The Unofficial Newsletter of Delphi Users - Issue #8 - October 10th, 1995

I apologize for the tardiness of this latest issue. The work level has been picking up a bit and it is hard to devote all the time needed for UNDU. I know that quite a few of you wait in the Borland Forum for each issue to come out, so I imagine the frustration level has been a bit high the last week. Anyway, I will try to keep them coming out as frequently as possible. Generally each issue will come out a month after the date of the previous issue.

I also wanted to take a few moments to thank all of my regular contributors. I receive mail every day from people who are new to Delphi and who appreciate the little bits of information found here that helps them get up to speed a little quicker. If you would like to contribute something to this newsletter, please email me at the above address. I am looking for all sorts of things: Book reviews, product reviews, programming tips & tricks, custom components, you name it! Don't worry if you feel unqualified to make a contribution... there is always someone who will benefit from what you have to offer. Also, don't be afraid that your contribution might be too simplistic... many of the readers of UNDU are programmers new to Delphi. If you can think of something to write about that you haven't seen elsewhere, send it to me, and I will find space in an upcoming issue!

Delphi Informant Magazine

Return to Front Page

The Main (?) Event: Delphi World Tour Reviewed

Paul H. Comitz - CIS 102565,3167

My 1989 Dodge RAM Van is on two wheels as I round the corner onto International Drive. On my right is the Blazing Pianos Rock 'n' Roll Piano Bar. I make a mental note to check it out sometime. But not now, I'm late for the Orlando, Florida edition of the **Delphi World Tour**, presented by a company called Softbite International. It's day one of a scheduled two day affair. Day one is billed as "The Main Event". The Orlando Marriot looms into my crosshairs. Parking is surprisingly easy and (shockingly) free. I'm in my chair by 8:47 am, just 17 minutes late.

I'm in a standard hotel convention center ballroom. Free copies of the September 1995 issue of Delphi Informant and a 44 page presentation package are available at the door.   About 30 people are sitting at long, lunchroom style tables. Coffee and orange juice are available in the back. The instructor is an early thirties type named Cary Jensen. He is not a Borland employee. He has a clear speaking voice and is directing the seminar via his laptop and an overhead projection system. The overhead projection system is first rate and easy to read. The instructors laptop is the only computer in the room. This is a look and listen event.

The instructor is just completing his opening remarks. The agenda, which runs from 8:30 to 4:00 PM (one hour and twenty minutes for lunch), is as follows:

- *Overview of Delphi*
- *Tour of the Delphi Desktop*
- *Delphi by Example: Building a Client/Server Project*
- *Fine Tuning an Application*
- *Controlling the Environment*
- *Support Tools*

As the instructor begins the *"Overview of Delphi"*, I leaf through the presentation package. It becomes immediately obvious that "The Main Event" is intended for someone who has never seen Delphi before. The overview explains that Delphi is an optimizing compiler with a library of reusable "objects" in a visual development environment etc., etc., etc. While the material is   well presented and thoroughly explained, there's nothing here that most folks can't figure out on their own in a one or two hour session. This is my general feeling throughout the entire day.

"Tour of the Delphi Desktop" is exactly what it sounds like. We look at the various Delphi menus, the speedbar, and some accelerator keys. Ctrl-oo inserts the current compiler directives at the top of the unit. There's one I didn't know!!!

The section on *"Building a Client/Server Project"* should be called *"Building an Application   that uses Tables from a   Database"*. We build an application   that uses previously created tables. The tables identify Customers, Purchase Orders, Sales etc. The trick here is to link Delphi Table and Data Source objects to existing tables in a database. We use the DBNavigator object to cruise through the table. I suppose that if we were on a network and the database was located on a server, one could call this a client/server application. This doesn't really come across since we're looking at an instructor with a laptop and an overhead projector. What does come across is how easy it is to develop an application like this. The whole exercise takes about 45 minutes. This is by far and away the most interesting and useful part of the days activities. Once again, the material is skillfully presented and thoroughly explained.

The remainder of the day is anticlimactic. We look at things like the menu editor, the debugger, and the object browser . Again, there's nothing here that one can't pick up on their own.

The Bottom Line

One can attend the first day of the Delphi World Tour for $295 or both days for $550. These prices include a copy of Delphi. If you already own a copy of Delphi, you can attend without receiving a copy for $75 less. If you've never used Delphi before, its a pretty good deal and the information will be right up your alley. If you already know a bit of Delphi, day one is not worth the cost. I had to return to my boring day job so I missed day two. As I've told you three or four times by now, day one is titled "The Main Event". From MY perspective, "The Main Event" was probably day two. When I called Softbite International and asked for more detail on "The Main Event", they did not explain that the target audience was absolute Delphi beginners. I guess I should have been wary when the Softbite representative didn't know what I meant by 16 and 32 bit operating systems (hint Softbite). Unfortunately, the option to attend ONLY day two was not available (another hint Softbite).

Return to Front Page

The Beginners Corner - The State of the World and the Delphi VCL

Paul H. Comitz - Desperate Man Engineering - CIS 102565,3167

Last month I told you that I used to teach Turbo Pascal programming at a community college in China Lake, California. Recently, a former student saw one of my posts in the **comp.benchmarks** newsgroup. This student is now a Computer Science major at UC Santa Cruz. He asked me what I had been up to lately and I told him that Pascal was on the rebound because of a great new product called Borland Delphi. Here is his response:

"Does Delphi really solve the many problems of Pascal? If so, I wonder how good a chance it has at being made a standard. (That's the real problem: if it remains just a set of proprietary extensions to Pascal, like Turbo Pascal, available only in one compiler, then there will be a lot of difficulties getting people to use it.)"

There's a message here about computer science education at our universities. Despite the great things that have come out of UC and places like it, visual 4th generation programming simply hasn't arrived on campus. We're right in the middle of a developing trend where information is flowing, expanding, and mutating at a rate unlike anything the world has ever seen. The State of the World is a connection of millions of minds. You're part of it. The place to learn Delphi is right here, online, in the worlds biggest campus. Here at Online U, we engage our afterburners at takeoff and exchange information and ideas at mach 3. Extensions, you say??? I've never seen anything as extensible as Borland Delphi. Millions of minds contribute to these extensions. Their vehicle is the Delphi Visual Component Library.

The Visual Component Library (VCL) might be the hippest thing about Delphi. We've all seen printed circuit boards. Printed circuit boards are filled with Integrated Circuits (ICs) connected by copper traces. A 555 Timer IC takes DC voltage as an input and outputs a nice clean square wave at a frequency proportional to the voltage. This IC has only a few inputs and outputs. The DEC Alpha processor is a type of IC with 499 inputs and outputs. With the VCL, ICs aren't just for hardware types and OOP programmers anymore. The VCL allows us to take a software IC, insert it into our programs, and control the inputs, and therefore the outputs. Our Delphi ICs are called components. They reside in the Delphi Visual Component Library in a file called **COMPLIB.DCL**.

Plain Vanilla Delphi ships with about 90 components. If you'd like to see them, open up DELPHI.HLP and do a topic search on VCL components. You'll see all the standard components starting with TApplication and ending with TWordField. There's other ways to do this. Note the tabbed component palette. The "Standard" palette contains 14 components beginning with TMainMenu and ending with TPanel. The "Additional" palette contains 14 more components. If you continue to the "Samples" tab, you'll find a total of 77 components. You can select and drop any of these components onto a Delphi form. The same exercise can be accomplished by selecting **View|Component** List. The grouping in the component list is alphabetical by common name.

The State of the World is not perfect. You'll notice that some of the "Visual" components described in DELPHI.HLP don't appear in either the palette or the component list. I've heard some folks describe these items as "non-visual" components. One immediately wonders why a "Visual" component library contains "non-visual" components. It's a fair question. For those of you who are interested, stay tuned to this channel. Non-visual components are the subject of next month's "Beginners Corner". In the meantime it's important to understand that even though Delphi is a slick "VISUAL" compiler, many ESSENTIAL parts of Delphi are non-visual. We use some of these non-visual objects EVERY time we use Delphi. To convince yourself, select **View|Project Source**. You'll always see something like "Application.Run". Now read the DELPHI.HLP description of the object TApplication. We automatically use these non-visual objects all the time.

Let's continue to look at Visual components and the component library. As mentioned earlier. Delphi ships with 77 components. The default component library is called COMPLIB.DCL. If you look in your DELPHI\BIN subdirectory you'll find this file. Out of the box, the size of this file is 1,058,784, and the date stamp is

2/15/95. Select **Options|Install** Components. Note the library filename. The "Installed Units" listbox shows the contents of the VCL. Single click on the StdReg installed unit. 28 components (starting with TBevel and ending with TTabSet) are displayed in the classes listbox. These 28 components correspond to the "Standard" and the "Additional" notebook tabs on the component palette. Single click the DBReg installed unit. 19 components are displayed. The DBReg components correspond to the Data Access and Data Controls notebook tabs on the component palette. If you continue this exercise you'll find all 77 of the components that Delphi ships with.

Earlier I said that I'd never seen anything as extensible as Delphi. This extensibility is a result of the ability to add to the component library we just examined. Check the CompuServe Borland Delphi Forum (GO DELPHI) or the Informant Communications Group Forum (GO ICGFORUM). You'll find free downloadable components all over the place. You can add these components to your VCL and use them in your applications. Two of the interesting components that recently became available are the SlideBar (written by UNDU's editor, Robert Vivrette) and the StopWatch components. Both are available in the ICGFORUM and it's easy to add these components to your library.

Download the SlideBar and the StopWatch zip files. I like to put the zip files in their own subdirectory (i.e. \cmpnt\slidebar etc). When you unzip the SlideBar file. you'll find the dcu., .dcr and .pas files. When you unzip the StopWatch file be sure to use the -d option of pkunzip because the StopWatch zip file creates subdirectories when it expands. It's frighteningly easy add these components to your VCL:

1. Select **Options|Install** Components
2. Select **Add** (The Add Module Dialog is displayed)
3. Select **Browse**
4. Select slidebar.pas from wherever you've unzipped it (stpwatch.pas for the stopwatch component. stpwatch.pas is unzipped into its own subdirectory called \stpwatch).

At this point you'll notice several differences in the Installed Components dialog. First you'll notice that the Search Path has been updated to include the path to the directory where slidebar.pas (stpwatch.pas) resides. You'll also notice Slidebar (StpWatch) in the installed units listbox. However, if you single click on your new component "Not available" is displayed in the "Component Classes" listbox. This is because we have not yet recompiled the component library. The component library will be automatically recompiled as soon as you select OK. When you select OK, the hourglass will be displayed while the component library is recompiled. On my 120 mhz Pentium it took about 18 seconds to recompile the library when I added the StopWatch component. After you've recompiled, take a look at \delphi\bin\complib.dcl. Note the change in size and in the date and time stamp.

Now for the cool part. The SlideBar component is visible on the Additional component tab (use the image editor or the Borland Resource Workshop to examine slidebar.dcr if your wondering where the new icon on the component palette came from). Two StopWatch components are available on the System Tab. You can use these new components in your applications the same way you use the 77 original components that Delphi ships with. All three (remember you get two varieties of the StopWatch component) of the new components we've added here unzip with example projects so we can figure out to use them in our applications.

That's it for this month. Next month we'll continue our discussion of Delphi components by exploring the "non Visual" components that Delphi ships with.

Return to Front Page

Take the Delphi Informant Magazine Test Drive!

Mail or fax this form and receive a free issue of *Delphi Informant*TM Magazine, The Complete Monthly Guide to DelphiTM Development. If you choose to subscribe, you'll get a full year's worth (12 additional issues, 13 in all) for the super low price of $49.95*. If you don't, simply write "Cancel" on the invoice and owe nothing. *Delphi Informant*, no one does DelphiTM better.

Each big issue is packed with Delphi*TM *tips, techniques, news, and more!

Delphi-Related News
Product Reviews
Delphi Programming Tips and Techniques
Extending Delphi with DLLs and VBXs
Building Delphi Components
Client/Server Development
Book Reviews

And much, much more...

YES! I want to sharpen my DelphiTM programming skills. Sign me up to receive one free issue of Delphi InformantTM, The Complete Monthly Guide to Delphi Development. If I choose to subscribe, I'll get a full year's worth (12 additional issue, 13 in all) for the super low price of $49.95. If I don't, I'll simply write "Cancel" on the invoice and owe nothing

```
   Name: ________________________________________________________________

Company: ________________________________________________________________

Address: ________________________________________________________________

         ________________________________________________________________

   City: ___________________________   State: _____  Zip Code: __________

Country: _________________________   Telephone: _________________________

    Fax: ____________________________   E-Mail: _________________________
```

This offer good only to first-time, qualified subscribers.

*US and International Subscription Rates

United States
Magazine-Only One Year: $ 49.95, Two Year: $ 89.95
Magazine/Disk One Year: $119.95, Two Year: $219.95

Canada
Magazine-Only One Year: $ 54.95, Two Year: $ 99.95
Magazine/Disk One Year: $124.95, Two Year: $229.95

Mexico
Magazine-Only One Year: $ 74.95, Two Year: $139.95
Magazine/Disk One Year: $154.95, Two Year: $289.95

All Other Countries
Magazine-Only One Year: $ 79.95, Two Year: $149.95
Magazine/Disk One Year: $179.95, Two Year: $339.95

Subscription rates are subject to change without notice.

Mail or Fax the above order form to:

Informant Communications Group, Inc.
ATTN: *Delphi Test Drive Offer*
10519 E. Stockton Blvd. Suite 142
Elk Grove, CA 95624-9704
Phone: (916) 686-6610 Fax : (916) 686-8497

Return to Delphi Informant
Return to Front Page

Did you know you can record and playback keystrokes in the Delphi IDE? To record a series of keystrokes, simply hit **ALT-SHIFT-R** to begin recording. Then enter the keystrokes you wish to record. It appears that you can do just about anything here including cursor movement keys and function keys. To stop recording, simply hit **ALT-SHIFT-R** again. When you want to play back the series of keys, simply hit **ALT-SHIFT-P**. Pretty Cool!

Tips & Tricks

Return to Front Page

INTRODUCING

DELPHITM INFORMANT WORKSTM: 1995

The Must-Have Reference Source for the Delphi Developer

Delphi Informant Works is a collection of all of the articles and code from every issue of *Delphi Informant* Magazine in 1995. Available for the first time in electronic form, *Delphi Informant Works* is an invaluable reference to the serious Delphi developer.

Delphi Informant Works Includes:
* All of the Technical Articles, product reviews, and book reviews from *Delphi Informant* in 1995
* Many issues that are sold out and no longer available in print
* Text and Keyword Search
* Index to Delphi Informant Articles
* All Supporting Code, Sample Files, and Utilities
* 16-Page Color Booklet
* Improved Speed and Performance
* Electronic Version of *Delphi Power Tools* Catalog
* CompuServe Starter Kit with $15 usage credit

Available December 1995! Order Today!

YES! Please send me my copy of Delphi Informant Works: 1995. This $90* value is mine for the low price of $39.95 and I have added $5 for shipping and handling.

Number of Copies @ $39.95 ___ = $__________

California Sales Tax = $__________

Shipping & Handling Per Copy* = $__________

TOTAL ---> $__________

Name : ____

Company : ____

Address : ____

____

City : ____________________ State: ________ Zip Code: ________

Country : ____________________ Telephone: ____________________

E-Mail : ____

___Check/Money Order Enclosed ___Please charge my Visa/Mastercard/AmExpress

|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___| |___|___|-|___|___|

Credit Card Number Expiration Date

______________________________ ______________________________

Name as it appears on Credit Card Signature (please print)

***** **IMPORTANT** *****
* Payment must be made in US dollars on US banks
* Make checks payable to Informant Communications Group
* California Residents add 7 1/4% Sales Tax
* US RESIDENTS, please add $5 for shipping and handling per copy.
* INTERNATIONAL ORDERS, please add $US15 for shipping and handling per copy.

Mail, Fax, or E-mail this order form to:
Informant Communications Group, Inc.
ATTN: Delphi Informant Works CD Offer
10519 E. Stockton Blvd.
Suite 142
Elk Grove, CA 95624-9704

Phone : (916) 686-6610
Fax : (916) 686-8497
E-Mail: 74651,1557 (CompuServe)

* Value based on pro-rated one-year subscription price to *Delphi Informant* Magazine and Companion Disk, *Delphi Power Tools* catalog, and CompuServe Membership credit.

Return to Delphi Informant
Return to Front Page

The Life Of A DLL: A Relationship Between Motion and Forces

by Maga WiseCarver - CIS 102507,3374

OK, let's try to make this simple. Sound familiar? Actually with Borland Delphi it can be quite simple to create, and access, your own Dynamic Link Libraries. I will attempt to shed some light on many aspects of the DLL in this article, but along with this I suggest that you take a look at the PASSWORD DLL example that was shipped with Delphi, (\Delphi\Demos\Calldll), along with a few good examples that can be found on CompuServe in the Delphi forum.

In this example we are going to create a simple DLL that will serve as our main programs AboutBox. This example serves as a profitable one for you since it will demonstrate not only how to "fire-up" a DLL, but also one way to reduce your programs executable size. Our example here is an AboutBox, but in the real world you could very easily use this same technique to design any number of endless options for your program such as an Order form, Information box, Product registry, etc.

Before we begin designing our DLL, and the program that will use it, there are a few things about DLLs that you really need to know. Some of this subject matter is going to be a bit dry, however it will be helpful to look it over as there are bound to be at the very least a few things which will help you out later.

Note: Although I have compiled the following information, much of it is not my own writing but comes directly from Borland. For your benefit I feel this is best. Please understand that the use of this information is in cooperation with Borland's No-Nonsense License Statement and should be treated as such.

Dynamic-Link Libraries (DLLs)

A dynamic-link library (DLL) is an executable module (extension .DLL) that contains code or resources that are used by other DLLs or applications. In the Windows environment, DLLs permit multiple applications to share code and resources. The Delphi concept most comparable to a DLL is a unit. However, routines in units are linked into your executable file at link time (statically linked), whereas DLL routines reside in a separate file and are made available at run time (dynamically linked). DLLs provide the ability for multiple applications to share a single copy of a routine they have in common. The .DLL file must be in the same directory as the application at run time. When the program is loaded into memory, the application dynamically links the procedure and function calls in the program to their entry points in the DLLs used by the program. **Note:** DLLs can export procedures and functions only.

Delphi applications can use DLLs that were not written in Object Pascal. Also, programs written in other languages can use DLLs written in Object Pascal.

Global Memory and Files in a DLL

As a rule, a DLL does not own any files that it opens or any global memory blocks that it allocates from the system. Such objects are owned by the application that (directly or indirectly) called the DLL.

When an application terminates, any open files owned by it are automatically closed, and any global memory blocks owned by it are automatically deallocated. This means that file and global memory-block handles stored in global variables in a DLL can become invalid at any time without the DLL being notified. For that reason, DLLs should refrain from making assumptions about the validity of file and global memory-block handles stored in global variables across calls to the DLL. Instead, such handles should be made parameters of the procedures and functions of the DLL, and the calling application should be responsible for maintaining them.

Global memory blocks allocated with the **gmem_DDEShare** attribute (defined in the WinTypes unit) are owned by the DLL, not by the calling applications. Such memory blocks remain allocated until they are explicitly deallocated by the DLL, or until the DLL is unloaded.

Reusing Forms as DLLs

When you create a form that you want to use in multiple applications, especially when the applications are not Delphi applications, you can build the form into a dynamic-link library (DLL). A DLL is a compiled executable file, so applications written with tools other than Delphi can call them. For example, you can call a DLL from applications created with C++, Paradox, or dBASE.

DLLs are standalone files that contain the overhead of the component library (about 100K). You can minimize this overhead by compiling several forms into a single DLL. For example, suppose you have a suite of applications that all use the same dialog boxes for checking passwords, displaying shared data, or updating status information. You can compile all of these dialog boxes into a single DLL, allowing them to share the component-library overhead.

Building a form into a DLL takes three steps:

1. Declaring Interface Routines

When you are compiling an application into a DLL, interface routines enable you to access the routine in the DLL from an outside application. Adding an interface-routine declaration involves declaring a procedure or function in the interface section of the unit to be compiled into the DLL, and following that declaration with the export directive. When writing interface routines that will be called from languages other than Object Pascal, you must declare parameters and return values using types that are available in the calling language. For example, you should pass strings as null-terminated arrays of characters (the Object Pascal type PChar) rather than Object Pascal's native string type. After declaring an interface routine, you can define the routine in the implementation section of the unit.

Example

The following example declares the function GetPassword as a interface routine. The export directive following the declaration ensures that the DLL can export the routine to outside programs.

DLL Example 1

2. Modifying the Project File

When you are compiling a project into a DLL, you need to make the following edits in the project file:

1. Change the reserved word **program** in the first line of the file to **library.**
2. Remove the **Forms** unit from the project's uses clause.
3. Remove all lines of code between the **begin** and **end** at the bottom of the file.
4. Below the uses clause, and before the **begin..end** block, add the reserved word **exports,** followed by the names of the interface routines and a semicolon. Delphi will not create the list of interface routines to be exported.

After these modifications, when you compile the project, it produces a DLL instead of an application. Applications can now call the DLL to open the wrapped dialog box. The following example shows a typical project file before and after modification:

DLL Example 2

3. Accessing Routines Stored in DLLs

There are two ways to access and call a routine stored in a DLL: Using an external declaration in your program (static import or implicit loading) and using GetProcAddress and LoadLibrary to initialize procedure pointers in your program (dynamic import or explicit loading).

Using an external declaration to perform a static DLL import causes the DLL to be loaded before execution of your program begins. In this case you cannot change the name of the DLL at run time. Your program cannot be executed if it specifies a DLL that isn't available at run time.

Using GetProcAddress and LoadLibrary (the two must be used in conjunction) to import a DLL gives your program control over what DLL file is actually loaded. For example, Windows device drivers are all DLLs with the same interface, but that internally perform hardware-specific functions. Programs can use the device driver DLLs without knowing anything about the hardware. With a dynamic import, even if LoadLibrary fails to locate a DLL your program can continue to run.

Although a DLL can have variables, it is not possible to import them into other modules. Any access to a

DLL's variables must take place through a procedural interface.

When you compile a program that uses a DLL, the compiler does not look for the DLL, so it need not be present. If you write your own DLLs, you must compile them separately.

In imported procedures and functions, the external directive takes the place of the declaration and statement parts that would otherwise be present. Imported procedures and functions must use the far call model (use a far procedure directive or a {$F+} compiler directive)

Object Pascal provides three ways to import procedures and functions: By Name, New Name or Ordinal Number.

When you import a routine from a DLL with no index or name clause specified, the procedure or function is imported explicitly by name. The name used is the procedure's or function's identifier. When a name clause is specified, the procedure or function is imported by a different name than its identifier. **Note:** The DLL name specified after the external keyword and the new name specified in a name clause do not have to be string literals. Any constant string expression is allowed.

When you import a routine from a DLL with a name clause specified, the procedure or function is imported by a different name than its identifier.

When you import a routine from a DLL with an index clause present, the procedure or function is imported by ordinal. Importing by ordinal reduces the load time of the module because Windows does not have to look up the name in the DLL's name table.

Example

```
{The following example imports the GetPassword routine using the name PASSWORD.}
function GetPassword(const Password: string): Boolean; external 'PASSWORD';
```

Example

```
{The following example imports the GetPassword routine using the name NEWPASSWORD.}
function GetPassword(const Password: string): Boolean; external 'PASSWORD' name
'NEWPASSWORD';
```

Example

```
{The following example imports the GetPassword routine as the fifth entry point in
the DLL called PASSWORD.}
function GetPassword(const Password: string): Boolean; external 'PASSWORD' index 5;
```

External Declarations

External declarations provide a means so you can interface with separately compiled procedures and functions written in assembly language Import procedures and functions from DLLs

External directives consisting only of the reserved word **external** are used in conjunction with *{$L filename}* directives to link with external procedures and functions implemented in .OBJ files. External directives that specify a DLL name (and optionally an import name or import ordinal number) are used to import procedures and functions from DLLs.

The **external** directive takes the place of the declaration and statement parts in an imported procedure or function. Except for using the far call model, imported procedures and functions behave like regular procedures and functions.

Examples

```
function GetMode: Word; external;
procedure SetMode(Mode: Word); external; {$L CURSOR.OBJ}
function GlobalAlloc(Flags: Word; Bytes: Longint): THandle; far; external 'KERNEL'
index 15;
```

Run-Time Errors In DLLs

If a run-time error occurs in a DLL, the application that called the DLL terminates. The DLL itself is not necessarily removed from memory at that time because other applications might still be using it. To ensure that DLLs are removed from memory if your application encounters a run-time error, use exception handling.

Because a DLL has no way of determining whether it was called from a Delphi application or an application written in another programming language, it is not possible for the DLL to invoke the application's exit procedures before the application terminates. The application is simply aborted and removed from memory. For this reason, make sure there are sufficient checks in any DLL code so such errors do not occur. If a run-time error does occur in a DLL, the safest thing to do is to exit Windows entirely. If you simply try to modify and rebuild the faulty DLL code, when you run your program again, Windows will not load the new version of the DLL if the faulty one is still in memory. Exiting Windows and then restarting Windows and Delphi ensures that your corrected version of the DLL is loaded.

Now then, don't you feel much better knowing all of that? There's a lot of information you can find about creating and using DLLs with Delphi. We covered much of it here, but as you can tell by now, creating and using DLLs is also subjective to your own imagination.

Here are some very basics that you should try to remember about DLLs:

a. A DLL is an executable module that contains code or resources that are used by other DLLs or applications. b) DLL routines reside in a separate file and are made available at run time (dynamically linked).
c. DLLs can export procedures and functions only.
d. Memory blocks remain allocated until they are deallocated by the DLL, or until the DLL is unloaded.
e. DLLs are standalone files that contain the overhead of the component library (about 100K).
f. DLLs give, but they don't receive.

Now let's create our Aboutbox DLL using a new project with only one form. Design the form any way you want, and then refer to the Borland example "Reusing Forms as DLLs - Modifying the Project File" to make all of the necessary changes to your project file. Once you have this technique mastered you can put as many forms in each of your DLLs as you like, and you will never look at DLLs as an adversary again.

Project Source code

DLL Source Code

Return to Front Page

Index of Past Issues

Below is a complete index of all principle articles in past issues of the Unofficial Newsletter of Delphi Users. Provided that you have the prior issues in the same directory as this issue, you can click on any of these hotspots to go directly to that article. To return to the index, you can click on the **Back** button, or you can use the **History** list. Once you jump to one of these issues, you can navigate through the issue as you would normally, but you will need to go to the **History** list to get back to this index. There will be an updated index included in all future issues of UNDU.

Return to Front Page

Core Concepts with Delphi - The Compiler and You

by Alan G. Labouseur - CIS: 70312,2726

So you want to program in Delphi, eh?   Okay.   Fire it up, draw some controls, write some code, compile the program.   Now all that's left to do is test, test, and test some more.   But it hasn't always been this easy.   I'm not saying that programming is easy now, but the mechanics of writing code and translating it into a form that makes sense to a computer is easier.   It hasn't always been this way.

Remember non-integrated development environments, command line compilers with long argument strings, overnight batch processing, punch cards, flipping banks of switches, wiring vacuum tubes?   As a student of programming for many years (and many more years to come) I've heard some horror stories about how it was "back in the old days".   Some folks would have you believe that you had to chisel your instruction codes in stone and carry them on your back into the underground dungeon of mainframe batch processing where the unlucky or unwary would be bound with reel-to-reel tape and battered with tiles from the raised floor.   Perhaps that's a little exaggerated (although I'm not sure), but suffice it to say that we've got it pretty good now-a-days.   And it's getting better all the time.   In order to appreciate and understand a contemporary tool like Delphi, we first need to examine the programming languages of yore. Here's a quick look at source code, interpreters and compilers . . .

In The Beginning...

In the beginning, there was the IBM 650, and it was good. Programmers communicated with spinning drums of data by way of cryptic CPU instruction codes and memory addresses on punched cards. (Instruction codes are the precise numbers that the computer uses to represent its basic operations.) There were no tools to simplify matters or assist in this tedious and error-prone process.   It wasn't long until necessity mothered the invention of pseudo-code interpreters.   Pseudo-code provided a way to represent CPU operations in a more user-friendly manner.   The trade-off here (and there's always a trade-off) was that these pseudo codes which made sense to human programmers weren't understandable to the CPU.   They had to be translated, or interpreted.   This translation occurred statement by statement as the program was executing.   Thus, the birth of interpreted processing.

During the 1960's, first generation programming languages such as FORTRAN expanded the earlier pseudo-codes into English-like terms such as   DO, ASSIGN, and GOTO.   We can now call this a "programming language" rather than just user-friendly instruction codes.   Also, these programming languages had their own programs, called compilers, that translated their English-like terms into machine-readable instruction codes that the CPU could handle.   Rather than translating line by line at run time (as was the case earlier), we could now translate the program, or compile it, in advance.

And Now...

Let's jump forward to the present.   (We're skipping three or four generations of programming languages, but I won't tell if you won't.)   Now we're working with Delphi.   We're writing code in Object Pascal, and using the built-in compiler.   Let's look at these two activities in more detail.

We can begin writing Object Pascal source code by opening a new unit (**File | New Unit**).   Delphi uses these "units" to store the programs we write, our source code.   Each unit is actually a text file on our disk drive with the ".pas" extension.   We could put program source code in any old text file, but it needs to have the ".pas" extension in order for Delphi to recognize it as source code.

When we have written a nice program, we need it translated into the ".exe" file that can be run.   This is where the compile option comes in.   Selecting **Compile** from the **Compile** menu will cause Delphi to analyze our code, report any syntax errors, and generate an ".exe" if it was error free.   What's a syntax error you ask?   (Good question.   I'm glad you're paying attention.)   A syntax error occurs when you have used the programming language in a way that the compiler didn't expect you to or allow you to.

For example, the **writeln** command can be used alone or with an argument specified in parentheses. So **writeln;** and **writeln('Delphi');** are both proper uses. **Writeln[;** is not. This would be flagged by the compiler as an error in the programming language syntax. You have to correct all of these errors before the compiler can complete the translation task and give you your ".exe" to show off to your family and friends.

You've probably noticed that there is a Syntax Check selection in the Compile menu. Use this when all you want is your syntax checked and you don't want your source code translated into an executable file, even if it is syntax error free. This saves you a little time when a syntax check is all you need.

There are several events that cause compile-time errors other than mistakes in syntax. Data type mismatches will make the compiler complain, errors and omissions in variable scope can also cause problems, and there's more. Luckily, I don't have to address these in this article. Consider it something to look forward to.

Let me sum up our peek at compiled versus interpreted languages in the following table:

	Compiled Languages		**Interpreted Languages**
1.	Source code in a text file	a.	Source code in a text file
2.	Translated in advance by a compiler	b.	Not translated in advance
3.	Executed directly	c.	Requires an interpreter

Delphi is a compiled language. Our ".pas" source code files (1) are translated by the compiler (2) into ".exe" files that are run directly (3). An interpreted language, such as QBASIC, requires an interpreter (c) that translates your source code (a) as it's running (b). There are some interpreted languages widely in use today. Visual Basic is one such beast. When you select the "Make EXE File" option from the VB file menu, it takes your source code (a) and changes it slightly as it saves it in an ".exe" file, but it's not compiled. It's actually a partially translated version of the source code, which is later interpreted at run time (b) by the VB interpreter: VBRUN300.DLL (c).

I hope this has been helpful in providing some background information upon which we can build in the future. Feel free, encouraged even, to e-mail me with any comments, questions, or suggestions. Thank you and good night.

About the Author

Alan G. Labouseur is Vice President and co-owner of AlphaPoint Systems, Inc., a custom programming consultancy located in Brewster, NY. Alan has a Masters degree in Computer Science and has been developing custom software for ten years. He is currently working on Clipper and Delphi applications for clients in the NY-NJ-CT area. You can reach Alan through e-mail at AGL007@IX.NETCOM.COM or 70312,2726 here on CompuServe.

Return to Front Page

How to Use Code Examples

by Robert Vivrette - CIS: 76416,1373

A few readers have commented that it is not entirely clear how to utilize the code samples provided in this newsletter. It actually quite simple! Here's how its done:

Unit Files

If a Delphi Unit file (*.PAS) has been included with an article, simply go to the page containing the source code and pick **Copy** from the **Edit** menu. This will present a small dialog box that shows the textual portion of the page being viewed. Simply highlight the text in the box and then press **Copy**. This copies the selected text to the clipboard.

Now start up Delphi and start a new project. Go to the source code page for the unit that was just created (generally **Unit1**). Delete all of the text in the unit (by choosing **Select All** from the **Edit** menu and then hitting the **DEL** key). Next, pick Paste from the Edit menu. The code that was copied out of the newsletter issue will now be inserted.

Lastly, go to the **File** menu and use the **Save File As...** command to save the file while giving it an appropriate name.

If you are using Windows 95, the procedure for copying the text is a little different. They have removed the extra step of going to the dialog box and selecting the text to copy there. Instead, Win95 allows you to use the mouse to select the text right on the help page. Then you can pick **Copy** from the **Edit** menu to copy the text directly to the clipboard.

Form Files

If a Delphi Form file (*.DFM) file has been included, it will be in a textual form. Copy the text out of the issue using the same procedure mentioned above. When you get over to Delphi, pick **New Unit** from the **File** menu. Again, delete all of the default text that appears in the resulting code window. Now paste the textual description of the form into the file. Lastly, go to the **File** menu and pick the **Save File As...** command. Change the **Save File as Type** combo box to read **Form File (*.DFM)**. Then type in the name that you will give the form (include the DFM extension) and click on OK to save the file.

Now close the project. It will ask if you want to save the project... which you should do... and name it something appropriate. Then open the project again and you should be all set!

You might want to create a directory to hold the files in advance. That way, the files will not get jumbled together with other projects and source files.

Return to Tips & Tricks
Return to Front Page

Delphi QSort Source Code

```
type
  { callback routine types - make sure you declare them as FAR }
  { expected result:  <0 if elem[e1] < elem[e2];
                       0 if elem[e1] = elem[e2];
                      >0 if elem[e1] > elem[e2] }
  TCompNdxFunc = function(e1, e2: word): integer;
  TSwapNdxFunc = procedure(e1, e2: word);


{ This is the main sorting routine. It is passed the number of elements and the
  two callback routines. The first routine is the function that will perform
  the comparison between two elements. The second routine is the procedure that
  will swap two elements if necessary }
procedure QSort(uNElem: word; FCmp: TCompNdxFunc; FSwap: TSwapNdxFunc);
{ uNElem - number of elements to sort }

  procedure qSortHelp(pivotP: word; nElem: word);
  label
    TailRecursion,
    qBreak;
  var
    leftP, rightP, pivotEnd, pivotTemp, leftTemp: word;
    lNum: word;
    retval: integer;
  begin
    TailRecursion:
      if (nElem <= 2) then
        begin
          if (nElem = 2) then
            begin
              rightP := pivotP +1;
              if (Fcmp(pivotP, rightP) > 0) then Fswap(pivotP, rightP);
            end;
          exit;
        end;
      rightP := (nElem -1) + pivotP;
      leftP :=  (nElem shr 1) + pivotP;
      { sort pivot, left, and right elements for "median of 3" }
      if (Fcmp(leftP, rightP) > 0) then Fswap(leftP, rightP);
      if (Fcmp(leftP, pivotP) > 0) then Fswap(leftP, pivotP)
      else if (Fcmp(pivotP, rightP) > 0) then Fswap(pivotP, rightP);
      if (nElem = 3) then
        begin
          Fswap(pivotP, leftP);
          exit;
        end;
      { now for the classic Horae algorithm }
      pivotEnd := pivotP + 1;
      leftP := pivotEnd;
      repeat
        retval := Fcmp(leftP, pivotP);
        while (retval <= 0) do
          begin
            if (retval = 0) then
              begin
                Fswap(leftP, pivotEnd);
                Inc(pivotEnd);
              end;
            if (leftP < rightP) then
```

```
              Inc(leftP)
            else
              goto qBreak;
            retval := Fcmp(leftP, pivotP);
          end; {while}
        while (leftP < rightP) do
          begin
            retval := Fcmp(pivotP, rightP);
            if (retval < 0) then
              Dec(rightP)
            else
              begin
                FSwap(leftP, rightP);
                if (retval <> 0) then
                  begin
                    Inc(leftP);
                    Dec(rightP);
                  end;
                break;
              end;
          end; {while}
      until (leftP >= rightP);
    qBreak:
      if (Fcmp(leftP, pivotP) <= 0) then Inc(leftP);
      leftTemp := leftP -1;
      pivotTemp := pivotP;
      while ((pivotTemp < pivotEnd) and (leftTemp >= pivotEnd)) do
        begin
          Fswap(pivotTemp, leftTemp);
          Inc(pivotTemp);
          Dec(leftTemp);
        end; {while}
      lNum := (leftP - pivotEnd);
      nElem := ((nElem + pivotP) -leftP);
      if (nElem < lNum) then
        begin
          qSortHelp(leftP, nElem);
          nElem := lNum;
        end
      else
        begin
          qSortHelp(pivotP, lNum);
          pivotP := leftP;
        end;
      goto TailRecursion;
    end; {qSortHelp }

begin
  if (uNElem < 2) then  exit; { nothing to sort }
  qSortHelp(1, uNElem);
end; { QSort }
```

[Return to QSort Article]
[Return to Front Page]

Delphi QSort Example

```
{This is the user-defined function that will be used to compare the elements}
function SortCompare(e1, e2: word): integer; far;
begin
  with MyForm.StringGrid1 do
    begin
      if (Cells[1, e1] < Cells[1, e2]) then
        Result := -1
      else if (Cells[1, e1] > Cells[1, e2]) then
        Result := 1
      else
        Result := 0;
    end; {with}
end;

{This is the user-defined function that will be used to swap 2 elements}
procedure SortSwap(e1, e2: word); far;
var
  s: string[63];  { must be large enough to contain the longest string in the
grid }
  i: integer;
begin
  with MyForm.StringGrid1 do
    for i := 0 to ColCount -1 do
    begin
      s := Cells[i, e1];
      Cells[i, e1] := Cells[i, e2];
      Cells[i, e2] := s;
    end; {for}
end;

procedure TMyForm.Button1Click(Sender: TObject);
begin
  QSort(StringGrid1.RowCount-1, SortCompare, SortSwap);
end;
```

Return to QSort Article
Return to Front Page

```
unit PassForm;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Forms, Controls, Forms, Dialogs, StdCtrls, Buttons;

type
  TPasswordForm = class(TForm)
  ...     { various declarations go here }
  end;

var
  PasswordForm: TPasswordForm;

function GetPassword(APassword: PChar): WordBool; export;
```

Sorting Objects With QSort

by Peter Szymiczek, 100351,2260

Being also a C++ programmer, I often feel that some language constructs or library functions are missing in Delphi. Recently for example, I needed a sort routine, and a fast one. Having no support in the Delphi library, I resorted to C++, thinking that I should be able to extract the qsort function and its helpers from the Borland C++ library, and somehow make it work in Delphi. I was even prepared to support these calls with assembly language, necessary to cover up differences in stack usage.

Then I examined the source code of the qsort, and decided that it's easier, and certainly more elegant to port it to Pascal. An hour later I had a working equivalent of the C function, but it turned out to be not good enough. Okay, it was fast, but all it could do was sort arrays. What I needed was a procedure that would sort just about anything, not necessarily occupying a continuous memory block, for example a string grid.

The original qsort operates on pointers to data records occupying continuous memory area. To determine the order of elements, it references a caller-supplied function, which knows how to compare two records, given pointers to them. Whenever a swap is required, it calls fast assembly language code that does the job.

To achieve my goal, I had to operate on a bit higher abstraction level. Instead of processing pointers, I used element indices, assuming that the sorted object is indexed [1..n]. Instead of providing the compare function with direct data addresses of elements, I pass indices and rely on the compare function to find the two data records. If the actual data is indexed [0..n-1] or in any other way, the compare routine must convert the index.

Another modification I had to make was to handle swapping of elements. As the sort function does not know where these elements are, it has to resort to a callback again. Caller-supplied swap function receives numbers of two records to swap and is expected to find them and swap their order, whatever it involves.

Note that this organization lets sort elements of uneven size, as long as supplied swap and compare routines can handle them!

Delphi QSort Source Code

Delphi QSort Example

Return to Tips & Tricks

Return to Front Page

Getting A List Of Running Programs

by Daniel Kinnaer, 102035,1527

Below is a piece of code in which all the names of running or minimized tasks in Windows are searched and thrown into a listbox. This piece of code can be used to search if a particular program is somewhere in memory or for some other purpose. I use this piece of code to find out if there are other programs that may be conflicting with my own.

```
procedure TForm1.Button1Click(Sender: TObject);
{Places the modulenames of the running/minimized tasks into a listbox   }
var pTask : pTaskEntry;
    Task  : bool;
    Pstr  : array [0..79] of Char;
    Str   : string[80];
    byt_j : byte;
begin
  ListBox1.Clear;                            {clear contents of the listbox}
  GetMem(pTask, SizeOf(TTaskEntry));          {Reserve memory for TaskEntry}
  pTask^.dwSize:=SizeOf(TTaskEntry);
  byt_j:=0;                           {Set up a counter for number of tasks}
  Task:=TaskFirst(pTask);                                   {Find first task}
  While Task do
    begin
      inc(byt_j);                         {count number of different tasks}
      Str:=StrPas(pTask^.szModule);       {Convert PStr into Pascal string}
      Listbox1.Items.Add(Str);           {Store Pascal string into listbox}
      Task:=TaskNext(pTask);                  {Check for next possible task}
    end;
  Label1.Caption:=IntToStr(byt_j)+ ' tasks found';          {Show counter}
end; {--------------------------- TForm1.Button1Click(Sender: TObject);}
```

On the form there is a Label1, ListBox1 and Button1. If a user clicks on the button, the above procedure is started and the listbox is filled up with a few strings. The number of strings in the listbox is represented using the label.

Important: Because of the references to the various Task-related procedures and types, you will need to add **ToolHelp** to the **uses** clause.

If you have played with the DebugBox component introduced here 2 issues ago , you could even add this procedure into that component and tie it to a public method. That way, you could call the method to have it record in the debug box a line showing the current names of (or number of) currently running programs.

Return to Tips & Tricks
Return to Front Page

Occasionally, you find that really neat use for a certain function. Or, maybe you find that the documentation on a particular part of Delphi is a bit confusing. Whenever I run across situations like this, I use the Windows Help system's **Annotate** feature. It allows you to put your own little comments into <u>any</u> windows help file (*.HLP).

To use it, simply choose **Annotate** from the **Edit** menu of the help file you are currently viewing. Then type in any comment you may wish to make and click **Save**. Back in the help file you will see a little paper-clip symbol. This is a reference to your annotation. Simply click on it and your note will be displayed!

Windows is not modifying the help file, but is rather storing these notes in a separate annotation file (*.ANN) in the \WINDOWS directory.

Delphi Articles Recently Printed

by Mark Meyer - CIS: 72260,2524

Some recent articles published relating to Delphi:

PC Tech June/July 95, vol 6. No. 2, pg 33, - *Using Delphi as a front end to Winsock.*
PC Tech Aug/Sept 95, vol. 6. No. 3, pg 81 - *Creating business object classes. pg 219 (same issue) - playing .wav files in delphi the easy way.*
Windows Developer May 95, pg 92 - *Building Modular Apps with Borland's Delphi.*
Microsoft Systems Journal June 95, pg 37 - *Borland Delphi: A New Entry in the Pantheon of Visual Programming Tools.*
Client Server Advisor May 95, pg 84 - *Brave New World Predicted for Borland Delphi.*
Datamation May 15, 95, pg 67 - *Borland's Delphi: The Object is Control.*

You may also find the following files helpful when trying to understand your options when it comes to development tools.

ftp.borland.com /pub/techinfo/techdocs/language/delphi

delphifs.zip - delphi fact sheet
delphiqi.zip - delphi quick introduction
revr_doc.zip - 33 page review of delphi, underlying philosophy, and how it relates to other products of the RAD genre.

ftp.borland.com /pub/techinfo/techdocs/language/delphi/gen

del&pb.zip - comparison of delphi & powerbuilder
del&vb.zip - comparison of delphi & visual basic
allfaq.zip - all frequently asked questions from all forums (cis)

Return to Front Page

```
program Password;

uses Forms,
     PassForm in 'PASSFORM.PAS' {PasswordForm};

{$R *.RES}
begin
  Application.CreateForm(TPasswordForm, PasswordForm);
  Application.Run;
end.
```

And after modifications...

```
library Password;                     { 1. reserved word changed }

uses                                  { 2. removed Forms, }
  PassForm in 'PASSFORM.PAS' {PasswordForm};
exports
  GetPassword;                        { 4. add exports clause }

{$R *.RES}
begin                                 { 3. remove code from main block }
end.
```

Project Source

```
program TheMain1;

uses
  Forms,
  TheMain in 'THEMAIN.PAS' {MainForm};

{$R *.RES}

begin
  Application.CreateForm(TMainForm, MainForm);
  Application.Run;
end.
```

Main Form Source

```
{*This is the main form that will call our DLL}
unit TheMain;

interface

uses  {*Nothing need be added to the uses clause for this example}
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls;

{*Add this line of code}
procedure LoadOurAboutbox(Handle: THandle); {*Export from our DLL}

type
  TMainForm = class(TForm)
    LoadAboutBtn: TButton;
    procedure LoadAboutBtnClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

{*Add this line of code}
procedure LoadOurAboutbox; external 'THEDLL' index 1;
{*Our procedure declaration and the name of the DLL}

procedure TMainForm.LoadAboutBtnClick(Sender: TObject);
begin
  LoadOurAboutbox(Application.Handle); {*Final step, load the Aboutbox}
end;

end.
```

Return to DLL Article
Return to Front Page

DLL Source

```
library Thedll;                           {*Modified line}

uses
  {Forms,}                                {*Forms unit removed}
  Aboutbox in 'ABOUTBOX.PAS' {Form1};     {*Our Aboutbox form}

exports                                   {*Add exports clause}
   LoadOurAboutbox index 1;               {*We will use this to show our Aboutbox}

{$R *.RES}

begin
{ Application.CreateForm(TForm1, Form1);
  Application.Run; }                      {*Remove all code between begin and end}
end.
```

About Box Source

```
unit Aboutbox;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages,
  Classes, Graphics, Controls, Forms, Dialogs;

type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

{*Add this line of code}
procedure LoadOurAboutbox(Handle: THandle); export; {*We created this-
                procedure and gave it the export directive}

implementation

{$R *.DFM}

{*Add the following lines of code}
procedure LoadOurAboutbox(Handle: THandle); {*Our procedure}
begin
  Application.Handle := Handle; {*Delphi default application calling}
  Form1:= TForm1.Create(Application); {*Create the Aboutbox}
  try                   {*Delphi reserved word}
    Form1.ShowModal;    {*Try and make the Aboutbox modal}
  finally               {*Finally, or "make sure"}
    Form1.Free;         {*Unload the instance of our Aboutbox}
  end;
end;

end.
```

Return to DLL Article
Return to Front Page