Migrating 32-bit Applications to 64-bit

Contents

1 Introduction 1
2 Migration Justification 2
    2.1 Theoretical Benefits 2
    2.2 Determining Migration ROI 2
3 Migration Considerations 4
    3.1 PB IDE Limitations 4
        3.1.1 Database Connection 4
        3.1.2 Developing 5
        3.1.3 External Objects and Controls 5
        3.1.4 PowerBuilder Native Interface (PBNI) 5
        3.1.5 Testing & Debugging 5
        3.1.6 Deployment 5
    3.2 Unsupported Features 5
4 Migration Steps 7
    4.1 Step 1 - Database Connections 7
    4.2 Step 2 - Datatypes 7
    4.3 Step 3 - Unsupported Features 8
    4.4 Step 4 - Objects and Controls 9
    4.5 Step 5 - PowerScript Functions 10
    4.6 Step 6 - Compilation and Deployment 11
        4.6.1 Client/Server Deployment 11
        4.6.2 PowerClient Deployment 12
        4.6.3 PowerServer Deployment 13
5 Conclusion 14

1 Introduction

This guide is focused on migrating existing PowerBuilder 32-bit applications to a 64-bit architecture. We will walk-through determining if it makes sense to migrate, the key migration issues you may face, and the key steps to achieve a successful migration.

This guide assumes your applications have already upgraded to PowerBuilder 2019 R3 or higher. The reasons for this is that older versions of PowerBuilder did not provide robust support for the 64-bit project type and also older versions are now end of life (EOL).

2 Migration Justification

2.1 Theoretical Benefits

On paper 64-bit applications sounds great! If we compare a 32-bit processor against a 64-bit processor, the 64-bit processor can process a larger set of data, while also being faster and more responsive. The technology world is trending towards 64-bit. In addition, the step-up to 64-bit enables greater compute capabilities that are needed to meet the demands of today's greater workloads. Other benefits also include, but are not limited to:

- Access to larger amounts of memory (and thus, less disk swapping);
- A richer instruction set (i.e. more CPU registers – for faster handling of arrays and passing data without using a "stack");
- Enhanced security features (e.g. Kernel Patch Protection, Support for hardware-backed Data Execution Protection (DEP), Mandatory driver signing);
- Greater precision in 64-bit numbers and calculations;
- Interoperability (supports both 64 & 32-bit operations);
- Policy compliance with corporate requirements; and
- Performance increase in calculations per second.

2.2 Determining Migration ROI

But before migrating your existing applications, it is critical to ask yourself: "What is the ROI of migrating my application to a 64-bit architecture?" You really need to answer this question because converting an existing 32-bit application to a 64-bit architecture is not just a matter of recompiling your application. Typically, it involves substantial work, requiring source code changes, configuration changes, possibly replacing third-party software required by your application, and then fully retesting your application, just to name a few of the key tasks.

The following is a short list of some of the scenarios that may justify the move to a 64-bit architecture, which might help you to make the right decision:

- **Life-cycle**

  If you plan a prolonged development and maintenance of your application. In other words, it probably does not make sense to migrate to 64-bit if you will sunset your application in the next few years or less.

- **Performance requirements**

  If your application is memory-intensive, then its execution speed is likely to increase a minimum estimate of 5%-15%. This is achieved due to architectural features of the 64-bit processor and availability of more RAM. Plus a performance gain by the absence of the WoW64 (Windows on Windows) layer that continually translates calls between 32-bit application and the 64-bit operating system.

- **Interoperability**

  Applications that require specific 64-bit third-party libraries

  - Applications tend to be constantly evolving and sometimes, there are very specific needs for interfacing with third-party libraries or software packages that are specifically developed for a 64-bit architecture. Such requirement justifies migration because generally speaking 32-bit processes cannot interact with 64-bit software directly during execution.

  Third-party applications that require connecting to your application

  - If you are developing libraries, components or other items intended for third-party developers to create software with, or simply to interface with your applications natively, you will need to create a 64-bit version of your product. Otherwise, your customers interested in 64-bit versions will have to search for an alternative product.

- **Policy requirements (Internal and External)**

  There is an increasing initiative for enforcing 64-bit architecture in the industry. This is in response to higher requirements for performance, security, interoperability and infrastructure. Industry statistics predict that 64-bit OS upgrades are one of the top-budgeted IT initiatives in the next two years*, and companies have shown to purchase more software upgrades during budgeted OS upgrade initiatives than any other time. **Per a Gartner report referenced by* DailyTech.com

It is important to keep in mind that migrating to 64-bit should not be the first answer to deal with technical issues, such as performance and interoperability. While there are performance benefits, improperly designed code will continue to be problematic after migration. And interoperability can often times be achieved by using open standards (e.g. REST APIs) to interface with other software.

3 Migration Considerations

When migrating 32-bit PowerBuilder applications to a 64-bit architecture, you need to consider many factors. Some of the key things to consider is the development environment, unsupported features, and deployment differences. In other words, it is prudent to do a technical feasibility enhancement of such migration before you really begin.

In performing such assessment, it is vital that developers knowledgeable about the source code of the application carefully go through the code to identify the issues. The reason for this is that the PowerBuilder IDE does NOT identify every single possible migration issue.

3.1 PB IDE Limitations

The PowerBuilder IDE is only available as a 32-bit application. This IDE bitness then affects a number of things, for example how you develop, test, debug, and deploy your applications.

3.1.1 Database Connection

The PowerBuilder IDE can only connect to 32-bit database drivers (since it is 32-bit). A compiled 64-bit version of your application, however, will require a 64-bit database driver. Thus, it is recommended to have BOTH the 32 and 64 bit database drivers installed on your development machine. The client database driver only needs to match the bitness of your application and does not need to match the bitness of the database server itself. In other words, a 64-bit driver can connect to a 32-bit database server and vice versa.

The following is a short list of the 3 most popular database drivers you can directly download. For other database drivers, please refer to your database vendor's website.

1. SQL Server (read more)

   MSOLEDBSQL SQL Server 18.5

   Download Microsoft OLE DB Driver for SQL Server (x64)

   Download Microsoft OLE DB Driver for SQL Server (x86)

   Microsoft ODBC Driver for SQL Server

   Download Microsoft ODBC Driver 17 for SQL Server (x64)

   Download Microsoft ODBC Driver 17 for SQL Server (x86)

2. SAP SQL Anywhere (read more)

   SAP SQL Anywhere 17.0 ODBC driver

   Windows x86 and x64

   SAP SQL Anywhere 16.0 ODBC driver

   Windows x86 and x64

3. Oracle Instant Client (read more)

   Instant Client for Microsoft Windows (x64)

   Instant Client for Microsoft Windows (32-bit)

3.1.2 Developing

Application Target

There is no special PowerBuilder Target object for 64-bit applications. To build a 64-bit application, select the 64-bit option in the Project Painter’s General Tab instead of the default (32-bit). If you need to deliver both 32-bit and 64-bit versions of your application, you should use separate projects, separate folders and separate executable names for the deployed output. (read more)

3.1.3 External Objects and Controls

Since the IDE is 32-bit, this introduces some challenges with 64-bit external objects and controls. You can use OLE and ActiveX components in your applications, but you must use the 32-bit versions in the PowerBuilder IDE. At runtime you must have the correct 64-bit ActiveX components installed. (read more)

3.1.4 PowerBuilder Native Interface (PBNI)

Since PowerBuilder IDE is 32-bit, this introduces some challenges with PBNI interfacing with 64-bit. In the PowerBuilder IDE, you can only import the 32-bit PowerBuilder extensions into the PBL (using the **Import PB Extension** context menu) or create the PBD file from the 32-bit PBX file using the pbx2pbd210.exe tool. But for runtime, you can still package and distribute the 64-bit extension libraries with your 64-bit applications. What you need to do is compile the extension libraries to 64-bit PBX files using the Visual Studio Wizards, and make sure your 64-bit PBX files have the same name as the 32-bit files, since the application references it by file name. (read more)

3.1.5 Testing & Debugging

Since the PowerBuilder IDE is 32-bit, to test the 64-bit features of your application you must compile it as a 64-bit executable and run the resulting executable. Running the application within the PowerBuilder IDE will run the project as a 32-bit application. But you can still compile and distribute your application as native 64-bit. (read more)

The debugger in the PowerBuilder IDE is only able to debug the 32-bit version of your application. A 64-bit debugger is tentatively planned for PowerBuilder 2022. So for the time being, if you have a 64-bit specific bug that cannot be reproduced in 32-bit, you will have to use more rudimentary troubleshooting techniques, such as the infamous "message box".

3.1.6 Deployment

During the deployment process, PowerBuilder checks and reports key unsupported features used in the application. However, the PowerBuilder IDE does NOT identify every single possible issue that must be addressed for successful migration. So it is vital to not solely rely on this report. (read more)

3.2 Unsupported Features

The following is a list of key unsupported features to consider before migrating to 64-bit architecture: (read more)

- Web Services Client (Obsolete)

  Creating Web service proxy for connecting to SOAP server is no longer eligible for technical support. Additionally there is no 64-bit version of the client. So, **you must** *use the PowerBuilder* HTTPClient Object instead. You can follow these instructions: Call SOAP Web Services using HTTPClient Object in order to successfully replace the Web Services Client before migrating your application to 64-bits.

- COM+ runtime (read more)

  Building a COM+ client in 64-bit is currently not supported.

- Machine code generation

  Generating PowerBuilder machine code DLLs is not supported for 64-bit applications. But you can still opt for compiling as P-code which is supported (read more).

  This feature is tentatively planned for PowerBuilder 2022.

- TabletPC

  These TabletPC features are not supported on 64-bit applications:

  - InkPicture (read more)
  - InkEdit (read more)
  - Gesture PowerScript Event (read more)
  - UseMouseForInput property for PowerScript controls (read more)

  Currently there are no substitute for these features.

- PBNI SDK

  Currently there is no 64-bit version of the PBNI SDK.

- Application server support (read more)

  PowerBuilder developers can build clients that invoke the services of COM+ and third-party application servers, and build components (or objects) that execute business logic inside each of these servers.

  Currently it is not supported to create 64-bit components for application servers with the PowerBuilder IDE. However, you can use the SnapDevelop IDE (bundled with PowerBuilder) to create 64-bit components in C# with DataWindow technology.

4 Migration Steps

To migrate 32-bit PowerBuilder applications to a 64-bit architecture, code changes are almost always necessary, and the degree of such changes depends on how your application has been written. We strongly recommend you systematically follow the 6 steps below to make the necessary code changes: Datatypes, Objects and Controls, Database Connections, PowerScript Functions, Compilation, and Deployment.

4.1 Step 1 - Database Connections

Install BOTH 32-bit and 64-bit database drivers to your development machine. As already mentioned in this guide, the PowerBuilder IDE requires 32-bit database drivers while your compiled app will require 64-bit database drivers.

In addition, make the necessary changes to your database connection as applicable.

- For ODBC connections:

  If you are using ODBC drivers to connect to your database, then you need to create the ODBC data source for your 64-bit driver. Follow the instructions here.

  If your new ODBC data source requires any special settings, like specific database parameters, apply those changes to the newly created data source. For a complete list of database parameters read the Database parameters and supported database interfaces online document.

- For SAP Sybase Open Client Client-Library (CT-Lib)

  You need to specify the following database parameter:

  - **Release** (read more)

    - Specifies what version of SAP Sybase Open Client Client-Library (CT-Lib) software is in use on the client workstation.

    - Set Release to 15 and use Open Client 15 or higher and Adaptive Server 15 or higher to access the UniText and 64-bit integer (BigInt) SQL datatypes added in version 15 of Adaptive Server.

4.2 Step 2 - Datatypes

Find the variables whose datatypes needs to be upgraded in order to support 64-bit and change them as applicable.

- **Longptr**

  In the 32-bit platform, longptr is the same as long; you can continue using long wherever longptr is required in 32-bit applications. In 64-bit applications, however, using long to hold longptr variables will lead to data truncation from 8 bytes to 4 bytes, or memory corruption if you pass a long ref variable when a longptr ref is required. If you want to migrate to 64-bit, use longptr wherever required as it does no harm to 32-bit.

  Since PowerBuilder does not have a datatype corresponding to the C++ pointer type, and there are no pointer operations in PowerBuilder, longptr is not a full-fledged PowerBuilder datatype. You can use it to hold/pass window handles, database handles, and other objects that are essentially memory addresses. Doing complex operations on longptr type might not work. If you want to represent/compute 8-byte long integers, use longlong.

- **LongLong**

  To take advantage of the 64-bit architecture, you can change your **Long** datatype variables to **LongLong**. Because this datatype is for 64-bit signed integers, from -9223372036854775808 to 9223372036854775807.

  Keep in mind that **LongLong** continues using literals as for integers, but longer numbers are permitted.

- **Using 1-byte structure member alignment in external function** (read more)

  When you use a structure or structure array as parameters to external function in the older versions of PowerBuilder, the structure member alignment was one byte. However, the default alignment is 8 bytes on Windows platform, which means that most (if not all) Windows standard APIs use this value to align arguments with structure members. This will cause a mismatch between Windows APIs and PB applications in PowerBuilder 12.5 and earlier versions. A well adopted solution to this issue was to add some bytes within PowerBuilder structures manually to fill those gaps. Such gap filling can be complex and error-prone if involving complex nested structures. And what is worse, this solution fails with the introduction of 64-bit application development in PowerBuilder because the number of bytes you have to fill may be different between 64-bit and 32-bit applications. This was the major reason to make this change in PowerBuilder. The default structure member alignment for Windows APIs and Visual C++ is now on an 8 byte boundary. The structure member alignment was changed to 8 bytes for both 64-bit and 32-bit applications. This was an intentional change so that you can now call Windows APIs easier and use the same code for both 64-bit and 32-bit applications.

  Customers can switch to the old behavior in two ways:

  1. Check "Use 1-byte structure member alignment in external function" (or set UseZp1=1 in [pb] section, pb.ini, the results are the same). The effect is global with this setting changed. To make this work at runtime, please remember to deploy your pb.ini file with your application.

  2. Add "progmapack(1)" external function’s declaration, like this: FUNCTION int STLAREGIO (ref struckfzrechnerneu struckfz) LIBRARY "KFZSS.DLL" alias for "STLAREGIO;Ansi" progma_pack(1)

  progmapack(1) is 1-byte alignment, progmapack(8) is an 8-byte boundary (double-word) alignment. These settings only affect an external function declaration’s alignment usage.

4.3 Step 3 - Unsupported Features

Remove and/or remediate unsupported features. The key unsupported features are documented in Section 3.3 of this guide, and will also be reported when you attempt to deploy your application to 64-bit.

4.4 Step 4 - Objects and Controls

Look for the following objects and controls in your application and make the changes as applicable.

- **RichTextEdit** (read more)

  Select the Built-in TX Text Control ActiveX 28.0 (32-bit and 64-bit) which is used as the rich text editor for both 32-bit and 64-bit applications; otherwise, the obsolete Microsoft RichEdit Control will be used as the 64-bit rich text editor, which is very different from the 32-bit TX TextControl.

  The built-in TX Text Control ActiveX is also used to support RichText-based DataWindows.

  To select the Built-in TX Text Control ActiveX 28.0 follow the instructions here.

- **External** (OLE's, OCX, WinAPI, etc...) (read more)

  You can use OLE and ActiveX components in your applications, but **you must use** the 32-bit versions in the PowerBuilder IDE. At runtime **you must have** the correct 64-bit ActveX components installed.

- **PBNI** (read more)

  You can use PBNI in your applications, but **you must use** the 32-bit extensions in the PowerBuilder IDE. At runtime **you must have** the correct 64-bit extensions installed. The file names of your 64-bit extension should match the 32-bit file names, since the application references it by file name.

- **MAPI** (read more)

  PowerBuilder supports MAPI (messaging application program interface), so you can enable your applications to send and receive messages using any MAPI-compliant electronic mail system. However, you need to keep in mind the following considerations when using MAPI with a deployed 64-bit application: 64-bit PowerBuilder mail applications can only work with 64-bit Windows MAPI. 32-bit PowerBuilder applications can only work with 32-bit Windows MAPI. (read more)

- **Environment Object** (read more)

  The Environment object is a system structure used to hold information about the computing platform the PowerBuilder application is running on. You populate the Environment object using the GetEnvironment function. (read more)

  This feature becomes very useful when deploying, running and testing applications compiled as 64-bit because you can determine the bitness of your application using the *ProcessBitness* property. (read more)

- **PFC**

  If you are using the PFC framework (in part or whole) then you need to make sure you are using the latest version. The PFC framework should only require few Window API calls to be modified in order to work properly in a 64-bit environment. The following are the particular API calls that PFC makes, and the only real difference is that you need to change the value of the size of structures attribute passed in on the call.

- **ToolTipMsg:** for *ToolInfo.cbSize* pass 64 value for 64-bit and 40 value for 32-bit.
- **TrackMouseEvent:** for *TraceMouseEvent.cbSize* pass 24 value for 64-bit and 16 value for 32-bit.
- **ShellExecuteEx:** for *ShellExecuteInfo.cbSize* pass 112 value for 64-bit and 60 value for 32-bit.

4.5 Step 5 - PowerScript Functions

Apply the following changes to your PowerScript functions, if found in your source code, in order to use the corresponding 64-bit features.

- If you are passing **Structures** to **External Functions**: (read more)

  You can pass PowerBuilder structures to external C functions if they have the same definitions and alignment as the structure's components. The DLL or shared library must be compiled using byte alignment; no padding is added to align fields within the structure.

  The default structure member alignment for Windows APIs and Visual C++ is now 8 bytes. The structure member alignment was changed to 8 bytes for both 64-bit and 32-bit applications. This was an intentional change so that you can now call Windows APIs easier and use the same code for both 64-bit and 32-bit applications.

  This structure member alignment was covered in Step 1.

- If you need to use the Registry to store a value of type **RegLonLong!** then use the following PowerScript function:

  RegistrySet (key, valuename, valuetype, value) (Source)

  valuetype

  - **RegLongLong!** -- A 64-bit number which is the longlong type ranging from 0 -9,223,372,036,854,775,807, because the registry key value cannot accept a negative number.

- If you need to use the Registry to get a stored value of type **RegLongLong!** then use the following PowerScript function:

  RegistryGet (key, valuename {, valuetype }, valuevariable) (Source)

  valuetype

  - **RegLongLong!** -- A 64-bit number which is the longlong type ranging from 0 -9,223,372,036,854,775,807, because the registry key value cannot accept a negative number.

- If you need to convert two **Long** datatype values into a **LongLong** datatype, use the following method:

  LongLong

  Combines two unsigned longs into a longlong value.

Example:

- UnsignedLong lLow //Low long 32 bits
- UnsignedLong lHigh //High long 32 bits
- longlong LLValue //LongLong value 64 bits
- lLow = 1234567890
- lHigh = 9876543210
- LLValue = LongLong(lLow, lHigh)
- MessageBox("LongLong value", LLValue)
- Combines two unsigned longs into a longlong value.

4.6 Step 6 - Compilation and Deployment

To compile your 64-bit PowerBuilder application you first need to decide what deployment method you will use. PowerBuilder supports 3 deployment methods, and we strongly recommend considering the last 2 options while you are in process of investing to modernize your existing application:

1. Client/Server (*manual client/server deployment*) (read more)
2. PowerClient (*automated client/server deployment*) (read more)
3. PowerServer (*cloud-native app deployment*) (read more)

4.6.1 Client/Server Deployment

Create a Project Object for the specific 64-bit version of your application.

- The Project painter for executable applications allows you to streamline the generation of executable files and dynamic libraries.
- You can select if the executable runs on 32-bit or 64-bit machines by setting the Platform option. (read more)

If you automate the build of your application, you can use the PBC (standalone compiler), ORCA, or PBAutoBuild:

- PBC - If you will be using the PowerBuilder Compiler then you must use the "/x 64" parameter in order to compile your 64-bit application; otherwise, a 32-bit executable is created by default. Here is a simple example: pbc190 /d MyApp.pbt /x 64.
- ORCA - If you will be using the ORCA then you need to use the "x64" argument in order to compile your 64-bit application; otherwise, a 32-bit executable is created by default. Here is simple example: build executable exeName iconName pbrName pbdflags [machinecode] [newvstylecontrols] [x64].

- PBAutoBuild - (available in PowerBuilder 2021 and newer) If you will be using PBAutoBuild210 then you need to set the "Platform" property to 1 in the build file in order to compile your 64-bit application; otherwise, a 32-bit executable is created by default. Here is simple example: PBAutoBuild210 /f buildFile /l logFile /le errorFile". (read more)

Once your application is compiled, you will also need to manage packaging & distributing the appropriate runtime files to your end users. Since this is a manual process, it is important to carefully follow the Installation Checklist, and we strongly recommend using the Runtime Packager to minimize chance that required runtime DLLs are missing.

If using the Runtime Packager:

- Select the proper **PowerBuilder 64-bit Components** in order to generate the runtime package needed for 64-bit apps.

- Keep in mind where your application will look for the runtime files. The runtime file location is no longer recorded in the PATH environment variable; instead it is recorded in the system registry: HKEYLOCALMACHINE\SOFTWARE\Sybase\PowerBuilder Runtime. (read more)

If manually packaging the runtime DLLs:

- The core runtime files (and majority of additional runtime files) can be used universally for 32-bit and 64-bit projects. So you can package these as before.

- If the application uses RichText features, package either the TX Text Control ActiveX 28.0 (64-bit) or Microsoft RichEdit Control (64-bit) runtime DLLs, as applicable. (read more)

- Include the 64-bit version of the Visual C++ runtime, Active Template, and GDI+ libraries (e.g. msvcr100.dll, msvcp100.dll, atl100.dll, etc.). These DLLs are located in %AppeonInstallPath%\Common\PowerBuilder\Runtime [version]\x64.

The best practice is to place all the runtime DLLs in one folder and then place the folder in the same directory as the application executable, so that you can flexibly switch runtime packages in the configuration file (*executable-name*.xml). (read more)

4.6.2 PowerClient Deployment

Create a PowerClient Project Object for the specific 64-bit version of your application: (read more)

- Once you have created a PowerClient project, you can open it from the System Tree and modify the properties as necessary.

- In the General Page of the PowerClient Project Object, you can select if the application compiles to 32-bit or 64-bit by setting the Platform option accordingly. Follow these instructions.

If you automate the build of your application, you MUST use the PBAutoBuild (available in PowerBuilder 2021 and newer):

- PBAutoBuild - (available in PowerBuilder 2021 and newer) If you will be using PBAutoBuild210 then you need to set the "Platform" property to 1 in the build file in order to compile a 64-bit version; otherwise, a 32-bit version is created by default. Here is simple example: PBAutoBuild210 /f buildFile /l logFile /le errorFile". (read more)

Once your application is compiled, you do not need to manage packaging & distributing the appropriate runtime files to your end users. This is automatically handled for you by PowerClient.

4.6.3 PowerServer Deployment

Create a PowerServer Project Object for the specific 64-bit version of your application: (read more)

- Once you have created a PowerServer project, you can open it from the System Tree and modify the properties as necessary.
- In the General Page of the PowerServer Project Object, you can select if the application compiles to 32-bit or 64-bit machines by setting the Platform option accordingly. Follow these instructions.

If you automate the build of your application, you MUST use the PBAutoBuild (available in PowerBuilder 2021 and newer):

- PBAutoBuild - (available in PowerBuilder 2021 and newer) If you will be using PBAutoBuild210 then you need to set the "Platform" property to 1 in the build file in order to compile a 64-bit version; otherwise, a 32-bit version is created by default. Here is simple example: PBAutoBuild210 /f buildFile /l logFile /le errorFile". (read more)

Once your application is compiled, you do not need to manage packaging & distributing the appropriate runtime files to your end users. This is automatically handled for you by PowerServer.

5 Conclusion

These days there probably is not many good reason to create a new 32-bit application, but migrating existing applications to 64-bit is not so clear cut. From what can be seen by Microsoft’s actions over the years, it appears they are strongly committed to ensuring the tremendous investments their customers have made in building Windows apps continue to work properly for the long-run. For example, Windows 10 runs both 32-bit and 64-bit apps, and there has been zero indication Microsoft has plans to discontinue 32-bit support. On the other hand, as this guide has outlined, significant work and tradeoffs may be required to migrate your PowerBuilder application to a 64-bit architecture. As such, making the decision to migrate to 64-bit simply to stay up to date may not necessarily be the best decision for your business.