Pocket Programming Language Newsletter
November 2006

Welcome to the monthly PPL newsletter. 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 seems PPL has been released 1 year ago for us. PPL has gone through several new released in a short amount of time. Part of this is because we have not done proper beta testing in the last phase of development. After releasing beta 0.97 in April 2006, we have extensively added new features into PPL but did insufficient testing before release. We realize it was not the best idea but the official release would have to be again pushed back for some time. We turned around and released multiple new versions (at least once a week since 1.0) to prove our dedication and to sort of apologize for the state PPL was in when released.

After releasing 1.06 at the end of October, we are preparing the release of 1.07 for early November which serves as a quick fix version for 1.06, then things should settle down a bit so we can focus on version 1.1. In version 1.1 we will add many new features based on requests that we have received since PPL was released. The new version, which is due out this month, will introduce Sockets, Bluetooth and Infrared support as well as a nice SQLite easy-to-use library. PPL now has an easy-to-use database development environment.

What's new

Demo contest is still on. $300 in prizes.

Our demo contest is still going on. The deadline is November 30th 2006. We are looking for the best graphical demo running on a PocketPC. The first prize is $150, the second $100 and the third prize is $50. Don't miss this chance to win big money and impress the community with your talent! Watch our forums and news section on www.arianesoft.ca for more information.

More news on PPL v1.1.

By the time you are reading this newsletter we have been actively working on PPL version 1.1 for more than 2 weeks. We should be releasing it in the third week of November. We don't have a fix date just yet since at the time we are writing this article it is still mid-October. Here is an update list of new features PPL 1.1 will have.

Here is brief unofficial list of what to expect:

- .OGG music file support is on the list.
- PIDE, component treeview browser with Classes, Controls, Menu Items, Procedures and Variables.
- Easy-to-use WinSock and Bluetooth library.
- Easy-to-use SQLite library.
- PIDE, find declaration, support for .frm files
- PIDE, find in files, support for .frm files.
- Particles sprite object in Game Level Editor.
- PocketOutlook library.
- Game Level Editor user interface sprites. (Like, button, editbox, listbox etc...)
- Updated Sprites function.
- New SetSpriteAnimSpeedVel to automatically adjust a sprite's animation speed based on its velocity.
- Registry library.
- Text to speech API library.

The following list will probably change and some features might be delayed until version 1.2.

Expect version 1.1 this November and 1.2 by the end of the year. Check our forums for updates later this month on www.arianesoft.ca/forum.php

Coding Tips section:

Put some structure in your life!

Life can be a real succession of disorder sometimes. Don't let this way of life turn your programs into nightmares. Put some structure into your code. Variables are a great way to organize and store information, but you need to classify this information into clean and organized structures to be able to keep your code expandable for the future.

A typical variable can be declared and accessed quite easily in PPL. You declare its scope (local or global) if wanted and then you assign values into it. Nothing new here. What if you have a whole lot of information you want to store into variables? Are you going to create one variable for each value you need to store? If you have answered yes to this question, you need to read further as you will discover that structured variables will give you benefits you probably never considered.

Let's pretend we need to write a simple three questions survey program. We need to gather user information first and then the user's answers to three questions. You could do the following:

```
Name$ = "Alain Deschenes";
Address$ = "Somewhere somehow";
Tel$ = "555-555-5555";
Age$ = "32";
Occupation$ = "Too busy";

Question1$ = "His answer #1";
Question2$ = "His answer #2";
Question3$ = "His answer #3";
```

This will probably turn into a real nightmare when you reach 500 lines of code or more. What if you spell a variable wrong? What if you need to add user information and questions?

With structured variables (called structures), you can group a series of variables into what you might call categories. In our scenario here, we would need a **user$** structure and a **questions$** structure. Each structure will hold a series of variables.

```
struct(user$, "Name", "Address", "Tel", "Age", "Occupation");
struct(questions$, "Question1", "Question2", "Question3");

User.Name$ = "Alain Deschenes";
User.Address$ = "Somewhere somehow";
User.Tel$ = "555-555-5555";
User.Age$ = "32";
User.Occupation$ = "Too busy";

Questions.Question1$ = "His answer #1";
Questions.Question2$ = "His answer #2";
Questions.Question3$ = "His answer #3";
```

Yes it is longer to write but if you use this technique you are guaranteed to get great benefits in the long run.

You can declare as many structure elements as you want inside the **Struct()** function. Each element you declare is by default a double type variable that can hold pretty much any numerical value. However you can change the type of element you need. There are multiple variable types that PPL can support including:

TBYTE : 1 byte value. Range from 0 to 255.
TSHORT : 2 bytes value. Range from 0 to 65535.
TINT : 4 bytes value.
TUINT : 4 bytes unsigned value.
TDOUBLE : 8 bytes value. Support decimal point.

You can create a structure that will hold 4 bytes with 4 elements of 1 byte each.

```
struct(mystruct$, "a", tbyte, "b", tbyte, "c", tbyte, "d", tbyte);
mystruct.a$ = 1;
mystruct.b$ = 2;
mystruct.c$ = 3;
mystruct.d$ = 4;
```

You can also define a custom number of bytes the element will hold. You can then access each byte of the element variable using [x] array syntax.

```
struct(mystruct$, "element", 256);
mystruct.element$[34] = 20;
```

You can also copy one structure to another variable by doing the following:

```
newstruct$ = mystruct$;
```

You can also pass structures as parameters of funcs or procs like this:

```
proc MyProc (s$)
  s.a$ = 10;
  s.b$ = 20;
  s.c$ = 30;
end;

proc main
  struct(mystruct$, "a", "b", "c");
  MyProc (&mystruct$);
  ShowMessage(mystruct.a$ + ", " + mystruct.b$ + ", " + mystruct.c$);
end;
```

The PPL Console (by Brad Manske)

A Console Program is a text only interface. It is still a windows program but it eschews the Graphical User Interface (GUI) and event oriented programming for the sake of simplicity. This is just the kind of program you want if your program just processes data and return results. In most cases, when testing the compiler it is quicker and simpler to use the PPL console than to write a Windows GUI program.

To see an example of a console program open the "PPL IDE" link in the PPL program group. Then select "Console..." from the file menu. The PPL console will evaluate what you type on the input line at the bottom and display the results in the output window.

Typing:

```
10+10<Enter>
```

for example, will display 20

It will also work with variables. Try this example:

```
a$=10
b$=20
a$+b$
```

The output window will display

```
> a$=10
10
> b$=20
20
> a$+b$
30
```

The expressions that PPL can evaluate can be quite complex in this mode and it makes for a handy tool. The real value in the console is using it in your own code. "Hello World" looks like this for a console program:

```
#include "console.ppl"
func WinMain
   InitConsole;
   ShowConsole;

   write("Hello World");

   return (true);
end;
```

Notice that this is a windows program, so it begins with **WinMain**. The Console is created and then made visible on the screen. The **Write()** statement sends the string to the console to be displayed. A **Writeln()** command also exists that will start a new line after the string has printed. The program ends by returning "true" so that the Console window stays open until the user closes it.

All that remains is to add your code in place of the write statement and you have a way to do unit testing on small pieces of your code. Here are a few string handling operations to get you going:

Write	- Send a string to the console
Writeln	- Send a string to the console then start a new line
+	- Concatenate 2 strings if alphanumeric ("ab"+"12"="ab12")
+	- add the value of 2 strings if numeric ("10"+"10"="20")
%	- Concatenate 2 strings ("ab"+"12"="ab12" or "10"+"10"="1010")
"\n"	- advance to the next line on the console

If you want even more control over how the console displays your data, then consult the manual for the "sprintf" statement. C language programmers will recognize this powerful formatting statement. For example:

```
MyValue$ = 1234;
sprintf(tmpString$, "Value printed in an 8 char field %8d",
MyValue$);
write(tmpString$);  // "    1234"
```

If you've followed along so far, you get rewarded with the best tip for using the console, which I have saved for last. **ShowMessage()** is often used to show the state of the program at some point to help with debugging. But sometimes it doesn't work or you spend all day clicking "OK" because you have to go through a large amount of data before you get to the point in the data where it doesn't work. Instead of dealing with all of that hassle, use the Console.

Create your window, and after the User Interface has been created add **InitConsole()** & **ShowConsole()**. Write out the debug statements and before exiting save the console to a file. Now you can search for the case you're interested in with a text editor.

You should enclose all of the Console calls in **#ifdef** statements so that they can easily be removed for a production build.

```
#undefine ProductionBuild   // change to #define for no console

#ifndef ProductionBuild
#include "console.ppl"
#endif

func WinProc

  // your code - create UI or call the form creation.
```

```
#ifndef ProductionBuild
  InitConsole;
  ShowConsole;
#endif

  // your code

#ifndef ProductionBuild
  Writeln("Show interesting data in your code.");
#endif

  return(true);
end;
```

That is the quick run down on the console. I've never tried using the Console to log the progress inside a game. I'm hoping that some game designer out there will give this a try. If you do, please tell us all about it in the Forums at http://www.arianesoft.ca/forum.php.

How to launch an executable from PPL (by Eric Pankoke)

For any number of reasons, you might want to launch an external program from your PPL application. It's actually rather simple to do. First of all, if you're not writing a GUI application, you need to include the following file:

```
#include "windows.ppl"
```

If you want to launch a program and have it open normally, here's a quick function to do so:

```
proc LaunchProgram(path$)
  #ifdef _WIN32_WCE
    path$ = wide(path$);
    verb$ = wide("open");
  #else
    verb$ = "open";
  #endif

  struct(info$, SHELLEXECUTEINFO);

  info.cbSize$ = sizeof(info$);
  info.lpFile$ = &path$;
  info.nShow$ = SW_SHOWNORMAL;
  info.fMask$ = SEE_MASK_NOCLOSEPROCESS;
  info.lpVerb$ = &verb$;
  result$ = ShellExecuteEx(&info$);
end;
```

For more details on how this works, look up the **ShellExecuteEx()** function on MSDN. To launch an application, you use "open" for the **lpVerb** member of the **SHELLEXECUTEINFO** structure. Other supported verbs are dependent on the program that you are attempting to launch, and it will be up to you to figure those out. **Path$** should be a fully qualified path / file name combination. Below is a quick demonstration that you can use in a non-GUI application to see how this works:

```
func WinMain()
  #ifdef _WIN32_WCE
    LaunchProgram(GetWinDir() + "addrbook.exe");
  #else
    LaunchProgram(GetWinDir() + "\\notepad.exe");
  #endif

  return(false);
end;
```

PPL Game Programming - Part 2

Draw me a picture I don't get it!

The GameAPI screen is represented by a series of pixels organized on an x and y axis. Position (0, 0) is the top left of the screen and (240, 320) is the bottom right for the typical **QVGA** display on a PocketPC and (480, 640) is the bottom right for a typical **VGA** display device.

Drawing to the screen is very simple with the GameAPI, you just need to know where to place the drawing code. Since the first part of our series of articles on the GameAPI talked about a basic code template to create a GameAPI program, we need to focus a little more on the **WM_PAINT** event here. The **WM_PAINT** event is called every frame the GameAPI needs to draw to the screen. The **WM_PAINT** is placed in the game procedure code, like this:

```
func GameProc(hWnd$, Msg$, wParam$, lParam$)
  case (Msg$)
    WM_PAINT:
      G_Clear(0);
  end;
end;
```

When you initialize the GameAPI you pass the **GameProc** pointer to the **InitGameApiEx()** function. PPL will then use this function to trigger custom events like **WM_PAINT**, **WM_TIMER** and **WM_COLLIDE**.

```
InitGameAPIEx(h$, &GameProc, 240, 320, false, 5, 60);
```

Inside the **WM_PAINT** code you can put any type of drawing code you want, like **g_clear(0)** to clear the screen with a color you like, **g_textout()** to draw informative text and **g_fillrect()** to draw a rectangle. PPL comes loaded with a ton of drawing functions. PPL clears the screen in black by default if no **WM_PAINT** event is defined.

What if you want to draw something on the screen outside the **WM_PAINT** event code? It's easy, but you need to follow some guidelines. You need to prepare the screen to be drawn to and when done you need to update the screen. Here is simple code to draw a rectangle, wait 5 seconds and then return to normal drawing of the screen either by triggering the **WM_PAINT** code or by simply clearing the screen with black.

```
g_beginscene;
g_fillrect(10, 10, 100, 100, g_rgb(100, 100, 100));
g_update;
delay(5000);
```

Be careful not to call **g_beginscene()** without calling a corresponding **g_update()**. Follow this rule and you will never have any problems.

Refresh your game with a sprite!

Now is the time to spice up your game knowledge and get into something really cool: Sprites. Remember the old days of the Nintendo (NES) or the Super Nintendo (SNES) video game consoles? Most games you played back then were using sprites. Sprites are basically just an image you can move around and animate. In PPL sprites are pretty advanced. You can stretch them, tint them, tile them, automatically animate them, and so much more...

Let's first start by loading an image from disk as a sprite:

```
MySprite$ = LoadSprite(AppPath$ + "mysprite.bmp", G_RGB(255, 0, 255),
4, 150, NULL);
```

You always need to retrieve the sprite handle from the **LoadSprite()** function to be able to access it later on. The sprite handle is just a unique integer value.

The first parameter of the **LoadSprite()** function is the pathname of the image you wish to use. The image file can be a bitmap (.bmp), a jpeg (.jpg), a Portable Image (.png) or a gif (.gif). The image file can be made of multiple images grouped together in one image file.

The second parameter is the transparent color to use. Sprites can be drawn on the screen with a transparent background to make them blend with the scenery.

The third parameter is the number of frames (images) the image file has. The images must be sequential (one after the other) horizontally within the file, and they all must be the same width and height in pixels.

The fourth parameter is the speed in milliseconds at which the animation will be played. The default animation will swap between each frame one after the other from the left to the right.

The last parameter is the sprite procedure to use for the sprite. We will get into more advanced sprite handling in a later article.

The image is by default visible on the screen at position (0, 0). You can hide or show the sprite using the following:

```
DelSpriteOption(MySprite$, SO_VISIBLE); // Hide the sprite by
removing the SO_VISIBLE flag from the sprite's options.
AddSpriteOption(MySprite$, SO_VISIBLE); // Show the sprite by adding
the SO_VISIBLE flag to the sprite's options.
```

Each sprite has a series of special options that can be removed or added at any time.

Now let's move our sprite on the screen, to move the sprite all you need to do is call the **MoveSprite()** function and pass a new coordinate. In our case we will move the sprite to where the stylus touches the screen.

In the **MainProc** of our code we will add a **WM_LBUTTONDOWN** event that will be triggered whenever the stylus touches the screen:

```
WM_LBUTTONDOWN:
  MoveSprite (MySprite$, wParam$ - (SpriteWidth(MySprite$) / 2),
lParam$ - (SpriteHeight(MySprite$) / 2));
```

Here we center the sprite **MySprite$** - which we should have made global at the time of loading (LoadSprite) - around the stylus position. **SpriteWidth()** returns the width in pixels of the sprite and **SpriteHeight()** return its height in pixels.

The **WM_LBUTTONDOWN** uses the **wParam$** variable to store the X coordinate position and the **lParam$** variable for the Y coordinate position the stylus was pointing to. You can write code for the following events:

WM_LBUTTONDOWN: The stylus is pressed.
WM_LBUTTONUP: The stylus is released from the screen.
WM_MOUSEMOVE: The stylus is being moved around while pressed.

You can stretch a sprite's display by doing the following:

```
SetSpriteWidth (MySprite$, 100);  // Makes the sprite 100 pixels
wide.
SetSpriteHeight (MySprite$, 200);  // Makes the sprite 200 pixels
high.
```

You can change the animation speed and sequence of a sprite, lets say you have frames for jumping at frame 6 to 10:

```
SetSpriteFrames (MySprite$, 6, 10, 250, true);
```

This will set the current animation for **MySprite$** from frames 6 to 10, animating at 250 milliseconds. The last parameter specifies if PPL should wait for the current animation timer to expire before going to the new animation frames or change right away.

What are your origins?

The screen display can be moved around. The starting origins are at coordinates (0, 0). You can move the screen in any direction. All the sprites will move according to the screen origins. You can design maps with lots of sprites on them, then scroll the whole map just changing the origin values.

```
SetOriginX(10);
SetOriginY(-10);
```

If you need sprites such as interface icons to remain at certain physical screen coordinates (always visible), rather than coordinates that are relative to the origin, you will need to add the following options to the sprites:

SO_FIXED, SO_FIXEDX or SO_FIXEDY.

SO_FIXED will keep the sprite at the pixels they are assigned to, even if the origins of the screen are changed. **SO_FIXEDX** will keep the X axis of the sprite fixed while the **SO_FIXEDY** will keep the Y axis location of the sprite fixed.

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 enthusiasts. 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