof
DWORD)],\
 addr acNumber
                        invoke wsprintf, addr acKernel,\
                                         addr szFmtMod,\
                                         addr szKernel32, \
                                         [esi][1*(sizeof DWORD)],\
                                         [edi][1*(sizeof DWORD)],\
                                         addr acNumber
                        invoke BigNumToString, [edi][2*(sizeof
DWORD)],\
 addr acNumber
                        invoke wsprintf, addr acUser,\
                                         addr szFmtMod,\
```

```
                                        addr szUser32,\
                                        [esi][2*(sizeof DWORD)],\
                                        [edi][2*(sizeof DWORD)],\
                                        addr acNumber
                        invoke BigNumToString, [edi][3*(sizeof
DWORD)],\
 addr acNumber
                        invoke wsprintf, addr acAdvapi,\
                                        addr szFmtMod,\
                                        addr szAdvapi32,\
                                        [esi][3*(sizeof DWORD)],\
                                        [edi][3*(sizeof DWORD)],\
                                        addr acNumber
                        invoke wsprintf, addr acBuffer,\
$CTA0("Module:\t\tVirtual:\t\tPhysical:\n\n%s\n%s%s%s"),\
 addr acThis,\
 addr acKernel,\
 addr acUser,\
 addr acAdvapi

                        assume esi:nothing
                        assume edi:nothing
                        invoke MessageBox, NULL,\
                                     addr acBuffer,\
                                     $CTA0("Modules Base
                                    Address"),\
                                     MB_OK + MB_ICONINFORMATION
                    .else
                        invoke MessageBox, NULL,\
 $CTA0("Can't send control code to device."),\
 NULL, \
 MB_OK + MB_ICONSTOP
                    .endif
```

اگر تابع DeviceIoControl با موفقیت بازگشت ، مقدار متغیر dwBytesReturned برابر با تعداد بایت‌هایی است که در متغیر adwOutBuffer قرار دارد. این متغیر توسط درایور مقداردهی می‌شود. از این جا به بعد کار ما ساده است. حال باید داده بدست آمده را فرمت‌بندی کرده و به کاربر نشان دهیم.



خروجی برنامهٔ *VirToPhys.exe*

```
invoke CloseHandle, hDevice
```

در این بخش کار ما این است که شماره‌های دسترسی باز را ببندیم. برای انجام این کار، I/O Manager دو IRP به درایور Device می‌فرستد. اولین IRP ، IRP_MJ_CLEANUP است یعنی شماره دسترسی Device آماده بسته شدن است. IRP دوم IRP_MJ_CLOSE است یعنی شماره دسترسی Device بسته شده است. البته با بازگرداندن کد خطا از روالی که مسئول بررسی IRP_MJ_CLEAN UP است، می‌توان از بسته شدن شماره دسترسی جلوگیری کرد.

ضمیمه

مجموعه دستورالعمل‌های 80x86

ضمیمه

مجموعه دستورات 80x86

این ضمیمه حاوی مجموعه دستورات به ترتیب حروف الفبایی است، اگرچه دستورات مرتبط با یکدیگر برای سهولت بیشتر، گروه‌بندی شده اند.

- adder آدرس موقعیت حافظه
- addr – high بایت سمت راست یک آدرس
- addr – low بایت سمت چپ یک آدرس
- data عملوند بلافصل (اگر w = 0 ، ۸ بیت اگر w = 1 ، ١٦ بیت)
- data-high بایت سمت راست یک عملوند بلافصل
- data-low بایت سمت چپ یک عملوند بلافصل
- disp جابجایی (مقدار افست)
- reg ارجاع به یک حافظه

پردازشگرهای ٨٠٢٨٦ و ما بعد تعدادی دستور خاص را پشتیبانی می‌کنند که در این جا آورده نشده است.

ARPL، BOUND، CLTS، ENTER، LAR، LEAVE، LGDT، LIDT، LLDT، LMSW، LSL، LTR، SGDT، SIDT، SLDT، SMSW، STR، VERR، VERW دستورات منحصر به فرد 80486 و ما بعد BSWAP ، INVD، WBINVD و INVLPG نیز در این جا آورده نشده‌اند.

خلاصه‌ای برای پرچم ها نیز چنین است : AF = معین، CF = نقلی، DF = جهت، IF = وقفه، OF = سر زیر، PF = توازن، SF = علامت، IF = اجرای مرحله‌ای و ZF = صفر.

AAA : تنظیم ASCII بعد از جمع

عمل. مجموع دو بایت ASCII را در ثبات AL اصلاح می کند. اگر سمت راست ترین چهار بیت ثبات AL دارای مقداری بزرگتر از ۹ باشد یا پرچم AF دارای مقدار ۱ باشد، AAA یک واحد به AH می‌افزاید و پرچم‌های CF , AF را با ۱ مقدار می‌دهد. دستور همواره سمت چپ ترین چهار بیت ثبات AL را با صفر پر می‌کند.

کد منبع . AAA (بدون عملوند)

AAD : تنظیم ASCII قبل از تقسیم

عمل. یک مقدار BCD غیر فشرده (مقسوم) در AX را قبل از تقسیم تنظیم می‌کند. AH, AAD را در ۱۰ ضرب می‌کند، حاصل را با AL جمع می‌کند و AH را پاک می‌کند حال مقدار دودویی در AX مساوی با مقدار BCD غیر فشرده اصلی است و برای یک عمل تقسیم آماده است.

کد منبع. AAD (بدون عملوند).

AAM: تنظیم ASCII بعد از ضرب

عمل. حاصل تولید شده در AL توسط MUL برای ضرب دو رقم BCD غیر فشرده را تنظیم می‌کند. AAM , AL را بر ۱۰ تقسیم می کند و خارج قسمت را در AH و باقی مانده را در AL ذخیره می‌کند.

کد منبع. AAM (بدون عملوند)

AAS : تنظیم ASCII بعد از تفریق

عمل. بعد از تفریق (بعد از SUB) دو بایت ASCII و AL را تنظیم می‌کند. اگر مقدار چهار بیت سمت راست بزرگتر از ۹ باشد، یا اگر CF یک باشد، AAS از AL ٦ را کم می‌کند یک را از AH تفریق می‌کند و CF, AF را یک می‌کند.در غیر این صورت CF, AF صفر می‌شوند. AAS همیشه چهار بیت سمت چپ را از AL پاک می‌کند.

کد منبع . AAS (بدون عملوند)

ADC : جمع با نقلی

عمل. معمولاً در جمع دودویی چند کلمه‌ای برای انتقال یک بیت ۱ سرریز شده به مرحله بعدی محاسبات استفاده می‌شود. ADC محتویات CF (۰/۱) را با اولین عملوند جمع می‌کند و دومین عملوند را با اولین عملوند مانند ADD جمع می‌کند. (SBB را نیز ببینید).

کد منبع. ADC register / memory , register / memory /immediate

ADD : جمع اعداد دو دویی

عمل. مقادیر دو دویی را از حافظه، ثبات، یا مقدار بلافصل با یک ثبات جمع می‌کند یا مقادیر یک ثبات یا بلافصل را با حافظه جمع می‌کند مقادیر باید بایت، کلمه یا دو کلمه‌ای باشند (۸۰۳۸٦ و ما بعد)

کد منبع. ADD register / memory , register / memory / immediate

AND : AND منطقی

عمل. روی بیت‌های دو عملوند یک عمل AND منطقی انجام می‌دهد. دو عملوند، هر دو بایت یا هر دو بایت یا هر دو کلمه، یا هر دو دو کلمه‌ای (در ۸۰۳۸٦ و ما بعد) هستند که دستور AND دو عملوند را بیت به بیت با یکدیگر تطبیق می‌دهد.

برای هر دو بیت ۱، یک بیت در اولین عملوند ۱ خواهد شد، در غیر این صورت بیت صفر خواهد شد. (OR ، XOR و TEST را نیز ببینید.)

کد منبع. AND register / memory , register / memory / immediate

BSR/BSF : پیمایش بیت به سمت جلو / پیمایش بیت معکوس (۸۰۳۸۶ و ما بعد)

عمل. یک رشته را برای یافتن اولین بیت ۱ پیمایش می‌کند. BSF را از راست به چپ پیمایش می‌کند، و ESR از چپ به راست پیمایش می‌کند. دومین عملوند (۱٦ یا ۳۲ بیت) شامل رشته‌ای است که باید پیمایش شود. اگر یک بیت ۱ پیدا شود، عمل موقعیت آن را در اولین ثبات عملوند باز می‌گرداند و ZF را یک می‌کند، در غیر این صورت ZF صفر می‌شود.

کد منبع. BSF / BSR register , register / memory

BT / BTC / BTR/ BTS : چک کردن بیت (۸۰۳۸۶ و ما بعد)

عمل. یک بیت مخصوص را در CF کپی می‌کند. اولین عملوند شامل رشته‌های بیت است که بررسی می‌شود و دومین عملوند شامل مقداری است که بر موقعیت آن دلالت دارد BT بیت را در CF کپی می‌کند. دیگر دستورات نیز بیت را کپی می‌کنند اما بر روی آن با این روش عمل می‌کند: BTC بیت را با معکوس کردن مقدار آن در اولین عملوند، مکمل می‌کند. BTR بیت را با صفر کردن مجدداً تنظیم می‌کند، BTS بیت را با یک تنظیم می‌کند ارجاع به مقادیر ۱۶ و ۳۲ بیتی خواهد بود.

کد منبع. BT / BTC / BTR / BTC register / memory , register / immediate

CALL : فراخوانی یک روال

عمل. یک روال دور یا نزدیک را فرا می‌خواند. اسمبلر یک CALL نزدیک تولید می‌کند. اگر روال فراخوانی شده NEAR باشد و اگر روال فراخوانی شده FAR باشد یک CALL دور تولید می‌کند. یک CALL نزدیک IP را بر روی پشته می‌گذارد (آدرس دستور بعدی)، سپس در IP آدرس افست مقصد را قرار می‌دهد. یک CALL دور CS را روی پشته قرار می‌دهد و یک اشاره گر بین سگمنت‌ها روی پشته می‌گذارد، سپس IP روی پشته قرار می‌دهد و در IP آدرس افست مقصد را قرار می‌دهد. در بازگشت، RETN یا RETF عکس مراحل فوق را انجام می‌دهد.

کد منبع. CALL register / memory

CBW : تبدیل بایت به کلمه

عمل. یک بایت مقدار علامت دار را به یک کلمه علامت دار بسط می‌دهد بدین وسله که علامت (بیت ۷)در AL را در بیت‌های AH کپی می‌کند (CDQ , VWDE , CWD را نیز ببینید.)

کد منبع. CBW (بدون عملوند)

CDQ : تبدیل دو کلمه‌ای به چهار کلمه‌ای (۸۰۳۸۶ و ما بعد)

عمل. یک مقدار علامت دار ۳۲ بیت را در یک مقدار علامت دار ٦٤ بیت بسط می‌دهد بدین وسیله که علامت (بیت ۳۱) EAX را در EDX کپی می‌کند. (CWDE , CWD , CBW را نیز ببینید.)

کد منبع. CDQ (بدون عملوند)

CLC : صفر کردن پرچم نقلی

عمل. CF را صفر می‌کند. برای مثال ADC یک بیت ۱ را جمع نمی‌کند. (STC را نیز ببینید).

کد منبع. CLC (بدون عملوند)

CLD : صفر کردن پرچم جهت

عمل. DF را صفر می‌کند، تا سبب شود عملیات رشته‌ای مانند MOVS پردازش را از چپ به راست انجام دهند. (STD را نیز ببینید).

کد منبع . CLD بدون عملوند

CLI : صفر کردن پرچم وقفه

عمل. IF را صفر می کند، وقفه‌های خارجی قابل پوشش را غیر فعال می‌کند (STI را نیز ببینید).

کد منبع. CLI (بدون عملوند)

CMC : مکمل کردن پرچم نقلی

عمل. CF را مکمل می‌کند. مقدار بیت CF معکوس خواهد شد بنابراین صفر تبدیل به یک و یک تبدیل به صفر خواهد شد.

کد منبع . CMC (بدون عملوند)

CMP : مقایسه

عمل. محتویات دو دویی دو فیلد داده را مقایسه می‌کند. CMP به‌طور داخلی دومین عملوند را از اولین کم می‌کند و پرچم صفر یا یک می‌کند، اما حاصل را ذخیره نمی‌کند. هر دو عملوند باید بایت، کلمه، یا دو کلمه‌ای (۸۰۳۸۶ و مابعد) باشند. CMP امکان مقایسه ثبات، حافظه، یا مقدار بلافصل را با یک ثبات یا امکان مقایسه ثبات یا بلافصل را با حافظه، دارد. (CMP مقایسه عددی انجام می‌دهد، CMPS را برای مقایسه رشته‌ای ببینید).

کد منبع. CMP register / memory , register / memory / immediate

CMDSD / CMPSW / CMPSB / CMPS : مقایسه رشته ای

عمل. رشته‌هایی با هر طول در حافظه را مقایسه می‌کند. معمولاً یک پیشوند RFPn قبل از این دستورات قرار می‌گیرد، با حداکثر طول که در CX است. CMPSB بایت‌ها را مقایسه می‌کند، CMPSW کلمات را مقایسه می‌کند و CMPSD (۸۰۳۸٦ و ما بعد) دو کلمه‌ای‌ها را مقایسه می‌کند. آدرس اولین عملوند در DS:SI و آدرس دومین عملوند در ES:DI قرار دارد. اگر DF صفر باشد، عملیات از چپ به راست مقایسه می‌کند و SI , DI را برای هر بایت یک واحد ، برای هر کلمه و واحد و برای هر دو کلمه ٤ واحد می‌افزاید ، اگر DF یک باشد، عملیات از راست به چپ مقایسه می‌کند و SI , DI را کاهی می‌دهد. REPn برای هر بار تکرار CX را یک واحد می‌کاهد. REPNE هنگامی که اولین مورد موافق پیدا شد خاتمه می‌یابد، REPE هنگامی که اولین مورد مخالف پیدا شد خاتمه می‌یابد، و هر دو زمانی که CX صفر شود نیز خاتمه می‌یابند، SI , DI بعد از بایت‌هایی که سبب خاتمه شدند قرار می‌گیرند. آخرین مقایسه پرچم‌ها را تنظیم می‌کند.

کد منبع. (بدون عملوند) CMPSB / CMPSW / CMPSD [REPNN]

CMPXCHG : مقایسه و تعویض (۸۰۳۸۶ و ما بعد)

عمل. دومین عملوند (AL , AX یا EAX)را با اولین عملوند (ثبات یا حافظه) مقایسه می‌کند. اگر مساوی باشند، CMPXCHG دومین عملوند را قرار می‌دهد و ZF را یک می‌کند، اگر مساوی نباشند اولین عملوند را در دومین عملوند قرار می‌دهد و ZF را صفر می‌کند.

کد منبع . CMPXCHG register / memory , AL / AX / EAX

CMPXCHG8B : مقایسه و تعویض (پنتیوم و مابعد)

عمل. ٦٤ بیت EDX: EAX را با اولین عملوند (ثبات یا حافظه) مقایسه می‌کند. اگر مساوی باشند، CMPXCHG8B ، EDX:EAX را در اولین عملوند قرار می‌دهد و ZF را یک می‌کند، اگر مساوی نباشند، CMPXCHG8B. اولین عملوند را در EDX:EAX قرار می‌دهد و ZF را صفر می‌کند.

کد منبع. (یک عملوند ٦٤ بیتی) CMPXCH8B register / memory

CWD : تبدیل کلمه به دو کلمه ای

عمل. یک مقدار یک کلمه‌ای علامت دار را در DX:AX دو کلمه ای علامتدار بسط می‌دهد بدین‌وسیله که بیت علامت (بیت ۱۵) AX را در AX کپی می:ند تا ۳۲ بیت تولید شود. (CDQ, CWDE , CBW را نیز ببینید).

کد منبع. CWD (بدون عملوند).

CWDE : تبدیل کلمه به دو کلمه ای بسط یافته (۸۰۳۸۶ و ما بعد)

عمل. یک مقدار علامت‌دار یک کلمه‌ای را در یک دو کلمه ای EAX بسط می‌دهد بدین‌وسیله که بیت علامت (بیت ۱۵) AX را کپی می‌کند و یک مقسوم ۳۲ بیتی تولید می کند. (CDQ , CWD, CBW را نیز ببینید).

کد منبع. CWDE (بدون عملوند)

DAA : تنظیم دهدهی بعد از جمع

عمل. حاصل موجود را بعد از ADD یا ADC که دو رقم BCD را جمع کرده است، تصحیح می‌کند. اگر مقدار چهار بیت سمت راست بزرگتر از ۹ باشد، یا اگر AF یک باشد، DAA ، ٦ را با AL جمع می‌کند و AF را یک می‌کند. سپس، اگر مقدار AL بزرگتر از 99H باشد، یا اگر CF یک باشد، DAA مقدار 60H را با AL جمع می‌کند و CF را یک می‌کند. در غیر این صورت AF , CF صفر می‌شوند. حال AL شامل یک حاصل ده دهی فشرده ۲ رقمی صحیح است. (DAS را نیز ببینید).

کد منبع. DAA (بدون عملوند)

DAS : تنظیم دهدهی برای تفریق

عمل. بعد از یک SUB یا SBB که دو عنصر BCD فشرده را تفریق می‌کند، حاصل تولید شده در AL را تصحیح می‌کند. اگر مقدار بیت سمت راست بزرگتر از ۹ باشد، DAS مقدار 60H را از AL کم می‌کند و CF را یک می‌کند، در غیر این صورت AF , CF صفر می‌شوند. حال AL شامل حاصل دو رقمی فشرده تصحیح شده است (DAA را نیز ببنید).

کد منبع. DAS (بدون عملوند).

DEC : یک واحد کاهش

عمل. یک را از بایت، کلمه، یا دو کلمه‌ای (در ۸۰۳۸٦ و ما بعد) موجود در ثبات حافظه می‌کاهد و با مقدار موجود در آن به‌صورت یک عدد صحیح بدون علامت برخورد می‌کند. (INC را نیز ببینید.)

کد منبع. DEC register / memory

DIV : تقسیم بدون علامت

عمل. یک مقسوم بدون علامت را بر یک مقسوم‌علیه علامت‌دار تقسیم می‌کند. DIV سمت چپ‌ترین بیت را به‌عنوان بیت دارد. در نظر می‌گیرد و نه به عنوان یک علامت منفی. تقسیم بر صفر سبب ایجاد وقفه تقسیم بر صفر خواهد شد.

(IDIV را نیز ببینید). در این جا عملیات تقسیم بر طبق اندازه مقسوم ذکر شده است:

Size	Dividend (Operand 1)	Divisor (Operand 2)	Quotient	Remainder	Example
16-bit	AX	8-bit reg/memory	AL	AH	DIV BH
32-bit	DX:AX	16-bit reg/memory	AX	DX	DIV CX
64-bit	EDX:EAX	32-bit reg/memory	EAX	EDX	DIV ECX

کد منبع. DIV register / memory

ESC : گریز

عمل. استفاده از کمک پردازشگرهایی مانند 87 × 80را جهت انجام عملیات ویژه سهولت می‌بخشد. ESC، کمک پردازشگر با یک دستور و عملوند را جهت اجرا مهیا می‌سازد. توجه کنید که نسخه MASM, 6.1 دیگر ESC را پشتیبانی نمی‌کند، به جای آن کد مقصد کامل موردنیاز برای دستورات کمک پردازشگر را تولید می‌کنند.

کد منبع. ESC immediate , register / memory

HLT : وارد کردن حالت توقف

عمل. سبب می‌شود زمانی که پردازشگر منتظر یک وقفه است به حالت توقف برود، هم اکنون ثبات‌های IP, CS به آدرس دستور بلافاصله بعد اشاره می‌کند. وقتی یک وقفه رخ دهد، پردازشگر IP, CS را برروی پشته می‌گذارد و روتین وقفه را اجرا می‌کند. در بازگشت یک دستور IRET، از روی پشته ثبات‌ها را برمی‌دارد و پردزاش از HLT اصلی آغاز می‌شود.

کد منبع. HLT (بدون عملوند)

IDIV : تقسیم علامتدار (صحیح)

عمل. یک مقسوم علامت‌دار را بر یک مقسوم علیه علامتدار تقسیم می‌کند. IDIV با سمت چپ‌ترین بیت به‌عنوان علامت برخورد می‌کند (۰ = مثبت، ۱ = منفی). تقسیم بر صفر سبب وقفه تقسیم بر صفر خواهد شد. (CBW , CWD را برای بسط طول مقسوم علامت‌دار و هم چنین DIV را نیز ملاحظه کنید). در این جا عملیات تقسیم بر طبق اندازه مقسوم ذکر شده است.

Size	Dividend (Operand 1)	Divisor (Operand 2)	Quotient	Remainder	Example
16-bit	AX	8-bit reg/memory	AL	AH	IDIV BH
32-bit	DX:AX	16-bit reg/memory	AX	DX	IDIV CX
64-bit	EDX:EAX	32-bit reg/memory	EAX	EDX	IDIV ECX

کد منبع. IDIV register / memory

IMUL : ضرب علامت‌دار (صحیح)

عمل. یک مضروب علامت‌دار را در یک مضروب فیه علامتدار ضرب می‌کند. IMUL با سمت چپ‌ترین بیت مانند بیت علامت برخورد می‌کند (۰ = مثبت و ۱ = منفی) . این عملیات در نظر می‌گیرد که مضروب در EAX , AX , AL است و این اندازه بر حسب مضروب فیه به دست می‌آید. (MUL را نیز ببینید). در این جا عملیات ضرب بر طبق اندازه مضروب فیه آورده شده است.

Size	Multiplicand (Operand 1)	Multiplier (Operand 2)	Product	Example
8-bit	AL	8-bit register/memory	AX	IMUL BL
16-bit	AX	16-bit register/memory	DX:AX	IMUL BX
32-bit	EAX	32-bit register/memory	EDX:EAX	IMUL ECX

کد منبع. (در همه پردازشگرها) IMUL register / memory

IN : وارد کردن بایت یا کلمه

عمل. انتقال از یک درگاه ورودی به AL یا یک کلمه به AX . درگاه را به‌صورت یک عملوند عددی ثابت (IN AX, port#) یا متغییری در DX (به‌صورت IN AX, DX) کد نمایید. در صورتی از DX استفاده کنید که شماره درگاه بزرگتر از ۲۵۶ باشد. (INS و OUT را نیز ملاحظه کنید.)

کد منبع. IN AL , AX , Portno / DX

INC : یک واحد افزایش

عمل. یک بایت، یک کلمه، یا دو کلمه‌ای (در ۸۰۳۸۶ و ما بعد) موجود در ثبات یا حافظه را یک واحد افزایش می‌دهد و با مقدار موجود در آن به‌عنوان یک عدد صحیح بدون علامت برخورد می‌کند، برای مثال به‌صورت INC CX کد می‌شود (DEC را نیز ببینید).

کد منبع. INC register/ memory

INS / INSB / INSW / INSD وارد کردن رشته (در ۸۰۲۸۶ و ما بعد)

عمل. یک رشته را از درگاه (مقصد) دریافت می‌کند. مقصد توسط ES:DI آدرس‌دهی می‌شود و DX حاوی شماره درگاه است. تجربه عملی استفاده از INSn با پیشوند REP است که CX حاوی تعداد عناصر دریافتی است (به‌صورت بایت، کلمه، یا دو کلمه‌ای). بسته به DF (0/1)، عملیات بر طبق اندازه عنصر DI را افزایش می‌دهد. (IN و OUTS را نیز ببینید.)

کد منبع. (بدون عملوند) [REP] INSB / INSW / INSD

INT : وقفه

عمل. پردازش را متوقف می‌کند و کنترل را به یکی از ۲۵٦ آدرس وقفه که از سگمنت 0، افست 0 آغاز می‌شود. منتقل می‌کند. INT چنین عمل می‌کند: (۱) پرچم‌ها را روی پشته می‌گذارد و TF , IF را یک می‌کند، (۲) CS را روی پشته می‌گذارد و کلمه بالا رتبه آدرس وقفه را در CS قرار می‌دهد (۳) IP را برروی پشته می‌گذارد و IP را با کلمه پایین رتبه آدرس وقفه پر می‌کند. برای ۸۰۳۸٦ و ما بعد INT یک IP ۱٦ بیتی برای سگمنت ۱٦ بیتی ویک IP ، ۳۲ بیتی برای سگمنت ۳۲ بیتی برروی پشته می‌گذارد. IRET از روتین وقفه باز می‌گردد.

کد منبع . INT number

INTO : وقفه در حالت سر ریز.

عمل. سبب یک وقفه خواهد شد اگر سرریز رخ داده باشد (OF یک باشد) و INT 44H را اجرا می‌کند. آدرس وقفه در موقعیت 10H از جدول سرویس وقفه قرار دارد. (INT را نیز ببینید).

کد منبع. (بدون عملوند) INTO

IRET / IRETD : بازگشت از وقفه

عمل. یک بازگشت دور را از یک روتین وقفه فراهم می‌سازد. IRET روال زیر را انجام می‌دهد: (۱) کلمه بالای پشته را برداشته و در IP می‌گذارد، SP را ۲ واحد می‌افزاید و کلمه بالای پشته را در CS قرار می‌دهد، (۲) SP را ۲ واحد می افزاید و کلمه بالای پشته را در پرچم‌ها می‌گذارد این روال عکس مراحل وقفه را برای بازگشت انجام می‌دهد. در ۸۰۳۸٦ و ما بعد از IRETD (دو کلمه‌ای) برای برداشتن IP ، ۳۲ بیتی از روی پشته استفاده کنید. RET (را نیز ببینید).

کد منبع. IRET (بدون عملوند)

Jcondition : پرش بر حسب شرط

این بخش، دستورات پرش شرطی را به‌طور خلاصه بیان می کند که در صورت پرچم شرط بررسی شده به عملوند منتقل می‌شود. اگر شرط صحیح باشد، عملیات افست عملوند را با IP جمع می‌کند و پرش را انجام می‌دهد، اگر شرط صحیح نباشد، پردازش از دستور بعدی ادامه می‌یابد. برای ۸۰۸٦ – ۸۰۲۸٦ پرش باید کوتاه باشد (۱۲۸ – ۱۲۷ بایت)، برای ۸۰۳۸٦ و ما بعد اسمبلر یک پرش نزدیک درنظر می‌گیرد (۳۲۷٦۸ تا ۳۲۷٦۷) اما شما باید عملگر SHORT را در این روش کوتاه استفاده کنید.

عملیات پرچم‌ها را بررسی می‌کند ولی آنها را تغییر نمی‌دهد. کد منبع Jconditonal Label می‌باشد. همه کدهای مقصد به‌صورت |---disp --- |distnnn| درحالی که بیت‌های disp برای پرش‌های کوتاه 0111 و برای پرش‌های نزدیک 1000 می‌باشد. در اولین لیست، دستورات، بعد از عملیات مقایسه که اولین عملوند و دومین را با هم مقایسه می‌کند، مورد استفاده قرار می‌گیرد.

SOURCE CODE	OBJECT CODE	FLAGS CHECKED	USED AFTER COMPARISON
JA	|dist0111|	CF = 0, ZF = 0	Unsigned data, above (higher)
JAE	|dist0011|	CF = 0	Unsigned data, above/equal
JB	|dist0010|	CF = 1	Unsigned data, below (lower)
JBE	|dist0110|	CF = 1 or AF = 1	Unsigned data, below/equal
JE	|dist0100|	ZF = 1	Signed/unsigned data, equal
JG	|dist1111|	ZF = 0, SF = OF	Signed data, greater
JGE	|dist1101|	SF = OF	Signed data, greater/equal
JL	|dist1100|	SF not= OF	Signed data, lower
JLE	|dist1110|	ZF = 1 or SF not= OF	Signed data, lower/equal
JNA	|dist0110|	CF = 1 or AF = 1	Unsigned data, not above
JNAE	|dist0010|	CF = 1	Unsigned data, not above/equal
JNB	|dist0011|	CF = 0	Unsigned data, not below
JNBE	|dist0111|	CF = 0, ZF = 0	Unsigned data, not below/equal
JNE	|dist0101|	ZF = 0	Signed/unsigned, not equal
JNG	|dist1110|	ZF = 1 or SF not= OF	Signed data, not greater
JNGE	|dist1100|	SF not= OF	Signed data, not greater/equal
JNL	|dist1101|	SF = OF	Signed data, not lower
JNLE	|dist1111|	ZF = 0, SF = OF	Signed data, not lower/equal

در لیست دوم دستورات بعد از محاسبه یا عملیات دیگری که بر طبق نتیجه بیت‌ها را صفر یا یک می‌کند، مورد استفاده قرار می‌گیرد.

SOURCE CODE	OBJECT CODE	FLAGS CHECKED	USED TO TEST
JC	|dist0010|	CF = 1	If CF set (same as JB/JNAE)
JNC	|dist0011|	CF = 0	If CF off (same as JAE/JNB)
JNO	|dist0001|	OF = 0	If OF off
JNP	|dist1011|	PF = 0	If no (odd) parity: odd number of bits set in low-order 8 bits
JNS	|dist1001|	SF = 0	If sign is positive
JNZ	|dist0101|	ZF = 0	If signed/unsigned data not zero
JO	|dist0000|	OF = 1	If OF set
JP	|dist1010|	PF = 1	If even parity: even number of bits set in low order 8 bits
JPE	|dist1010|	PF = 1	Same as JP
JPO	|dist1011|	PF = 0	Same as JNP
JS	|dist1000|	SF = 1	If sign is negative
JZ	|dist0100|	ZF = 1	If signed/unsigned data is zero

JCXZ / JECXZ : پرش در صورتی که CX / ECX صفر است

عمل. در صورتی که CX یا ECX (در ۸۰۳۸۶ و ما بعد) حاوی صفر است به آدرس خاص پرش می‌کند. این عملیات در شروع یک حلقه می‌تواند مفید باشد، اگرچه محدود به یک پرش کوتاه است.

کد منبع. JCXZ/JECXZ label

JMP : پرش غیر شرطی

عمل. بر آدرس معین شده در هر شرایطی پرش می‌کند. آدرس JMP ممکن است کوتاه (۱۲۸ – تا ۱۲۷+) نزدیک (داخل 32K± پیش‌فرض) یا دور (در سگمنت کد دیگری) باشد. یک JMP نزدیک یا کوتاه IP را با آدرس افست مقصد جایگزین می‌کند. یک پرش دور (مانند JMP FAR PTR Lable) CS:IP را با آدرس سگمنت جدید جایگزین می‌کند.

کد منبع. JMP register / memory

LAHF : قرار دادن پرچم‌ها در AH

عمل. ۸ بیت سمت راست ثبات پرچم را در AH قرار می‌دهد. (SAHFرا نیز ببینید).

کد منبع. LAHF (بدون عملوند)

LDS / LES / LFS / LGS / LSS : بارگذاری ثبات سگمنت

عمل. آدرس و افست دور یک عنصر داده را مقدار دهی می‌کند به‌طوری که دستور بعدی بتواند به آن دسترسی داشته باشد. اولین عملوند هر ثبات عمومی، شاخص یا اشاره‌گر را ارجاع می‌کند. دومین، عملوند چهار بایت حافظه شامل یک افست و یک آدرس سگمنت را ارجاع می‌کند. عملیات آدرس سگمنت را در ثبات سگمنت و آدرس افست را در ثبات اولین عملوند قرار می‌دهد. برای مثال LDS یعنی بارگذاری ثبات سگمنت داده. LSS , LGS , LFS توسط ۸۰۳۸۶ و ما بعد پشتیبانی می‌شوند.

کد منبع. LDS / LES / LFS / LGS / LSS register , memory

LEA :بارگذاری آدرس مؤثر

عمل. یک آدرس (افست) نزدیک را در یک ثبات قرار می‌دهد

کد منبع. LEA register , memory

LES / LFS / LGS :بارگذاری ثبات سگمنت اضافی

عمل. LDS را ببینید.

LOCK : قفل درگاه

عمل. 80X87 یا دیگر پردازشگرها را از تغییر یک عنصر داده در لحظه‌ای خاص باز می‌دارد. LOCK یک پیشوند یک بایتی است که ممکن است قبل از هر دستور کد نمایید. عملیات یک سیگنال به پردازشگر می‌فرستد تا از استفاده داده قبل از تکمیل دستور جاری جلوگیری کند.

کد منبع. LOCK instruction

LODS / LODSB / LODSW / LODSD : بارگذاری رشتهٔ بایت، کلمه یا دو کلمه‌ای

عمل. ثبات انباشتگر را با مقداری از حافظه بارگذاری می‌کند. اگرچه LODS یک عملیات رشته‌ای است، نیازی به پیشوند REP ندارد آدرس ثبات‌های DS:SI یک بایت (درصورت LODSB) یک کلمه (درصورت LODSW) یا دو کلمه‌ای (درصورت LODSD در ۸۰۳۸۶ و مابعد) هستند و

بترتیب از حافظه در AL , AX و EAX قرار می‌دهند. اگر DF صفر باشد، عملیات یک (بایت)، ۲ (کلمه) یا ٤ (دوکلمه‌ای) را با SI جمع می‌کند، درغیر این‌صورت ۱، ۲، ٤ را تفریق می‌کند.

کد منبع. (بدون عملوند LODSB/LODSW/LODSD)

LOOP / LOOPW / LOOPD : حلقه تا زمانی که کامل شود.

عمل. اجرای یک روتین را به تعداد مرتبه مشخص شده کنترل می‌کند. قبل از شروع حلقه، CX باید حاوی تعداد شمارش باشد. LOOP را در انتهای حلقه ظاهر می‌شود و CX را یک واحد می‌کاهد. اگر CX صفر نباشد، LOOP به آدرس عملوند (یک پرش کوتاه) منتقل می‌شود که به شروع حلقه اشاره دارد (افست با IP جمع می‌شود) در غیراینصورت به دستور بعدی می‌رود.

برای ۸۰۳۸٦ و مابعد ، LOOP در حالت ١٦ بایتی و از ECX در حالت ۳۲ بیتی استفاده می‌کند. برای مشخص کردن CX با ١٦ بیت از LOOPW و EXC با ۳۲ بیت از LOOPD استفاده کنید.

کد منبع. LOOPnnlable

LOOPE / LOOPZ / LOOPEW / LOOPZW / LOOPED / LOOPZD :

تکرار حلقه تا زمانی که مساوی یا صفر است.

عمل. تکرار اجرای یک روتین را کنترل می‌کند. LOOPE و LOOPZ مشابه LOOP است، بجز آنکه به آدرس عملوند (یک پرش کوتاه) در صورتیکه CX صفر نباشد و ZF یک باشد منتقل می‌شود (شرط صفر، با دیگر دستورات تنظیم می‌شود)، در غیراین صورت، عملیات به دستور بعدی می‌رود (LOOPNE/LOOPNZ را نیز ببینید).

برای ۸۰۳۸٦ و ما بعد، LOOPE و LOOPZ از CX در حالت ١٦ بیتی و از ECX در حالت ۳۲ بیتی استفاده کنید. می‌توانید از LOOPEW و LOOPZW برای ١٦ بیت CX و LOOPED و LOOPZD برای ۳۲ بیت ECX استفاده کنید.

کد منبع. LOOPnnlabel

LOOPNE / LOOPNZ / LOOPNEW / LOOPNZW : تکرار حلقه تا زمانی که مساوی نیست یا صفر نیست.

عمل. تکرار اجرای یک روتین را کنترل می‌کند. LOOPNE و LOOPNZ مشابه LOOP است بجز آنکه در صورتی‌که CX صفر نباشد و ZF صفر باشد به آدرس عملوند منتقل می‌شود (یک پرش کوتاه)، (شرط غیرصفر توسط دستورات دیگر تنظیم می‌شود). در غیراین‌صورت عملیات به دستور بعدی می‌رود (LOOPE/LOOPZ را نیز ببینید). برای ۸۰۳ و مابعد، LOOPNZ و LOOPNE از CX در حالت ۱۶ بیتی و از ECX در حالت ۳۲ بیتی استفاده می‌کند. می‌توانید از LOOPNZW و LOOPNEW برای مشخص کردن ۱۶ بیت CX و LOOPNZD و LOOPNED برای مشخص کردن ۳۲ بیت ECX استفاده کنید.

کد منبع. LOOPNE / LOOPNZ lable

LSS بارگذاری ثبات سگمنت پشته

عمل. LDS را ببینید.

MOV : انتقال داده

عمل. داده‌ها را بین دو ثبات یا بین یک ثبات و حافظه منتقل می‌کند و داده بلافصل را به یک ثبات یا حافظه منتقل می‌کند. داده ارجاع شده تعداد بایت‌های انتقالی (۱، ۲، ٤) را تعریف می‌کند، عملوندها باید هم اندازه باشند. MOV نمی‌تواند بین دو موقعیت حافظه، (از MOVS استفاده کنید)، یا از داده بلافصل به‌ثبات سگمنت، یا از ثبات سگمنت به ثبات سگمنت انتقال انجام دهد (MOVSX/MOVZX را نیز ببینید).

کد منبع. MOV register / memory , register / memory / immediate

MOVS / MOVSB / MOVSW / MOVSD : انتقال رشته

عمل. داده‌ها را بین موقعیت‌های حافظه منتقل می‌کند. عموماً با پیشوند REP استفاده می‌شود و در طول CX قرار می‌گیرد، MOVSB بایت‌ها را منتقل می‌کند، MOVSW کلمات را منتقل می‌کند و MOVSD دو کلمه‌ای را منتقل می‌کند (در ۸۰۳۸٦ و مابعد). اولین عملوند توسط ES:DI و دومین عملوند توسط DS:SI آدرس‌دهی می‌شود. اگر DF صفر باشد، عملیات، داده‌ها را از راست به چپ منتقل می‌کند و DI و SI را می‌کاهد. REP، CX را برای هر با تکرار می‌کاهد. عملیات وقتی CX صفر شود خاتمه می‌یابد، DI و SI بعد از آخرین انتقالی قرار دارند.

کد منبع. (بدون عملوند) [REP] MOVSB / MOVSW / MOVSD

MOVSX : MOVZX : انتقال یا بسط علامت یا بسط سفر (۸۰۳۸۶ و مابعد)

عمل. یک عملوند منبع ۸ یا ۱۶ بیتی را به عملوند مقصد ۱۶ یا ۳۲ بیتی کپی می‌کند. MOVSX بیت علامت را در بیت‌های سمت چپ پر می‌کند و MOVZX بیت‌ها را با صفر پر می‌کند.

کد منبع. MOVSX/MOVZX register / memory, register / memory / immediate

MUL : ضرب بدون علامت

عمل. یک مضروب بدون علامت را در مضروب فیه علامت‌دار ضرب می‌کند. MUL با سمت چپ‌ترین بیت به‌صورت بیت داده ، نه یک بیت علامت برخورد می‌کند. عملیات مضروب را در AX , AZ یا EAX فرض می‌کند و اندازه آن را از مضروب فیه در نظر می‌گیرد (LUML را نیز ببینید). در اینجا عملیات ضرب بر طبق اندازهٔ مضروب فیه آورده شده است.

Size	**Multiplicand (Operand 1)**	Multiplier **(Operand 2)**	**Product**	Example	
8-bit	AL	8-bit register/memory	AX	MUL	BL
16-bit	AX	16-bit register/memory	DX:AZ	MUL	BX
32-bit	EAX	32-bit register/memory	EDX:EAX	MUL	ECX

کد منبع. MUL register/memory

NEG : منفی کردن

عمل. یک مقدار دو دویی را از مثبت به منفی یا از منفی به مثبت معکوس می‌کند. NEG مکمل ۲ عملوند خاص را با تفریق عملوند از صفحه و جمع آن با یک مهیا می‌سازد. عملوندها باید یک بایت، کلمه یا دو کلمه‌ای (در ۸۰۳۸٦ و مابعد) در یک ثبات یا حافظه باشند(NOT را نیز ببینید).

کد منبع. NEG register memory

NOP : عملی انجام نده

عمل. برای حذف یا جایگزینی کد ماشین یا تأخیر اجرا برای زمان استفاده می‌شود. NOP یک عمل پوچ با اجرای XCHG AX , AX انجام می‌دهد.

کد منبع. NOP (بدون عملوند)

NOT : NOT منطقی

عمل. بیت‌های 0 را با یک و برعکس تعویض می‌کند. عملوند یک بایت، کلمه، یا دو کلمه‌ای (در ۸۰۳۸٦ و مابعد) در یک ثبات یا حافظه است(NEG را نیز ببینید).

کد منبع. NOT register/memory

OR : OR منطقی

عمل. یک عملیات OR منطقی برروی بیت‌های دو عملوند انجام می‌دهد. هر دو عملوند، بایت، کلمه، یا دو کلمه‌ای‌اند (در ۸۰۳۸٦ و مابعد) که OR بیت به بیت وفق می‌دهد. برای هر جفت بیت جور شده، اگر هر دو یا یکی از آنها ۱ باشد، بیت اولین عملوند یک خواهد شد، در غیر این صورت بیت تغییری نمی‌کند. (AND و XOR را نیز ببینید).

کد منبع. OR register / memory , register / memory / immediate

OUT : خروجی بایت یا کلمه

عمل. یک بایت را از AL یا کلمه را از AX به درگاه خروجی می‌فرستد. درگاه یک عملوند عددی ثابت یا یک متغیر در DX است. از DX هنگامی استفاده کنید که شمارهٔ درگاه بزرگتر از ۲۵٦ باشد(IN و OUTS را نیز ببینید).

کد منبع. درگاه ثابت : OUT port# , AX

درگاه متغیر: OUT DX , AX

OUTS / OUTSB / OUTSW / OUTSD : خروج رشته (۸۰۲۸۶ و مابعد)

عمل. یک رشته (منبع) را به درگاه ارسال می‌دارد. منبع توسط DS:SI آدرس‌دهی می‌شود و DX حاوی شماره درگاه است. تجربهٔ عملی استفاده از OUTSn با پیشوند REP است، با CX که حاوی تعداد عناصری (به‌صورت بایت، کلمه، یا دو کلمه‌ای) ارسالی است. بسته به DF(0/1) عملیات SI را طبق اندازه عنصر افزایش یا کاهش می‌دهد (IN و OUTS را نیز ببینید).

کد منبع. (بدون عملوند) [REP] OUTSB / OUTSW / OUTSD

POP : برداشتن کلمه از روی پشته

عمل. یک کلمه یا دو کلمه‌ای (در ۸۰۳۸٦ و مابعد) را که قبلاً روی پشته گذاشته شده برمی‌دارد و در یک مقصد خاص قرار می‌دهد – یک موقعیت حافظه ، ثبات عمومی، یا ثبات سگمنت. SP به کلمهٔ جاری بالای پشته اشاره دارد، POP آن را به مقصد خاص منتقل می‌کند و SP را ۲ واحد افزایش می‌دهد. در ۸۰۳۸٦ و مابعد ، یک عملوند ۳۲ بیتی بر یک مقدار دو کلمه‌ای دلالت دارد و ESP ٤ واحد افزایش می‌یابد (PUSH را نیز ببینید).

کد منبع. POP register / memory

POPA : (در ۸۰۲۸۶ و مابعد) POPAD : (در ۸۰۳۸۶ و مابعد)

همهٔ ثبات‌های عمومی را از روی پشته برمی‌دارد.

عمل. POPA ، هشت کلمه بالای پشته را برداشته و در DI , SI , BP , SP , BX , DX , CX و AX بترتیب قرار می‌دهد. POPAD هشت کلمه دو کلمه‌ای بالای پشته را برداشته و در EAX ,

ECX , EDX , ESP , ESI , EDI قرار می‌دهد. مقدار SP بعد از بارگذاری دور انداخته می‌شود. معمولاً ، یک PUSHA / PUSHAD قبلاً ثبات‌ها را بر روی پشته گذارده‌اند.

کد منبع. (بدون عملوند) POPA / POPAD

POPF/POPFD : برداشتن پرچم‌ها از روی پشته

عمل. POPF کلمه بالای پشته را برداشته و در ثبات پرچم قرار می‌دهد و SP را ۲ واحد می‌افزاید. POPFD (در ۸۰۳۸٦ و مابعد) دو کلمه‌ای بالای پشته را برداشته و در ثبات پرچم ۳۲ بیتی قرار می‌دهد و SP را ٤ واحد افزایش می‌دهد. معمولاً یک PUSHF قبلاً پرچم‌ها را روی پشته گذاشته‌اند.

کد منبع. POPF / POPFD (بدون عملوند)

PUSH : قراردادن پرچم‌ها از روی پشته

عمل. یک کلمه یا دو کلمه‌ای (در ۸۰۳۸٦ و مابعد) را جهت استفادهٔ بعدی برروی پشته می‌گذارد. ثبات SP به کلمه جاری بالای پشته اشاره می‌کند. PUSH و SP را ۲ واحد کاهش می‌دهد یا ESP را ٤ واحد کاهش می‌دهد و یک (دو) کلمه را از عملوند خاص به بالای جاری پشته منتقل می‌کند. منبع ممکن است یک ثبات عمومی ، ثبات سگمنت، یا حافظه باشد (POP و PUSHF را نیز ببینید).

کد منبع. (همه پردازشگرها) PUSH register / memory

(۸۰۲۸٦ و مابعد) PUSH immediate

PUSHA (۸۰۲۸٦ و مابعد) PUSHAD (۸۰۳۷٦ و مابعد) قراردادن همهٔ ثبات‌های عمومی برروی پشته

عمل. PUSHA همه ثبات‌های AX , CX , DX , BX , SP , BP , SI و DI را به ترتیب روی پشته قرار می‌دهد و SP را ۱٦ واحد کاهش می‌دهد. PUSHAD همهٔ ثبات‌های EAX , ECX , EDX , EBX , ESP , EBP , ESI , EDI را روی پشته قرار می‌دهد و SP را ۳۲ واحد می‌کاهد. معمولاً یک POPA / POPAP بعداً ثبات‌ها را برمی‌دارد.

کد منبع. PUSHA / PUSHAD (بدون عملوند)

PUSHF / PUSHFD : گذاشتن پرچم‌ها از روی پشته

عمل. محتویات ثبات پرچم را جهت استفادهٔ بعدی روی پشته می‌گذارد. PUSHF و SP را ۲ واحد می‌کاهد. PUSHFD (در ۸۰۳۸٦ و مابعد) ثبات پرچم ۳۲ بیتی را روی پشته می‌گذارد و SP را ٤ واحد می‌کاهد (POPF و PUSH را نیز ببینید).

کد منبع. PUSHF (بدون عملوند)

RCL / RCR : چرخش به چپ با رقم نقلی و چرخش به راست با رقم نقلی

عمل. بیت‌ها را در CF چرخش می‌دهد. عملیات بیت‌های یک بایت، کلمه یا دو کلمه‌ای (در ۸۰۳۸٦ و مابعد) را در یک ثبات یا حافظه به چپ یا راست می‌چرخاند. عملوند ممکن است یک ثابت بلافصل یا ارجاع به CL باشد. در 8088/86، ثابت فقط می‌تواند ۱ باشد و تعداد چرخش بیشتر در CL قرار می‌گیرد. در پردازشگرهای مابعد، مقدار ثبات تا ۳۱ نیز می‌تواند باشد. برای RCL سمت چپ‌ترین بیت وارد CF می‌شود و CF وارد بیت 0 مقصد می‌شود و همه دیگر بیت‌ها به سمت چپ می‌چرخند. برای RCR، بیت 0 وارد CF می‌شود و بیت CF وارد بیت سمت چپ مقصد می‌شود و همهٔ دیگر بیت‌ها به راست می‌چرخند (ROL و ROR را نیز ببینید).

کد منبع. RCL / RCR register / memory , CL/immediate

REP : تکرار رشته

عمل. یک عملیات رشته‌ای به تعداد خاص تکرار می‌شود. REP یک پیشوند تکرار انتخابی است که قبل از دستورات رشته‌ای MOVS , STOS , INS و OUTS کد می‌شود). قبل از اجرا در CX تعداد شمارش را قرار دهید. برای هر بار اجرای دستور رشته‌ای، REP و CX را یک واحد می‌کاهد و عملیات تا زمانی که CX صفر شود تکرار می‌شود و در اینجا پردازش با اجرای دستور بعدی ادامه می‌یابد. REPNE / REPNZ (REPE/REPZ را نیز ببینید).

کد منبع. REP string-instruction

REPE / REPZ / REPNE / REPNZ : تکرار مشروط رشته

عمل. یک عملیات رشته‌ای را به تعداد مرتبه خاصی تکرار می‌کند تا شرط خاصی برقرار شود. REPE , REPNE و REPNZ پیشوند تکرار انتخابی هستند که قبل از دستورات رشته‌ای SCAS و

CMPS که می‌شوند که این دستورات ZF را تغییر می‌دهند. در CX قبل از اجرا تعداد تکرار را قرار دهید. برای REPE / REPZ (تکرار تا مساوی/ صفر شدن)، عملیات تا زمانی که ZF یک شود (شرط صفر/مساوی) و CX صفر نباشد ادامه می‌یابد. برای REPNE/REPNZ (تکرار تا هنگامی که مساوی/صفر نیست)، عملیات تا زمانی که ZF صفر است (شرط نامساوی / غیرصفر) و CX صفر نیست، تکرار می‌شود. تا زمانی که شرط برقرار است، عملیات CX را یک واحد می‌کاهد و دستور رشته‌ای اجرا می‌شود.

کد منبع. REPNE/REPNZ: 1110010

REPE/REPZ : 1110011

RET/RETN/RETF : بازگشت از روال

عمل. از یک روال که قبلاً توسط یک CALL نزدیک یا دور آن وارد شده، باز می‌گردد. اسمبلر ، در صورتیکه روال برچسب NEAR داشته باشد، یک RET نزدیک تولید می‌کند و در صورتی‌که روال برچسب FAR داشته باشد یک RET دور تولید می‌کند. در حالت نزیک ، RET کلمه بالای پشته را به IP منتقل میکند و SP را ۲ واحد می‌افزاید. برای حالت دور، RET کلمه بالای پشته را به IP و CS منتقل می‌کند و SP را ٤ واحد می‌افزاید. هر عملوند عددی (مثلاً RET 4) با SP جمع می‌شود. RETN و RETF توسط MASM 50 معرفی شده‌اند که می‌توانید بازگشت نزدیک یا دور را دقیقاً کد نمایید.

کد منبع. RET / RETN / RETF [POPمقدار]

ROL/ROR : چرخش به چپ یا چرخش به راست

عمل. بیت‌های یک بایت ، کلمه، یا دو کلمه‌ای (در ۸۰۳۸٦ و مابعد) در یک ثبات یا حافظه را به چپ یا راست می‌چرخاند. عملوند ممکن است یک ثابت بلافصل یا ارجاعی به CL باشد. در 8088/86 مقدار ثابت فقط یک می‌تواند باشد و مقادیر چرخش بزرگتر در CL قرار می‌گیرد. در پردازشگرهای مابعد، ثابت حداکثر ۳۱ می‌تواند باشد. برای ROL سمت چپ‌ترین بیت وارد بیت 0 مقصد می‌شود. دیگر بیت‌ها به سمت چپ چرخش دارند. در ROR، بیت 0 وارد سمت چپ‌ترین بیت مقصد می‌شود، دیگر بیت‌ها به سمت راست چرخش دارند (RCL و RCR را نیز ببینید). بیت چرخش یافته نیز وارد CF می‌شود.

کد منبع. ROL / ROL register / memory , CL / immediate

SAHE : ذخیره‌سازی محتویات AH در پرچم‌ها

عمل. بیت‌های AH را در بیت‌های سمت راست ثبات پرچم ذخیره می‌کند. (LAHF را نیز ببینید).

کد منبع. SAHF (بدون عملوند)

SAL/SAR : شیفت جبری به چپ یا شیفت جبری به راست

عمل. بیت‌های یک بایت ، کلمه یا دو کلمه‌ای در یک ثبات یا حافظه را به راست یا چپ شیفت می‌دهد. عملوند باید یک مقدار بلافصل یا ارجاع به CL باشد. در 8088/86 ، ثابت باید فقط یک باشد و مقادیر بزرگتر شیفت باید در CL قرار گیرند. در پردازشگرهای مابعد مقدار ثبات حداکثر ۳۱ می‌تواند باشد. SAL به تعداد خاصی بیت‌ها را به سمت چپ شیفت می‌دهد و بیت‌های 0 در موقعیت‌های خالی سمت راست قرار می‌دهد. SAL دقیقاً مانند SHL عمل می‌کند. SAR یک شیفت محاسباتی است که علامت فیلد ارجاع شده را در نظر می‌گیرد. SAR بیت‌ها را به تعداد خاصی به سمت راست شیفت می‌دهد و بیت علامت (1 یا 0) در سمت چپ پر می‌کند. همه بیت‌های شیفت داده شده از بین می‌روند.

کد منبع. SAL / SAR register / memory , CL / immediate

SSB : تفریق با رقم فرضی

عمل. نوعاً در تفریق دودویی چند کلمه‌ای ، برای انتقال رقم سرریز شده به مرحله حسابی بعدی استفاده می‌شود SBB ابتدا محتویات CF(0/1) را از اولین عملوند تفریق می‌کند و سپس دومین عملوند را از اولین عملوند مانند SUB تفریق می‌کند (ADC را نیز ببینید).

کد منبع. SBB register / memory , Register / memory / immediate

SCAS / SCASB / SCASW / SCASD : پیمایش رشته

عمل. یک رشته موجود در حافظه را برای یافتن مقدار خاصی پیمایش می‌کند. برای SCASB مقدار را در AL قرار دهید، برای SCAW مقدار را در AX و برای SCASD (در ۸۰۳۸٦ و مابعد) مقدار را در EAX قرار دهید. جفت ES:DI رشته‌ای در حافظه را که باید پیمایش شود ارجاع می‌دهند. این عملیات عموماً با یک پیشوند REPE/REPNE استفاده می‌شود که تعداد شمارش در CX قرار می‌گیرد، از REPE برای یافتن اولین مورد مخالف و REPNE برای یافتن اولین مورد موافق

استفاده کنید. اگر DF صفر باشد، عملیات حافظه را از چپ به راست پیمایش می‌کند و DI را می‌افزاید. اگر DI یک باشد، عملیات حافظه را از راست به چپ پیمایش می‌کند و DI را می‌کاهد. REPn برای هر بار تکرار CX را کاهش می‌دهد. وقتی یک شرط مساوی (REPNE) یا نامساوی (REPE) رخ دهد یا CX صفر شود، عملیات خاتمه می‌یابد. آخرین مقایسه پرچم‌ها را صفر/ یک می‌کند. اگر شرط خاصی پیدا نشود، CX , REP را کاهش می‌دهد تا صفر شود، در غیر این‌صورت، DI و SI حاوی آدرس عناصر بعدی است.

کد منبع. (بدون عملوند) [REPnn] SCASB / SCASW / SCASD

SETnn : تنظیم مشروط بایت (در ۸۰۳۸۶ و مابعد)

عمل. یک بایت خاص را برحسب یک شرط تنظیم می‌کند. یک گروه با ۳۰ دستور ، شامل SET(N)S , SET(N)E , SET(N)C , و SET(N)L وجود دارد که مشابه تنظیم پرش‌های شرطی هستند. اگر یک شرط بررسی شده صحت داشته باشد، عملیات بایت عملوند را با یک و در غیر این‌صورت با صفر تنظیم می‌کند. یک مثال چنین است:

مقایسه محتویان BX,AX ; CMP AX , BX

اگر مساوی است، CL را با یک و در غیراین‌صورت با صفر تنظیم کن ; SETE CL

کد منبع. SETnn register / memory

SHR / SHL : شیفت منطقی به چپ یا شیفت منطقی به راست

عمل. بیت‌های یک بایت، کلمه، یا دو کلمه‌ای در یک ثبات یا حافظه را به چپ یا راست شیفت می‌دهد. عملوند ممکن است یک ثبات بلافصل یا ارجاعی به CL باشد. در 8088/86 ثابت ممکن است یک باشد و اگر مقدار بزرگتری باشد باید در CL قرار گیرد. در پردازشگرهای مابعد، ثابت حداکثر ۳۱ می‌تواند باشد. SHL و SHR شیفت‌های منطقی‌اند که با بیت علامت مانند یک بیت داده برخورد می‌کنند.

SHL بیت‌ها را به تعداد خاص به چپ شیفت می‌دهد و موقعیت‌های سمت راست خالی شده را با بیت‌های صفر پر می‌کند. SHL دقیقاً مانند SAL عمل می‌کند. SHR بیت‌ها را به تعداد خاص به راست شیفت می‌دهد و بیت‌های سمت چپ را با بیت صفر پر می‌کند. همه بیت‌های شیفت داده شده از بین می‌روند.

کد منبع. SHL / SHR register / memory , cl / immediate

SHRD / SHLD : شیفت با دقت مضاعف (در ۸۰۳۸۶ و مابعد)

عمل. چندین بیت را به یک عملوند شیفت می‌دهد. دستورات به سه عملوند نیاز دارند. اولین عملوند یک ثبات ١٦ یا ٣٢ بیتی است با موقعیت حافظه است که شامل مقداری است که باید شیفت داده شود. دومین عملوند یک ثبات (با همان اندازهٔ عملوند اول) است که حاوی بیت‌هایی است که در اولین عملوند شیفت داده می‌شود. سومین عملوند CL یا ثبات بلافصل است که حاوی مقدار شیفت است.

کد منبع. SHLD / SHRD register / memory , register , CL / immediate

STC : یک کردن پرچم نقلی

عمل. CF را یک می‌کند (CLC پرچم نقلی را صفر می‌کند)

کد منبع. STC (بدون عملوند)

STD : یک کردن پرچم جهت

عمل. DF را یک می‌کند تا سبب شود عملیات رشته‌ای مانند MOVS پردازش را از راست به چپ انجام دهد. (CLD را برای صفر کردن DF ببینید).

کد منبع. STD (بدون عملوند)

STOS / STOSB / STOSW / STOSD

عمل. محتویات انباشتگر را در حافظه ذخیره می‌کند. وقتی با یک پیشوند REP با تعداد شمارش در CX استفاده شود، عملیات مقادیر یک رشته را به دفعات مشخص شده کپی می‌کند، این عملیات برای فعالیت‌هایی نظیر پاک کردن یک ناحیه حافظه مناسب است. برای STOSD مقدار را در AL قرار دهید، برای STOSW مقدار را در AX و برای STOSD مقدار را در EAX قرار دهید. جفت ES:DI موقعیتی در حافظه را ارجاع می‌دهند که مقدار باید ذخیره شود. اگر DF صفر باشد، عملیات ذخیره سازی در حافظه از چپ به راست انجام می‌گیرد و DI را می‌افزاید. اگر DF یک باشد، عملیات از

راست به چپ، ذخیره‌سازی را انجام می‌دهد و DI را می‌کاهد. REP برای هر بار تکرار CX را می‌کاهد و وقتی صفر باشد عملیات خاتمه می‌یابد.

کد منبع. (بدون عملوند) [REP] STOSB / STOSW / STOSD

SUB : تفریق مقادیر دودویی

عمل. مقادیر دودویی در یک ثبات، حافظه یا بلافصل را از یک ثبات تفریق می‌کند، یا مقادیر در یک ثبات یا بلافصل را از حافظه تفریق می‌کند. مقادیر ممکن است بایت، کلمه یا دو کلمه‌ای باشند. (۸۰۳۸٦ و مابعد). (SBB را ببینید).

کد منبع. SUB register / memory , register / memory / immediate

TEST : بررسی بیت‌ها

عمل. عمل AND منطقی را انجام می‌دهد، اما عملوند مقصد را تغییر نمی‌دهد. هر دو عملوند باید بایت، کلمه، یا دو کلمه‌ای (۸۰۳۸٦ و مابعد) در یک ثبات یا حافظه باشد، دومین عملوند ممکن است بلافصل باشد. بعد از اجرای آن، ممکن است از JE یا JNE برای بررسی پرچم‌ها استفاده کنید.

کد منبع. TEST register / memory , register / memory / immediate

WAIT : قراردادن پردازشگر در حالت انتظار

عمل. به پردازشگر اصلی اجازه می‌دهد تا در حالت انتظار باقی بماند تا هنگامی که یک وقفه خارجی رخ دهد، بدین منظور که با یک کمک پردازشگر همزمان شود. پردازشگر اصلی منتظر می‌ماند تا کمک پردازشگر اجرا را خاتمه دهد و با دریافت یک علامت از پایه TEST پردازش را آغاز کند.

کد منبع. WAIT (بدون عملوند)

XADD : معاوضه و جمع (۸۰۴۸٦ و مابعد)

عمل. عملوندهای منبع و مقصد را باهم جمع می‌کند و حاصل را در مقصد ذخیره می‌سازد. همچنین مقدار اصلی مقصد را به منبع منتقل می‌کند.

کد منبع. XADD register / memory , register

XADD : معاوضه

عمل. داده‌ها را بین دو ثبات (به‌صورت XCHGAH , BL) یا بین ثبات و حافظه (به‌صورت CX , word) XCHG معاوضه می‌کند.

کد منبع. XADD register / memory , register

XCHG : معاوضه

عمل. داده‌ها را بین دو ثابت (به‌صورت XCHGAH , BL) یا بین یک ثبات و حافظه (به‌صورت CX, word) XCHG معاوضه می‌کند.

کد منبع. XCHG register / memory , register / memory

XLAT / XLATB : تبدیل

عمل. بایت‌ها را به یک قالب متفاوت مانند، ASCII به EBCDLC تبدیل می‌کند. یک جدول را تعریف کنید ، آدرس آن را در BX یا EBX برای اندازهٔ ۳۲ بیت قرار دهید و در AL مقداری را که باید تبدیل شود قرار دهید. عملیات از مقدار AL بعنوان یک افست جدول استفاده می‌کند، بایت را از جدول انتخاب می‌کند و آن را در AL ذخیره می‌سازد. (XLATB معادل XLAT است).

کد منبع. (عملوند AL اختیاری است) XLAT [AL]

XOR: OR انحصاری

عمل. یک OR انحصاری منطقی روی بیت‌های دو عملوند انجام می‌دهد. هر دو عملوند باید بایت، کلمه ، یا دو کلمه‌ای (۸۰۳۸۶ و مابعد) باشند که XOR بیت‌ها را با هم جور کند. برای هر جفت بیت جور شده، اگر هر دو یکسان باشند، بیت اولین عملوند صفر خواهد شد، اگر بیت‌های جور شده متفاوت باشند بیت اولین عملوند یک خواهد شد(AND و OR را نیز ببینید).

کد منبع. XOR memory , register / memory / immediate register