Pocket Programming Language NewsLetter
December 2006

Welcome to the PPL newsletter. We will be releasing a new one every month with lots of news about PPL, new releases, articles, coding tips and tricks to make your life easier. The PPL newsletter is open to contributions, if you have an article, tips, advertising or anything else related to PPL, we will be very happy to include your submissions.

Table of content

Editorial

It has been almost three full months since PPL was released and the response has been very good. The community is growing by the day and the future is bright my friend. As you read this, we are actively preparing to release version 1.1 which will be a major milestone for PPL. It will include major new features like the component viewer in the PIDE, better support for Bluetooth, TCP/IP and Infrared and two amazing libraries for database development. The Simple SQL Database will mark a new beginning for PPL as a powerful database development tool. We also have a Simple ADO library which allow to connect to Microsoft Access databases very easily. This new version offers even more new features, the list is so long we cannot list them here.

Yes, we know, we were suppose to have released 1.1 in November and 1.2 in December. We are sorry we couldn't stand by our promesses. The development is still small and we are all experiencing very hard times on the personal and work level. We would have liked to finish the year on a better note but some things are not under our control. We will take some time off during the holidays, spend some time with the family and prepare to be back in full force in January.

We wish you all a great Christmas time and a happy new year! See you next year, PPL has a bright future because of all your dedication and support. Thank you very much.

What's new

GameAPI Demo contest extended!

Our game demo contest has been extended to December 15th 2006. The prizes are as follow:

1st prize: $150 USD
2nd prize: $100 USD
3rd prize: $50 USD

Please send your entries to support@arianesoft.ca

Version 1.1 release date.

Finally, the greatly anticipated version 1.1 is coming out on December 8th 2006. This new version is packed with tons of new features and numerous bug fixes.

Price of PPL will be raised in January 2007.

As much as we love the fact that PPL was low-priced, we cannot keep the current introduction price for much more longer. As of January 2007, PPL Standard will be priced at $59.95 USD and PPL Profesional will be sold for $99.95 USD. If you are planning to buy PPL soon, now is a good time!

Coding Tips section:

Introducing Arrays.

When it comes time to store multiple values into a single variable, we have two choices:

1. Use an array variable
2. Use a linked-list variable

We will use the array variable for now since we will review the linked-list variable type in a later tutorial. PPL offers many ways to work with array variables, including arrays of different sizes and arrays of strings.

Arrays can be defined on local or global variables. To define an array, PPL comes with multiple functions to help you do this.

```
Dim(var$, 10);
```

This will create an array of 10 elements of the default type TDOUBLE (8 bytes) for variable var$. To declare multi-dimensional arrays, do the following:

```
Dim(var$, 10, 10, 10);
```

You can later access arrays just like other variables by specifying an offset within brakets [].

```
Dim(var$, 10, 10);
var$[0, 0] = 102.24;
var$[9, 9] = 23.2873;
```

Notice that we use 0,0. Arrays offsets start at 0 and goes to the array size – 1. If your array size is 10, 10, then the minimum offset if 0,0 and the maximum offset is 9,9.

To create arrays with custom element sizes, use SDIM().

```
SDIM(var$, TBYTE, 10, 10);
```

This will create an array of 10, 10 elements of type TBYTE (1 byte).

Now, what about strings? You will be happy to hear that PPL handles strings transparently with just a little twist. PPL stores only the string pointer address in the array's elements. Therefore the use of the @ operator is required to retrieve the string when accessing an array element.

```
Dim(var$, 10);
var$[2] = "Jack Bower";
var$[3] = "Joe Bloe";

ShowMessage(@var$[2] % "  " % @var$[3]);
```

Structures revisited!

Structures offers enormous flexibility when you design a program. They allow to store information in a nice clean way into your variables. I like to explain structures as a single record database. A structure is like a series of fields that can be stored in one record (the variable). Take the following example:

```
struct (s$, "a", "b");
```

The variable s$ has two elements, a and b. Each element can contain a separate value. The default type for structure's elements is a TINT (4 bytes) value. You can also specify which value type the element will be holding.You have multiple choices here:

TBYTE	**1 byte**
TSHORT	**2 bytes**
TINT	**4 bytes**
TUINT	**4 bytes (unsigned)**
TWIDE	**4 bytes (unicode character string)**
TDOUBLE	**8 bytes**
TLONG	**8 bytes (no decimal)**

```
struct (s$, "a", tbyte, "b", tdouble);
```

The following structure s$ would be 9 bytes in size. 1 byte for element a and 8 bytes for element b.

If you need to specify your own size in bytes you can also easily do it:

```
struct (s$, "a", tbyte, "b", 50);
```

Element b contains 50 bytes.

Now how do you access the structure variable elements you ask? Noting is easier:

```
s.a$ = 10;
s.b$ = 20;
```

What about strings in structures? You will see that PPL is very flexible but can also be a little more complex to use in some cases. You will need to be careful when using strings in structures. PPL either stores a pointer of the string that is assigned to the structure's element in the case where the element size is TINT, TUINT, TWIDE, TDOUBLE or TLONG. If the element size is a user-defined length, then the string is copied directly to the structure's element memory location.

```
struct (s$, "a", "b", 50);
s.a$ = "Hello World!";
s.b$ = "Hello Again!";
```

The main difference here is that the string "Hello World!" is not stored in s.a$ but rather stored somewhere in memory and only its pointer address is stored in s.a$. "Hello Again!" is stored directly into s.b$.

```
ShowMessage(s.a$);
```

If we try to access s.a$ like the previous code, only its pointer address value will be printed. To access s.a$ as a string we need to use the @ operator to convert a pointer to a string.

```
ShowMessage(@s.a$);
```

Now "Hello World!" will be printed in the dialog message.

To access s.b$ no need to do anything special.

```
ShowMessage(s.b$);
```

This will display "Hello Again!".

List of structures.

Now that we know about structures, how can we store multiple structures into one single variable? The best solution is either:

1. An array of structures
2. List of structures.

Let's take a look at list of structures and later array of structures.

Everytime you add a new item to a linked-list, the variable type is initialized, therefore you need to re-structure the item.

```
For (i$, 1, 10)
  Add(l$);
  struct(l$, "a", "b");
  l.a$ = i$;
  l.b$ = i$ + 10;
end;

ForEach(l$)
  ShowMessage(l.a$ % "," % l.b$);
end;
```

There is nothing more to add here other than the fact that you can define different types of structures for each list item. Powerful isn't it?

Arrays of structures.

There comes a time when you need to store a series of data in the same format. Storing values in a structure will help clarify your code and your programming task as well. However, you might want to store more of the same data in multiple structures. This is where arrays of structures become very handy.

Lets first define our structure variable format:

```
struct(s$, "a", "b");
```

Next we will make this structure an array but keeping the structure information at the same time. Notice we use TDIM() and not the regular DIM() function? The TDIM() function is a special function that can be used with structures only.

```
tdim(s$, 10);

s.a$[0] = 1;
s.a$[1] = 2;

ShowMessage(s.a$[0] % "," % s.a$[1]);
```

Arrays of structures are very ressemblant to lists of structures but offers an a great alternative is lists are too complicated for you.

Arrays in structures.

We have seen that structures can be defined as arrays using the TDIM() function. But what about having one element of the structure being an array of values?

PPL offers a transparent way to do just this using the DIM() function. It will create an array from

Lets first start by defining our variable structure:

```
struct(s$, "a", "b");
```

Now lets dimension our element, in our case we will use s.a$:

```
dim(s.a$, 10);

s.a$[0] = 1;
s.a$[1] = 2;
s.b$ = 3;

ShowMessage(s.a$[0] % "," % s.a$[1] % "," % s.b$);
```

You can even use strings with your array element, don't forget to use the @ operator to convert array elements to string since they only point to a pointer:

```
s.a$[2] = "Hello World!";
ShowMessage(@s.a$[2]);
```

As you see, structures are very flexible and quite efficient too. They can support multiple type of data, even arrays of double type values and even strings.

Packages Come In All Sizes by Eric Pankoke

There will be very few instances where you will need to distribute an application without some sort of supporting file set. Whether you're building a database system or a multimedia application, you might have one or more directories filled with extra material that you don't necessarily want the user to see outside of the application itself. PPL provides a nice mechanism for this through the use of a Package file. A package file is a file that can contain one or more files of any type all bundled up together into one file with a .pkg extension.

You can create a package file pro grammatically or you can use the Package Manager supplied with PPL to do so. Run the PPL console (it's called PPL.EXE, and resides under the RUNTIME directory of your PPL install). From the File menu, select Package Manger. You can use this tool to view, create and manage package files. It's pretty self explanatory, so I won't really go into details here.

Once you've created a package file, you'll of course want to use it in your program. To open a package file, you simply call:

```
handle$ = OpenPackage(AppPath$ + "mydata.pkg", key$);
```

The first parameter is a full path and name to the file you wish to open, and the second parameter is the key used to unlock the package file. If a file with the name you've specified does not exist, OpenPackage will create it and use the value specified in key$ as the password. When creating a package file in the Package Manager you do not get the opportunity to supply a key, so key$ would be an empty string (""). If you want a key, you will need to create the package pragmatically. You must be sure to keep track of the return value, as this is the handle that will be passed to all other package functions.

To retrieve the contents of a package file, call:

```
PackageFiles(&lst$, handle$);
```

This will provide you with a list containing the names of all of the files contained in the package. Of course, you will probably know all of these names already, so let's get to the heart of the matter: extracting and using the files. There are currently two ways of retrieving a file. The quickest way is to call the following:

```
data$ = LoadPackageFile(handle$, "filename");
```

This returns the contents of the file as a string in memory, which you can then display or manipulate however you choose. Currently, if the file is some sort of multimedia file or a database, this won't be of much use to you. Starting in PPL v1.1, however, there are two new functions that will work in conjunction with LoadPackageFile called LoadSpriteFromMem and LoadSoundFromMem. These will be discussed more after 1.1 is released. For cases where you need to interact with the file in some way, you'll want to call:

```
tmpfile$ = ExtractFileFromPackage(handle$, "filename");
```

This function will retrieve the contents of the requested file and store it in a temporary file. The return value is the name of the temporary file where the data is stored. So let's say you wanted to retrieve a database, do some work on it, then store the database back to the package. The code would look something like this:

```
dbfile$ = ExtractFileFromPackage(handle$, “MyData.db”);
dbhandle$ = sql_open(dbfile$);
if(dbhandle$ > 0)
  //Check out the October newsletter for an SQLite primer
end;
dbnew$ = ExtractFilePath(dbfile$) % "aviator.db";
MoveFile(dbfile$, dbnew$);
AddFileToPackage(package$, dbnew$);
DeleteFile(dbnew$);
```

If you will need to place the file back into the package after you've done your work with it, the 3 lines following end; are necessary. ExtractFileFromPackage creates a random name for the file, and AddFileToPackage takes the name of the file you're adding and uses that as the name inside of the package. So, if you want to replace the file that exists in the package with the version your application has just modified, you need to rename it to match the name that exists in the package. ExtractFilePath returns the path portion of a path / file name string, so if you use that on the file path returned from ExtractFileFromPackage, then use the file name that was used to add the file to the package initially, you can rename the temp file so that it will be stored correctly in the package again. The final step to updating the package is:

```
AddFileToPackage(handle$, “filename”);
```

The first parameter is that wonderful handle that was retrieved on your call to OpenPackage. filename is a string containing the full path and name of the file you wish to add. As mentioned before, the actual name of the file (no path) will be used to reference the file within the package. If you call AddFileToPackage with a file that already exists in the package, the file in the package will be replaced with the one you are adding.

Finally, when all of your work is done and you don't need the package any more, simply call:

```
ClosePackage(handle$);
```

This will make sure that the contents of the package have been updated and the handle will be released from memory.

Using Multiple Windows by Brad Manske

This month I want to address issues dealing with the creation of multiple windows in your PPL programs. This is not a tutorial on creating windows. It is more a collection of tips for when you do create multiple windows.

ShowMessage is the most simple way to create a second window. It is useful for debugging and simple informational pop-up windows.

```
ShowMessage("Hello World" + #13#10 + "and Country");
```

Take a look at the #13#10 added above. This is a Carriage Return/Line Feed added to produce a second line. By formatting the text, you can make this simple command very useful.

The Windows MessageBox is the next step up in complexity. It allows you to set the owner, the text in the message box, the caption and some flags. In the example below, there is no owner, the question about saving, "Save" is placed into the caption bar, finally the "yes", "no" and "cancel" buttons are created inside the dialog box.

```
i$ = MessageBox(NULL, "Do you want to save " + FileName$ + "?", "Save",
MB_YESNOCANCEL);
```

It is pretty obvious from the names below, which buttons get created when you use these constants.

```
MB_OK
MB_OKCANCEL
MB_ABORTRETRYIGNORE
MB_YESNOCANCEL
MB_YESNO
MB_RETRYCANCEL
```

You can also specify an Icon to be placed in the MessageBox from the list below. Simply use the "|" OR operator. For example MB_YESNOCANCEL | MB_ICONQUESTION

```
MB_ICONHAND
MB_ICONQUESTION
MB_ICONEXCLAMATION
MB_ICONASTERISK
MB_ICONWARNING = MB_ICONEXCLAMATION
MB_ICONERROR = MB_ICONHAND
MB_ICONINFORMATION = MB_ICONASTERISK
MB_ICONSTOP = MB_ICONHAND
```

MessageBox has additional features. Refer to MSDN for all of them.

MessageBox can return a value based when the button is pressed. These values are defined in Dialog.PPL.

```
#define IDOK         1
#define IDCANCEL     2
#define IDABORT      3
#define IDRETRY      4
#define IDIGNORE     5
#define IDYES        6
#define IDNO         7
```

I try to use these values when returning from displaying a dialog box. When using ShowModal to display a dialog any value less than 100 can be use for a button and it will return automatically, I try to avoid confusion and not change the meaning of the defined values.

Going back to my MessageBox example from above, "Yes" and "No" are pretty clear answers, but what about "Cancel"? If you are asking to save the file in response to a command to exit the program, the program flow go like this. User selects the Exit menu item. The menu exit handler would send a WM_CLOSE command. Then you need a handler for the close event.

In the Close event handler, call the MessageBox function and if you receive the "Cancel" ID, then return a FALSE from the OnClose event to stop the program from closing.

The operation of the OnClose event is a little different in windows than it is in PPL. Windows will only send you an OnClose event when the entire application is closing. PPL allows you to open multiple windows per application and will send you an OnClose event for each window. This makes it easy in PPL to implement the ability to "Cancel" for each window.

The next step is having more than one fully functional window for your application. For each new window that you create with the Form Editor, you need to:

- Select the Generate Library option in the Form -> Options menu.
- Change the form name so that a new windows class is created.
- Give each control on the forms a unique name.
- Give each control on the forms a unique ID. (required if getting the handles from the IDs)

For most forms you will probably want to select the "FormDefault" property. This property enables the use of the SIP (Soft Input Panel) on the PPC.

You may open several windows for your program at the same time, then manage input by selectively showing the window that is needed. To Hide or Show a window use:

```
ShowWindow(FormHandle$, SW_HIDE);  // or SW_SHOW
```

In creating more complex windows, you need to be careful about which styles are selected. The example below is how I added styles in the creation code for a window. This was necessary because the WS_EX_CaptionOKBtn style prevented my window from displaying on the PC as it is a PPC only style. So if your window refuses to display at all, you should review the styles that you have used and isolate the trouble-maker.

```
    ExStyles$ = GetWindowLong(SizeDlgHandle$, GWL_EXSTYLE);
#ifdef _WIN32_WCE
    // WS_EX_CAPTIONOKBTN is added here and not on the form because if added to
    // the form then the form will not display on the PC
    SetWindowLong(SizeDlgHandle$, GWL_EXSTYLE, ExStyles$ | WS_EX_CAPTIONOKBTN);
#else
    // WS_EX_TOOLWINDOW is added here for the same reason as above except it will
    // not show on the PPC if present.
    SetWindowLong(SizeDlgHandle$, GWL_EXSTYLE, ExStyles$ | WS_EX_TOOLWINDOW);
#endif
```

I hope that this collection of tips has helped.

Time to say goodbye

Thank you for your interest in PPL. We hope to see you in the forums and let's make it a source of good information, a library for PPL enthausiasts. If you any comments or questions, please visit our forums or contact us via support@arianesoft.ca

If you want to contribute to this newsletter with articles, tips, suggestions, please contact us, we will be happy to include them.

Regards,
Alain Deschenes
President
ArianeSoft Inc.
www.arianesoft.ca