DataAccess WORLDWIDE

Working with the DataFlex 2020 Technology Preview

What is DataFlex 2020?

DataFlex 2020 is the start of the newest generation of DataFlex, allowing your applications to take advantage of both 64-bit and full Unicode support.

The DataFlex 2020 Technology Preview is an early pre-release build of the latest revision of DataFlex, formerly referred to as "DataFlex NextGen". The purpose of the Technology Preview is to give interested developers an early experience with the new 64-bit and Unicode capabilities of DataFlex 2020 and experiment with migration of their existing DataFlex applications.

Unlike traditional Alpha or Beta releases where the main development phase is complete, DataFlex 2020 will still be undergoing significant changes while this Technology Preview is available. We'll discuss the areas of the product that are still "under construction" later in this document.

How to get the most out of the Technology Preview

Working with the Technology Preview is a straightforward process once you have a basic understanding of the differences between DataFlex 2020 and prior releases of the product. It's also important to understand which parts of the product are still "under construction". Our advice is to go step-by-step as you gain more familiarity with the Technology Preview…

Step 1 – Install the Technology Preview

Just as with all previous releases, you can safely install DataFlex 2020 side-by-side on a machine with previous revisions of DataFlex. Since the Studio is mainly a 64-bit product it defaults to the Program Files folder instead of Program Files (x86).

The Technology Preview uses its own special license, there is no need to install a registration code.

Please note that the Technology Preview currently uses the same revisions of the Codejock components (18.3.0) as DataFlex 19.1, so if the last action you take on a system is to uninstall either the Technology Preview or DataFlex 19.1 you will need to manually register those components for the remaining installation to function correctly. You can use the "RegisterCodejockControls" batch file in the Bin folder to do this (you need to run this batch file using "as Admin").

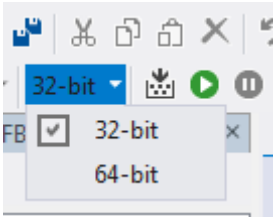
Step 2 – Read this Document

We know it's not the most exciting part of experimenting with a new toy, but your experience with the Technology Preview will be more satisfying if you take the time to review this document completely.

Step 3 – Use the Studio

We'll assume that you are already familiar with DataFlex 19.1 (if you are jumping into DataFlex 2020 from earlier revisions there will be more to get used to, depending on your usual working environment). Here are some areas to pay particular attention to:

- While the Studio itself is 64-bit, you can compile and debug both 64-bit and 32-bit applications. This is done on a project by project basis in a workspace and you can control this easily from the drop-down in the toolbar. You can also set the default for any project in the Compiler tab of Project Properties.



- When looking at the Project Properties, note that in addition to the current mode, there are optional suffixes for the compiled output. Our recommendation is to leave the 32-bit suffix blank and use "64" for the 64-bit suffix. Note that since web applications are always WebApp.exe, you must not set any suffix for either 32 or 64-bit.
- One aspect of working with DataFlex 2020 that is not readily apparent is that once source is modified with the Studio it is now UTF-8 encoded instead of OEM. The Studio will automatically insert a Byte Order Mark (BOM) at the top of every source file it touches. The differences in encoding in all areas of the product are fundamental to understanding Unicode support in DataFlex 2020 and your applications, so make sure you review the sections on Unicode in DataFlex and Character Encoding.

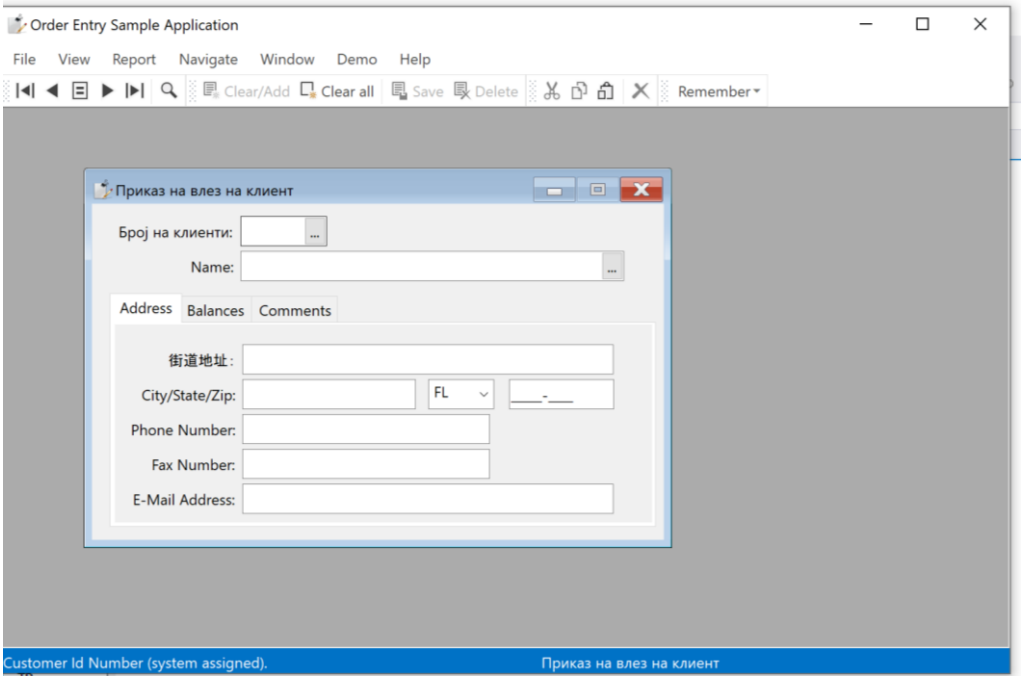
Step 3 – Experiment with the Examples

The main example workspaces all support compilation in both 64 and 32-bit. They also support Unicode but you need to convert the data to SQL Server to store Unicode data. While the examples are relatively simple (by design), they are a great way to play around with the Unicode features of DataFlex 2020 for that very reason; in just a few minutes you can experience what a fully Unicode application will feel like!

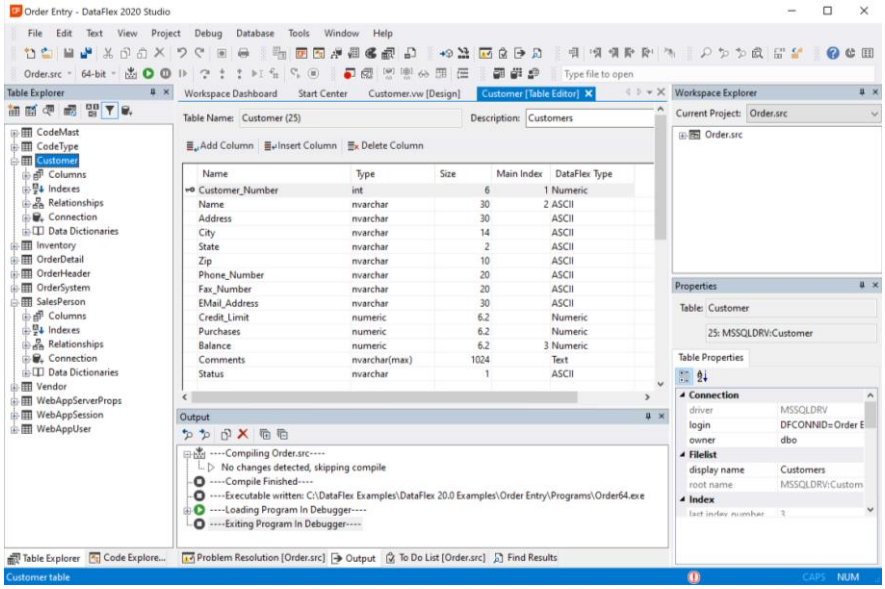
Here are some simple steps you can run through to experience all the new features of DataFlex 2020. They apply to any of the Order Entry - based examples (Web or Windows) but we'll use the standard Order Entry Windows workspace in the following steps:

1. Open the Order Entry workspace in the Studio
    a. You'll notice that the project is set to compile for 64-bit and compiler warnings are enabled so you can see it compiles "cleanly"
    b. You can easily toggle back and forth between 64- and 32-bit and run and debug both from the 64-bit Studio
2. Open the Customer view and translate any of the text to the language of your choice using Google Translate. We'll use Macedonian (it's an up and coming market) and change:
    a. Customer Entry View to Приказ на влез на клиент
    b. Customer Number: to Број на клиенти:
    c. Street Address: to 街道地址： ← yes, this is traditional Chinese, just to show that we can use any language or mix of languages
    d. Make as many changes as you wish and compile and run
3. Note that you can use any capability of the Studio to add Unicode text to your application; the code editor, object properties, wizards, etc.

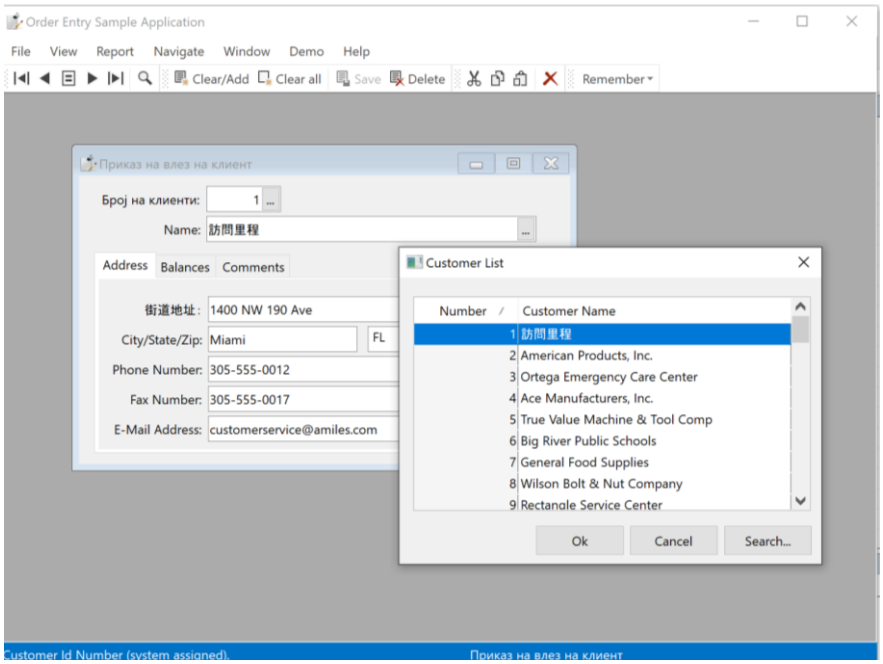
4. Compile and run (either 64- or 32-bit) and you'll see…



5. This application is currently using the embedded database, so what happens if t we try to use Unicode data? Find the "Access Miles" customer and translate the customer name to Chinese 訪問里程 and paste that into the entry form.
    a. At this point before DataFlex 2020, you would immediately see lossy data (???) as the controls converted the data to OEM. All user interface controls in DataFlex 2020 fully support Unicode data.
    b. If we save this record and re-find it, we'll notice that our wonderful Chinese data has turned to ??? because saving to the embedded database is limited to OEM data.
6. Our next step will be to convert our Order Entry database to use Microsoft SQL Server so we can use Unicode data. If you are not familiar with this process see the documentation on Data Connectivity, and in particular, Using Managed Connections and Converting Data.
    a. Create a Managed Connection and an Order Entry database
    b. Use the DataBase Conversion Wizard to convert the table structures and data to SQL Server
    c. Notice that all of the DataFlex ASCII columns have been converted to nvarchar columns; these support Unicode data (where varchar would not).



d. Compile and run and now you can use Unicode data throughout the application…



7. The same basic steps can be used for a WebApp as well – play to your heart's content!

Note that as we prepared for the Technology Preview, we have focused all our efforts on using Microsoft SQL Server as our back-end test environment. Until we have completed the reengineering of the DataFlex SQL drivers we suggest you do the same.

Step 4 – Migrate Your Applications

Of course, the most interesting aspect of experimenting with the Technology Preview is to see what happens with your own applications as you migrate them into DataFlex 2020. There are many layers to this process and those will change depending on the combination of your starting point (the current host revision for any application) and your goal. Some things to consider before you begin this process…

- Make copies of your application and library workspaces for experimenting with migration. This is important because, as mentioned earlier, any source files the Studio touches will be converted to UTF-8 encoding. Also, for complex workspace structures it's generally easier to "migrate in place" with workspace structures that have already been copied to new locations.
- For applications that use SQL Server, we strongly recommend that you make a copy of the database to use with the migrated applications. As you progress past the basic level of migration (see below) you may need to make changes to your database.
- Those of you who have already updated your applications to DataFlex 19.1 and used the compiler warning system to clean up your code have a huge step up in the process of migrating to DataFlex 2020. If you have not gone through this process, we strongly recommend that you review the "Language and Code Cleanup", "How we cleaned up our own code" and "Compiler Warnings" sections of the documentation before proceeding.
- Migrating to DataFlex 2020 can be done to different stages to achieve various goals. The first stage of basic migration involves making only those changes absolutely necessary to compile and run under DataFlex 2020, resulting in an application that is still 32-bit and does not need or use any Unicode capabilities. While the ultimate goal for most developers will be full Unicode applications running in 64-bit, there are many reasons for focusing on basic migration first:
    - You are starting with applications that still use the embedded database. The embedded database has always used OEM encoding and will continue to do so in DataFlex 2020.
    - You are using 3rd party components that do not have 64-bit and/or Unicode capable versions, or you are simply not ready to take on upgrading to their components that are capable.
    - One or more of your applications will not need 64-bit and/or Unicode capabilities, but you still want all the advantages of developing and deploying in the most up-to-date DataFlex environment.
- Once your application can compile and run at this basic stage of migration, you can begin the process of taking it to subsequent stages…
    - Address Unicode-related compiler warnings
    - Convert to full 64-bit
    - Convert data to a Unicode-capable database
        - Use Microsoft SQL Server in conjunction with the Technology Preview

Here are some step-by-step examples of running an application though these stages. We'll use our own DataBase Builder project (from 19.0 so that it hasn't been "cleaned up" yet) in our references. What you see will depend on the applications you choose to experiment with.

1. Make copies of your application and library workspaces for use with DataFlex 2020.
    a. Make sure that your copied structure is complete and any references to libraries are correctly using the copies
    b. If the application is already using an SQL back-end, create a copy of the databases it uses because you will likely be making changes on the backend
    c. Make sure that your managed connection(s) are using the new databases
    d. This entire new structure and data should be compliable and usable in their host revision before continuing

   e. For DataBase Builder there are no libraries and no database, so this was a simple copy of the workspace
2. Open the workspace in the DataFlex 2020 Studio and migrate all libraries and the application workspace "in place" (this is the easiest method for experimentation with the Technology Preview)
   a. The number of changes that are made during migration depends of the host revision.
   b. For applications already hosted in DataFlex 19.1, migration will simply change the revision in the .sws file and add a couple of new lines to the .cfg files of your applications
3. At this point, you'll notice that migrated projects are set to 32-bit and the compiler warning system is suppressed.
4. Press compile!
   a. DataBase Builder reports over 300 errors
5. If your application has not been previously migrated to DataFlex 19.1, some (perhaps most) of the errors you see will be due to the code cleanup done in that revision. Add the use statements for OldDFAllEnt.pkg and/or OldFMACCommands.pkg to check for those type of errors and recompile.
   a. DataBase Builder reports 8 errors
6. We are not going to start the process of updating the application to resolve the errors (or, in subsequent steps, warnings) at this time. The goal is to get a good feel for what the complete process will entail.
7. Use Project Properties to turn on Compiler Warnings by unchecking the "Suppress Compiler Warnings" option and press compile
   a. DataBase Builder reports over 4,000 warnings
8. Don't panic, it'll be fine
9. Since we've brought in the entire set of old classes with OldDFAllEnt.pkg, you will likely find that your application does not actually use them all and many of the warnings are just because the old packages did not go through the code cleanup process. You can copy OldDFAllEnt.pkg from the Pkg folder into your AppSrc folder and start pruning down the classes that get included to get rid of those you know are not used by your application.
   a. For DataBase Builder we commented out the following in our local copy of OldDFAllEnt.pkg

```
//Use WinQl32.pkg
//Use CrystalReport.pkg
//Use dafmac.pkg
```

   b. Database Builder reports just over 3,000 warnings
   c. You can continue to cycle through this culling process – the goal is to eliminate as much code that triggers unnecessary warnings or errors (note that you may find use statements or commands in your code that your application simply doesn't use anymore)
10. While the eventual goal is to clean up all the warnings, what we really want to know is how many of those warnings keep our application from being Unicode capable. We can easily find that out by adding the following to our application (just after the "Use DFAllEnt.pkg" statement for Windows applications or "Use AllWebAppClasses.pkg" statement for WebApps):

```
CompilerLevelWarning General Off
CompilerLevelWarning Unicode On
```

   a. DataBase Builder now reports 440 warnings (see, we told you it was gonna be fine)

11. At this point you can start to get a feel for the types of things you'll need to update for your application to be fully Unicode capable and 64-bit capable (from a DataFlex perspective – interfacing to external APIs and/or 3rd party components are subsequent stages of the migration process). Use the information in the rest of this document to help you understand the other necessary changes.
12. If your application is still using the embedded database, the next step is to convert your data to Microsoft SQL Server. Though your data will be more complex than the examples, the steps are essentially the same as outlined above. If you are not familiar with this process see the documentation on Data Connectivity, and in particular, Using Managed Connections and Converting Data.
    a. Create a Managed Connection and database for your application
    b. Use the DataBase Conversion Wizard to convert the table structures and data to SQL Server
    c. Notice that all of the DataFlex ASCII columns have been converted to nvarchar columns; these support Unicode data (where varchar would not).
13. If your application already uses SQL Server, there are different steps to take:
    a. When you originally converted your data, did you select ANSI or OEM conversion? If OEM, you'll need to update the tables to use ANSI data; see the Database section for details.
    b. Check the SQL data types used for the ASCII columns in your tables. Anywhere you want to use Unicode data you need to change the data type for that column to nchar or nvarchar. You can use any method you are comfortable with to make these changes. Since the data was previously limited to ASCII data, changing the data types to the Unicode capable data types will not result in lossy conversions of existing data.
    c. Don't feel like you need to convert all your columns at once, you can simply change a few to see how entering and sorting Unicode data feels

Obviously, the steps outlined are a very simplified view of application migration, and you'll still have to deal with all your external interfaces – but from a high-level perspective it's relatively straightforward:

- Determine which stage of migration you require for any particular project
- Understand the types of changes that may be necessary (reading this document completely will get you there)
- Use the Studio error and warning system to find as much of the code subject to change as possible
- Examine your use of external components and APIs and adjust them as required

Once you get into the process you'll find that its all about repetition; yes, there may be hundreds, or even thousands of changes necessary – but it is really only a small handful of different types of changes that you repeat a number of times. The learning curve is all in how to handle each different type of change.

Step 5 – Use the Forum

We have set up a special forum for discussions about the Technology Preview and that is where you will find all the most up-to-date information. As you explore and experiment with DataFlex 2020, be sure to check the forum on a regular basis.

https://support.dataaccess.com/Forums/forumdisplay.php?88-DataFlex-2020-Technology-Preview

There will be many developers working with the Technology Preview and getting involved in the forum discussions could save you hours or days of time. Conversely, sharing your own experiences could save other developers hours or days as well.

Step 6 – No Excuses

Come on, admit it, you didn't really read all the documentation now did you? Trust us, it won't take very long and you can relax with a refreshing beverage while doing so. Give it a shot; it certainly won't hurt...

What Comes Next

The development process for DataFlex 2020 will continue and the customary Alpha and Beta cycles will take over as we approach the new year. In addition to the usual focus on documentation, adjustments from your feedback and fixes, there are a few main areas of development still under way:

- SQL drivers are being reengineered to use UTF-16
- The Studio will take on Microsoft SQL Server as its default use profile
- Examples will be moved to Microsoft SQL Server
- Updated registration system (including replacing the old technology used for evaluation licenses)

Unicode in DataFlex

DataFlex 2020 is fully Unicode. The language itself (compiler & runtime) work with UTF-8 as their default encoding. In practice this means that source code is stored as UTF-8 so that string literals and comments can contain any Unicode character. String variables store their data in memory as UTF-8 so that they can contain any Unicode character. When communicating with external API's, conversions to UTF-16 will take place. To make this easier from within the DataFlex language there is a new WString type. Strings are automatically converted to UTF-16 when using this type.

String

While string variables in practice can contain any binary data, the runtime always treated them as OEM strings. Strings are now treated as UTF-8, which means that each string function and command assumes the data to be UTF-8 encoded. Since UTF-8 is a variable length encoding a character can be more than a single byte long. The string functions are adjusted for this and functions like Mid & Left assume their parameters to be character positions. The Length function also returns the number of characters in a string (which can be different than the number of bytes). A new function named SizeOfString returns the number of bytes used by a String.

It is still possible to convert a string to OEM or ANSI in memory. But instead of ToOEM and ToANSI this is done using Utf8ToOEM and Utf8ToANSI.

Collating

String comparisons with Unicode are much more complicated than with OEM / ANSI. DataFlex 2020 uses the ICU Library for comparing strings according to the Unicode standards. Multiple collations are supported and can be configured via the new DF_LOCALE_CODE string attribute. This global attribute defaults to the language of the operating system. It can be changed at runtime and it is configured using ISO639 language codes. See http://www.localeplanet.com/icu/iso639.html for available codes. Note that when using the embedded database, the indexes will be built up according to the old collating system configured via DF_Collate.cfg for backwards compatibility.

String Functions

SizeOfString

This function returns the size of a string in bytes (UTF-8 code-points). This can be different than the length returned by the Length function. which returns the number of characters in a string.

PointerToString

This function converts a pointer to a string in memory into a string. This can be used within an expression and the resulting string can then be moved into a string variable. This function replaces the special functionality that the address (now pointer) type had when moving an address to a string. This conversion is now illegal and this function can be used to perform the same operation.

WString

A lot of external API's, such as the Windows APIs, work with UTF-16 encoding. When calling these API's, the strings that need to be converted as DataFlex String variables are UTF-8 encoded. To make this easier a new WString type was added to the language. When moving strings to / from this type, the data is automatically converted to UTF-16. WStrings can be passed to external functions as parameters or as pointer (in case of a return buffer).

It is recommended to only use this WString type when actually calling an external API, and not instead of the regular String type. Even though string manipulations and string functions do work, internally the data is converted between UTF-16 and UTF-8 for each operation, which will slow down your application.

When working with COM there is no need to use the WString, as variants are already UTF-16 encoded (they always have been).

Example: WString Parameters

When an external function has a string as a parameter (technically this is usually a pointer to a string) like this:

```
External_Function WritePrivateProfileString "WritePrivateProfileStringA" Kernel32.dll ;
    String sSection String sKeyName String sValue String sFileName Returns Integer
```

Then converting it, to its wide version is now as easy as changing it into:

```
External_Function WritePrivateProfileString "WritePrivateProfileStringW" Kernel32.dll ;
    WString sSection WString sKeyName WString sValue WString sFileName Returns Integer
```

When calling this function, you can simply use strings as parameters. These can come from a parameter, an expression or a constant without problems. The runtime will automatically convert them to a WString before actually calling the external function. As with string, the runtime is smart

enough to pass a pointer to the wide string when executing the external function. So, the line below will work properly:

```
Move (WritePrivateProfileString(sSection, "", "", psFilename(Self))) to iRes
```

Example: WString with Pointer Parameters

It is common practice to define external API's with Pointer (formerly also called Address) parameters. This is done when needed to allow passing 0 (NULL) as parameter or when a string is returned. This can simply be done the same way as we used to do with String parameters.

```
External_Function GetModuleFileNameW "GetModuleFileNameW" Kernel32.dll ;
    Handle   hModule ;
    Pointer  lpFilename ;
    UInteger nSize ;
    Returns  UInteger
```

This function returns a string in the buffer that is passed. The size of the buffer is passed as separate parameter. Calling this function can be done like this:

```
Integer iNumChars
WString wApplicationFileName
String sApplicationFileName

Move (Repeat(Character(0), 1024)) to wApplicationFileName
Move (GetModuleFileNameW(0, AddressOf(wApplicationFileName), 1024)) to iNumChars
Move (CString(wApplicationFileName)) to sApplicationFileName
```

So, we define a WString and fill it with 1024 null characters. Do note that Repeat generates a UTF-8 string, which is then converted to UTF-16 when it is put into the WString buffer. Then we call the external function, passing a pointer to the WString. The external function changes the WString buffer. On the last line we convert the UTF-16 result string to a regular UTF-8 string.

The CString function is used to adjust the length of the string. DataFlex strings (both WString and String) can contain 0 characters, while in other environments the 0 usually terminates the string. To support this, DataFlex strings actually store a length with them. The external function will adjust the content of the string, and write a 0 terminator, but it will not change the length of the string (which remains 1024 characters). Calling the CString function fixes that.

WString Functions

A couple of WString specific functions have been added to make working with WString easier:

SizeOfWString

This returns the number of WChars (codeunits / double-bytes) of a WString.

PointerToWString

This takes a pointer to a WString (or Char array with two bytes per character) as a parameter and returns a WString.

External_Function Wrapper Functions

To properly support Unicode DataFlex uses the Wide versions of Windows API functions that use strings. A lot of these Windows API's are called using External_Function within the DataFlex packages. We have updated our packages while maintaining as much backwards compatibility as possible. In most cases where we had to make changes that requires changes in the calling code, we provide wrapper function that perform the necessary conversions.

In most cases these wrapper functions are slower, so our code calling these functions usually doesn't use the wrapper function but directly calls the wide version (functions ending with a W). It is recommended, but not mandatory, that developers also convert their interfaces to use the wide versions.

Direct_Output / Append_Output / Direct_Input

The file paths passed to these commands are now assumed to be regular UTF-8 strings and they can contain Unicode characters. Using the Read, ReadLn, Write and WriteLn commands does not perform any conversions on the data so strings will be written / interpreted as UTF-8 data. When working with OEM or ANSI files, the conversion functions (Utf8ToAnsi, AnsiToUtf8, OemToUtf8 and Utf8toOem) can be used to properly convert the data. Note that text files written with previous versions of DataFlex will usually contain OEM encoded strings unless conversions were made in the source code.

Database

The embedded database does not support Unicode and data written to it is converted to OEM by the runtime. It is backwards compatible and the database can be shared with older revisions of DataFlex. The sorting of the indexes is done according to the Df_collate.cfg in bin or bin64. Note that string comparisons in the language are now performed using the new Unicode comparisons and can be different than the embedded database collation.

It is recommended to use SQL databases where MS SQL is the recommended backend. For this Technology Preview it is the only backend that has been tested. To work with Unicode on MS SQL, use the NChar and NvarChar data types. Data is stored as UTF-16 and the SQL drivers will perform the necessary conversions for you.

Note that if the df_table_character_format attribute in your existing SQL tables is set to OEM, your data will be stored as OEM in the database. When converting fields to NVarChar or NChar the automatic conversion of your data by MS SQL will likely fail as it interprets your data as ANSI. So, it is recommended to convert your existing SQL data from OEM to ANSI before converting to Unicode data types.

Source Code

DataFlex 2020 uses UTF-8 encoding for all source code files. When you create new files in the Studio, they are automatically created as UTF-8 and use Byte Order Marks (or BOMs) to signify their encoding style. Source files from previous revisions of DataFlex used OEM encoding and those files do not need to be converted to UTF-8 to compile. Editing an older, OEM encoded, source file in the 2020 Studio will automatically convert it to the new UTF-8 encoding.

This change in the encoding of source files is one of the main reasons we recommend creating separate copies of your application and library workspaces before using DataFlex 2020.

Strings for Binary Data

In the past, DataFlex strings were sometimes used to store binary data. We recommend not to use that technique and use UChar arrays instead. The string functions and the debugger will now try to interpret the strings as UTF-8 data and binary data in a lot of cases is not valid UTF-8. String functions do not translate their parameters directly to memory offsets any more but interpret them as character offsets where they will analyze the string to convert them to memory positions. This will go wrong

when the data are not valid UTF-8 strings. An added advantage of using UChar arrays is that the Max_Argument_Size does not apply to them.

Replace TYPE Definitions with Structs

The TYPE command should not be used anymore. This is an obsolete way of defining structs, and it is no longer recommended. Instead, use the Struct command for defining structs. The 2020 Studio generates compiler warnings when it encounters Type commands.

This also means that the commands ZeroType, FillType, GetBuff, GetBuff_String, Put, Put_String, ArrayPut, Size_of_field, StoreField and RetrieveField should not be used. Also, the automatically created [TypeName]_Size constant cannot be used; use SizeOfType() instead. Again, the 2020 Studio generates compiler warnings when these commands are encountered.

Unicode 101

Understanding Unicode in DataFlex starts with a basic understanding of character encoding and how all the different components work together. If you haven't already seen it, we recommend you watch Harm Wibier's DataFlex NextGen presentation from Synergy in The DataFlex Learning Center. It covers all the aspects of the history of character encoding and how DataFlex worked in the past and in DataFlex 2020 and beyond.

Non-Unicode programs can only handle a single language and have a small set (255) of different characters that can be used. The system Codepage determines which language is being used and conversion between codepages is lossy. ANSI & OEM are encodings using this principle.

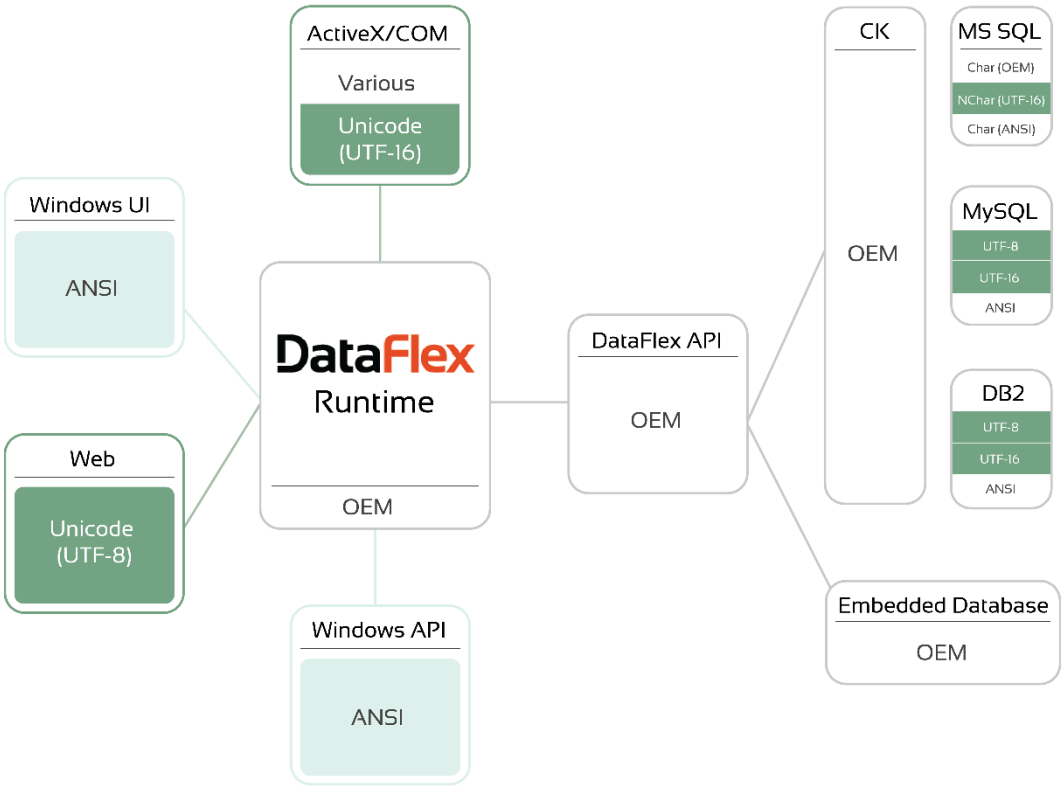
Unicode programs can mix languages because 1,114,112 different characters can be used. UTF-8, UTF-16, UTF-32 and UCS-2 are Unicode encodings. In addition to the character encodings themselves, it's important to understand which encodings the various components of the system are using.

Windows started with ANSI (8-bit) and then moved to UCS-2 (16-bit) using WideChar API's (double byte) and continued to support the ANSI API's. When UCS-2 didn't work out as expected, Windows moved to UTF-16 (16-bit or more) and changed their double byte API's (again, still supporting ANSI API's).

The other components (the Web, ActiveX components, databases) use various character encodings (mainly UTF-16 or UFT-8) and sometimes a mix of encodings, like databases based on the data types or generation of the products.

Character Translations in DataFlex 19.1 (and earlier)

DataFlex needs to communicate with all these components, regardless of the character encoding involved, so we use character translations between encodings as data moves throughout the system. Up until now, the core of DataFlex has used OEM character encoding so the system is constantly translating between OEM and the character encoding of the component being addressed and back. We can see how DataFlex interacts with the rest of the environment in the following diagram:



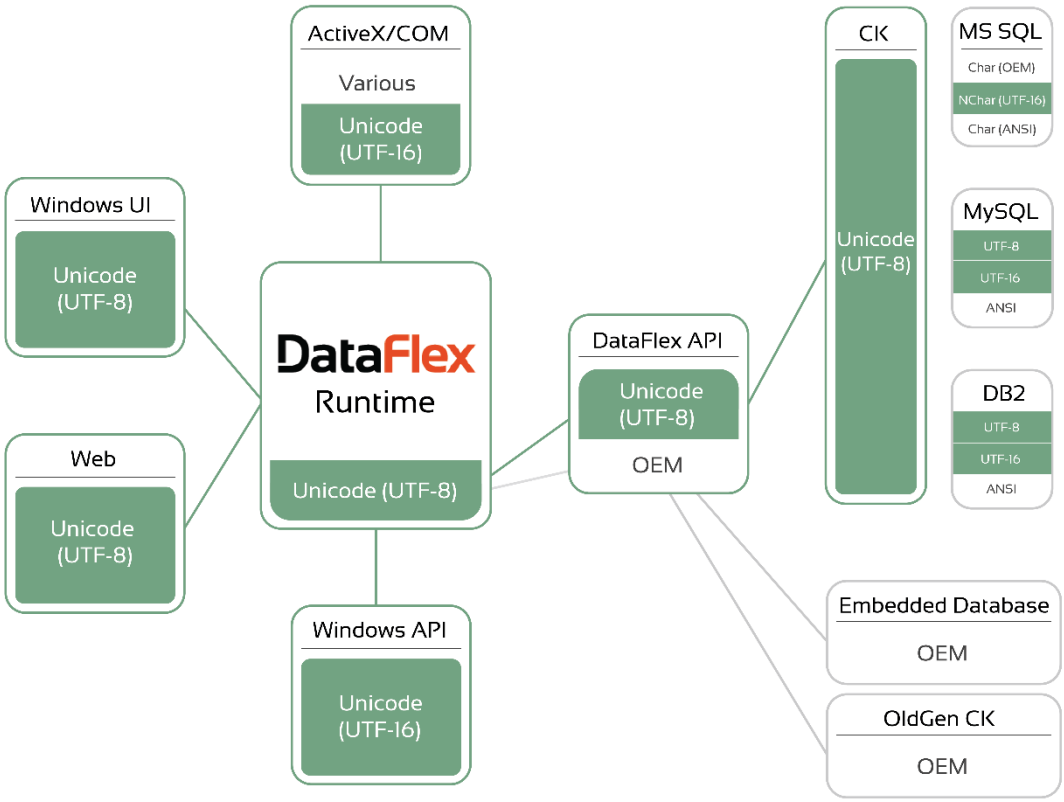
Character Translations in DataFlex 2020

DataFlex 2020 represents a fundamental shift in the core character encoding from OEM to UTF-8 and subsequently the various translations between DataFlex and the other components of the system.

We chose UTF-8 as our new core encoding because it provides the best backwards compatibility, is native to the Web and is best for Western languages. The source code is stored as UTF-8 with a byte order mark (or BOM). Non-ASCII characters allowed in string literals and comments and any source file that does not contain a BOM will be interpreted as OEM.

DataFlex 2020 uses the wide Windows API's and supports this though a new WString type for automatic conversions between UTF-8 and UTF-16.

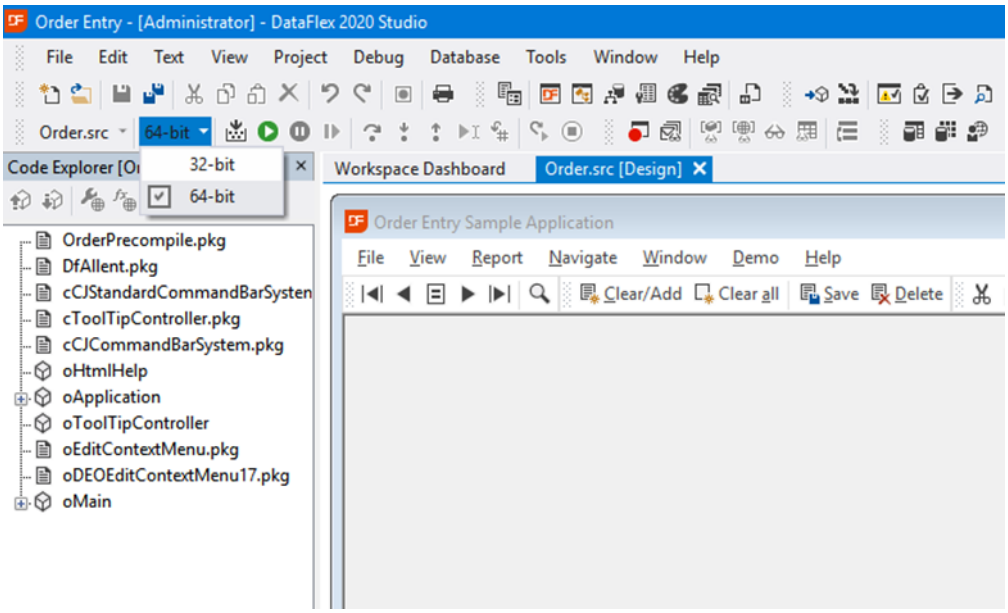
We can see how DataFlex 2020 interacts with the rest of the environment in the following diagram:


ActiveX/COM
Various
Unicode (UTF-16)
Windows UI
Unicode (UTF-8)
Web
Unicode (UTF-8)
DataFlex
Runtime
Unicode (UTF-8)
Windows API
Unicode (UTF-16)
DataFlex API
Unicode (UTF-8)
OEM
CK
Unicode (UTF-8)
MS SQL
Char (OEM)
NChar (UTF-16)
Char (ANSI)
MySQL
UTF-8
UTF-16
ANSI
DB2
UTF-8
UTF-16
ANSI
Embedded Database
OEM
OldGen CK
OEM

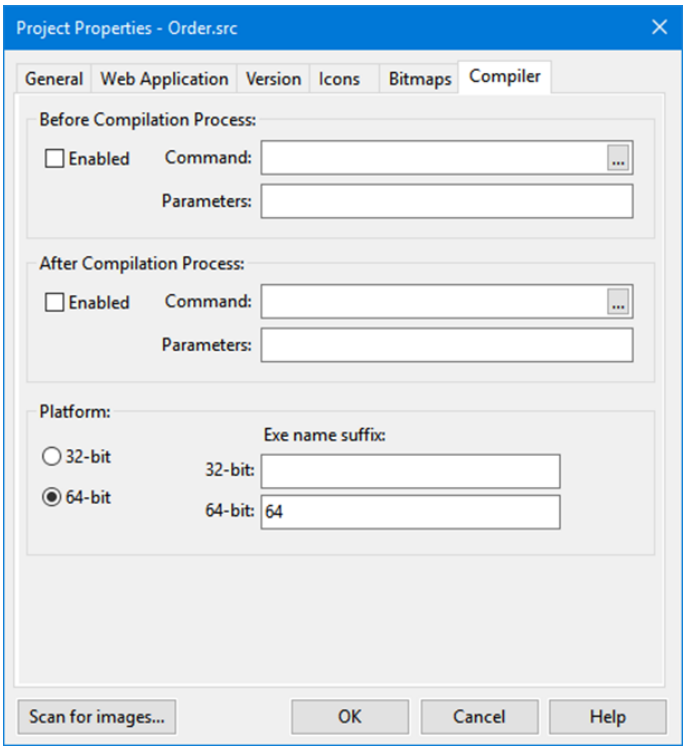
64-bit in DataFlex

With DataFlex 2020, programs can be compiled and run in both 32- and 64-bit. The Studio is 64-bit only, but you can choose to either compile programs to be 32- or 64-bit. So, a single codebase can serve to generate both a 32-bit and 64-bit application. Also, you can run and debug both a 32-bit and 64-bit program from the same 64-bit Studio.

There is a new dropdown selector available to quickly switch between 32- and 64-bit.



This setting can be configured per project and it can also be set through the Project Properties window, on the Compiler tab page.



Both the 32- and 64-bit compiled programs end up in the programs folder. They are differentiated by appending a suffix to the executable name. In the screenshot above, the 32-bit version will be named Order.exe, and the 64-bit version will be named Order64.exe. If you do not differentiate the executables with the appropriate use of suffixes, they would get the same name. Any compilation would overwrite the one that is already there. This is, in fact, the desired situation for a webapp.

The compilation platform and executable name suffix are stored in the project cfg file:

```
Platform=x64     (or  Platform=x86)
64BitSuffix=64
32BitSuffix=32
```

For a simple, well-written program, switching to 64-bit and hitting compile may be all that is needed (our examples illustrate this well). However, for more advanced applications, significant changes may be needed. The following sections contain detailed documentation about all the possible changes (though not all may apply to your specific applications).

Data Types

The main difference between 32-bit and 64-bit DataFlex is the fact that the pointer size is increased from 32 to 64 bits. The same is true for Handle. It is important to realize that no pointer in 64-bit mode can be moved to the integer data type because of pointer truncation. Pointer truncation means that a 64-bit value, which exceeds 2^{32} is transferred to a 32-bit data type (or smaller), leading to removal of the higher 32 bits, and thus to an incorrect value. Referring to the truncated pointer address will most often be illegal and also lead to a crash.

Handles (Windows data type) are treated differently: although its size is 64-bits in 64-bit mode, its upper half bits will be empty, which means that moving it to the integer type will work and will not truncate the value (with an exception, which is HTreeItem, which is actually a pointer). However, it is not advised to move a Handle to an Integer, but to keep it a Handle at all times.

The integer in 64-bit DataFlex stays 32-bit, which is in line with other Windows environments. Also, for technical reasons, we have added an integer-like data type that is equally sized to the pointer, called Longptr. This data type is only needed in advanced cases.

Longptr Data Type

DataFlex 2020 introduces the new data type Longptr, which is also available when compiling 32-bit. It is a memsize type: it is a 32-bit size integer in 32-bit compilation and a 64-bit size integer in 64-bit compilation. This way, it can always hold a pointer value without being truncated and without needing to use a compiler switch. The single-character identifier for this type is "P", while Integer is "I" and Timespan has changed to "?". This makes the following a statement to set a constant to a value of type Longptr:

```
Define SOME_LARGE_VALUE   for |CP$03762874671
```

Address and Pointer

In previous versions, Pointer was in fact a replacement for Integer. In DataFlex 2020 Pointer is a replacement for Address, which is the native pointer type. Either Pointer or Address can be used.

Alias Data Types

This table shows the aliasing of a number of data types. For example, internally, OLE_Handle is not a data type by itself, but it is an alias for a different data type, Integer in this case. To be clear: Longptr is a data type on its own, it is not an alias.

Alias data type	*Alias for* in 19.1 (and earlier)	*NEW Alias for* in 32-bit	*NEW Alias for* in 64-bit
OLE_Handle	Handle	Integer	Integer
Handle	Integer	Longptr	Longptr
Pointer	Integer	Pointer	Pointer
DWord	Integer	UInteger	UInteger
ULongptr	-	Uinteger	UBigInt

The take-away message here is to always implement the data type that it really is: use handle when it is a handle, use pointer (or address) when it is a pointer, use integer when it is an integer that will never exceed 2^{32}, and use Longptr when it is an integer type that may hold a pointer value.

The Longptr and ULongptr types were already available in DataFlex 19.1 as a preparation step towards DataFlex 2020. However, note that in 19.1 Longptr is not a data type on its own, but an alias for Integer. It allowed users to start preparing their code for 64-bit.

DWord

For many years, DWord has been an alias for Integer. In DataFlex 2020 this is changed to an alias for UInteger. This is not as straightforward and simple as it may seem. A realistic consequence would be that assignments of negative values (such as -1) to a DWord, which had always been possible, would now lead to an Out of Range runtime error. We have solved this issue by using value wrapping of Integer and UInteger (DWord), similar to how it works in C/C++.

For example, the binary value of -1 is 0xFFFFFFFF, and when this is assigned to a DWord, it will get the unsigned value of this binary value (which is 4,294,967,295). It also works the other way around: A UInteger with a value larger than 2^{32} (e.g. 4,294,963,020) will, when moved to an Integer, be wrapped to a negative value (-4276).

This way, we expect that (almost) all usage of DWord will still just work well, although one may want to check their code on correct usage of DWord.

A related small change is that logical evaluations of UInteger types is made possible. For example:

```
UInteger uiTest
If (uiTest iand 15) Begin
```

In previous versions this could not be compiled. Now it will just work.

32-bit DataFlex – Possible Code Changes

Two 64-bit related issues may be relevant to your applications when compiling 32-bit.

DWord

We have changed DWord to be an alias for UInteger instead of Integer. In theory this might have an influence on your code. For example, you could have a test on a DWord variable being less than zero. This would now never be true.

Handle arrays

The Handle data type is an alias, but will no longer be an alias for Integer, but for Longptr. As a consequence, the following code will raise a runtime error in DataFlex 2020, while it was fine before:

```
Property Handle[] phStaticViews
Integer[] iStaticViews
Get phStaticViews to iStaticViews
```

The reason for the error is that an array of handles (Longptr) is moved to an array of a different data type (Integer). In order to correct this, `iStaticViews` has to be changed to Handle[].

64-bit DataFlex – Possible Code Changes

Simple applications may not need any changes when compiling 64-bit. However, this depends a lot on the complexity, the use of third-party DLL's and correct data type coding. Below is a list of changes that you may need to make for your application to work in 64-bit.

Compiler Switch

In some cases, changes must only be active for the 64-bit environment, not 32-bit, for which you can use the new compiler switch **`IS$WIN64`**.

Example:

```
#IFDEF IS$WIN64
    #Replace LONGPTR_DTSIZE 8
#ELSE
    #Replace LONGPTR_DTSIZE 4
#ENDIF
```

Illegal Data Type Conversions

In 64-bit mode (not 32-bit), some conversions are now illegal, due to the risk of data loss (pointer truncation). For this reason, in 64-bit, conversions from Pointer or Address to Integer are not allowed. They will lead to runtime illegal conversion errors, whether or not overflow would happen. So, whenever such conversions are in your code, they will become problematic in 64-bit. The reverse, conversion from Integer to Pointer/Address is not illegal, but in 64-bit they make no sense in most cases. Conversions from Longptr to Integer or Pointer or Address and vice versa are allowed and most often do make sense.

Moving a pointer value to an integer type could be considered sloppy coding and has never been an advisable thing to do. When porting to 64-bit, it's time to correct that. The advised way to change this is to always use the Pointer data type, i.e. in local data types and function/procedure parameters. Alternatively, you could also use the new Longptr data type.

The biggest issue here is finding the illegal conversions (Pointer → Integer) since many of them only show up at runtime. A simple move of a pointer to an integer will be detected by the compiler, though, and reported as compile error. At this moment, a global search for the keywords Pointer and Address and the function AddressOf is likely to be a good start to find more possible problems.

Correct Data Type Usage

In general, it is advised to use the right data type for each variable and parameter in order to prevent potential problems. This is even more important in 64-bit. As mentioned above, pointer must be used correctly. Also, it is advised to use OLE_Handle for OLE Handles and Handle for all other handles, even though nothing will go wrong when mixing them up or using Integer or DWord (except when using arrays). One exception is the HTreeItem data type. While this looks like a handle, it is actually (in Windows) a pointer to a struct. The correct data type here is either Pointer, Longptr or Handle (not Integer or DWord).

External Functions

When implementing data types for input and output parameters of external functions, the data type used must match that of the Windows function. While implementations work in 32-bit, they might break in 64-bit. This table may be helpful in getting it right:

Windows data type	Advised DataFlex type in external function	Allowed alternatives
Handle, hWnd, hTreeItem, HItemList, HInstance, hIcon, HGlobal, HDC, etc.	Handle	Longptr / Pointer

Pointer (such as: `VOID *lpx`), LPCSTR, LPCTSTR, LPVOID, PUINT, LPDWORD (almost everything that starts with LP)	Pointer	Longptr
OLE_Handle	OLE_Handle	Integer / Uinteger / Dword
lParam, wParam, lResult	Longptr	Pointer
DWord	DWord	Integer / UInteger / OLE_Handle
Size_t, UINT_PTR, DWORD_PTR	ULongptr	Longptr / Pointer
INT_PTR, LONG_PTR	Longptr	Pointer
INT, INT32, UINT, UINT32, LONG, ULONG, etc.	Integer / UInteger	Dword / OLE_Handle
SHORT, INT16, UINT16, WORD	Short / UShort	
BYTE	UChar	
BOOL	Integer	DWord / OLE_Handle
DWORDLONG	UBigInt	

In particular, look out for parameters named LParam and WParam that are typed integer. Those will most often have to be corrected to Longptr.

Users may face a very peculiar and rare problem when using the POINT structure *by value* in an external function. Since DataFlex cannot pass structs by value, one should pass them using pointer. However, there are Windows functions that only accept them *by value*. For a solution to this, take a look at the system package Winuser.pkg.

Structs

Similar to external functions definitions, data types in Windows structs must be typed correctly when they are exposed to the outside world. In addition, there is an issue called *structure alignment* (or structure padding), when structs are passed to other DLLs or Windows functions. This is because the (C) compiler ensures that each struct instance will have the alignment of its widest scalar member (for performance reasons) and extra memory space may be inserted within the struct (unless structure alignment is explicitly switched off in the DLL). Please look at the documentation below for more information.

DataFlex does not do structure alignment. This means that you might have to add extra padding items yourself to exposed Windows structs. This issue is especially relevant to the 64-bit platform. Take this example:

```
Struct tWinChooseFont
  DWord lStructSize
  Handle hwndOwner
```

```
End_Struct
```

In 32-bit, both DWord and Handle are 32-bit items (4 bytes), which does not lead to any padding. However, in 64-bit, Handle has become 64-bit (8 bytes) and that causes the struct to have 8-byte alignment, which means that in Windows compilers there will be 4 bytes of space inserted after lStructSize. If this doesn't get corrected in 64-bit environments, there can be an unexpected runtime error or crash upon calling the external function. The solution in DataFlex code is:

```
Struct tWinChooseFont
  DWord lStructSize
#IFDEF IS$WIN64
  Integer iStructAlignment
#ENDIF
  Handle hwndOwner
End_Struct
```

Be aware that such changes might influence code where you do a SizeOfType() on that struct.

Note: The structure alignment issue is not relevant to structs in COM class interfaces. In that case, structs will be exposed correctly.

More information:
https://msdn.microsoft.com/en-us/library/ms253935.aspx
http://www.catb.org/esr/structure-packing/#_structure_alignment_and_padding

Third Party Binaries

This is probably often the biggest hurdle for making applications ready for 64-bit. Any 32-bit third party dependency must be replaced by a 64-bit version, since a mix of 32- and 64-bit binaries is not possible. This means that you may have to request a 64-bit version from the vendor or, when you are in possession of the code yourself, recompile it in 64 bits (and solve the possible issues that come with it). Theoretically, when neither option is possible, you may have to find another solution, such as removing the 3rd party library from the application or replacing it with an alternative component.

DataFlex applications can be compiled either 32- or 64-bit (using the same codebase). To use the right version, you can use a compiler switch to use either the 32- or 64-bit component:

```
#IFDEF Is$Win64
    External_Function FuncName "FuncName" xxx64.dll Integer iLength Returns Handle
#ELSE
    External_Function FuncName "FuncName" xxx32.dll Integer iLength Returns Handle
#ENDIF
```

COM Classes (Generated by the COM Class Generator)

The COM class generator can take a DLL or OCX as input for generating DataFlex wrapper classes. It can do that for both 32- and 64-bit binaries, which may result in either equal or dissimilar generated pkg-files depending on the contents. The CLSIDs may be equal, but that must be validated (Windows knows whether to use the 32- or 64-bit DLL when CLSIDs are equal). The classes in the generated PKG file will in many cases be identical, because the COM Class Generator will use Longptr (or ULongptr) for pointer-sized C-types like INT_PTR. This is the full list of supported platform dependent C data types that convert to either Longptr or ULongptr: `INT_PTR, LONG_PTR, LPARAM, HMODULE, ULONG_PTR, UINT_PTR, WPARAM.`

However, when the COM object signature of 32- and 64-bit is different, for example with C data type `LONG` in 32-bit and `__int64` in 64-bit (by using compiler switches), the generated classes will be different. Obviously, the COM class generator has no way of knowing about a platform-dependent type here. When there are only a few differences, one might decide to use just one class PKG file and edit it manually to use `IS$WIN64` switches. You can also use the define `OLE_VT_INT_PTR` , which is a replacement for `OLE_VT_I4` in 32-bit and `OLE_VT_I8` in 64-bit. Alternatively, when there are many changes or when the CLSIDs are different, it might be a better choice to have two files and use an `IS$WIN64` switch for the `USE` statement for that file.

GetWindowLong and Others

Although the Windows functions GetWindowLong, SetWindowLong, GetClassLong and SetClassLong are still available on the 64-bit platform, they will lead to errors when used with pointer values, because of pointer truncation. Therefore, it is strongly advised to replace those functions by the respective xxxPtr functions, such as GetWindowLong**Ptr**, which will work on both 32- and 64-bit platforms. So, it is not needed to use a compiler switch to use either of the two. All that is needed is to add “Ptr” to the name of the function call to make the code work well in both 32- and 64-bit.