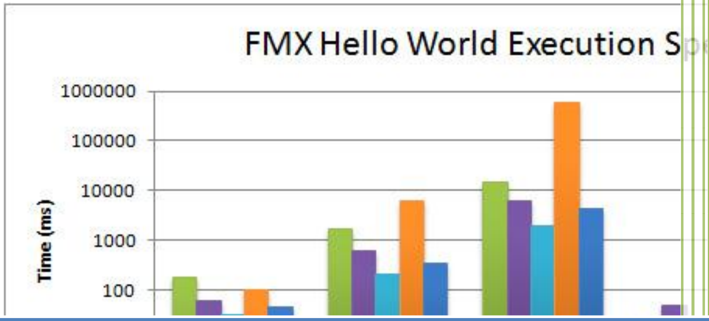


2014

Performance Comparison from Delphi 2010 to XE6

XML Parse: 1325 ms
Paint: 212 ms
Paint: 93 ms
Paint: 96 ms

Thomas G. Grubb

RiverSoftAVG.com

6/14/2014

This document is based on a series of blog posts at Delphi CodeSmith blog: (http://blogs.RiverSoftAVG.com):

http://blogs.riversoftavg.com/index.php/2014/05/12/performance-comparison-from-delphi-2010-to-delphi-xe6-introduction/

It has been lightly edited for clarity and continuity.

Contents

Introduction

With the release of Delphi XE6 and Embarcadero's emphasis on Quality, Performance, and Stability (QPS), I wanted to see for myself the level of improvement, especially in performance. Delphi XE6 is **definitely faster and more responsive** than the last few versions, especially in FMX, but I wanted to see if I could quantify the performance improvement. There have been a couple recent articles and posts about XE6's speed (see http://www.dewresearch.com/news/232-rad-studio-xe6-lo-and-behold- and http://www.delphitools.info/2014/05/07/a-look-at-improved-inlining-in-delphi-xe6/). This document records my explorations into the performance differences between Delphi 2010, XE, XE2, XE3, X4, XE5, and XE6. Before starting, I made some predictions about what I would see when comparing Delphi 2010-XE6.

- **EXE size will probably increase with every version of Delphi.** This was based on the fact that every version of Delphi has been adding to the runtime library (RTL). After using Borland/CodeGear/Embarcadero products for almost 2 decades, I would be surprised if the linkers have improved much to remove unused code.
- **FMX executables will be larger than VCL executables.** This is completely expected as FMX controls are non-native controls (i.e., they don't use OS level equivalents) so all the drawing and interaction code must be compiled into the executable.
- **FMX executables will be slower than VCL executables, though each new version of Delphi for a platform should improve.** See second bullet above. However, I expect Embarcadero has been working hard on improving FMX execution so I would expect every version to be slightly faster than the previous.
- **Win32 and Win64 compilation should be faster than other platforms.** Embarcadero has a long history with developing for the Windows platform so these versions should be far superior to other platforms. Also, with the use of LLVM compiler and linker, I *expect the compilation to be MUCH slower as now Object Pascal becomes a **at least** 2-pass compiler: compilation of object pascal into LLVM bytecode and then the compilation and linking of that code into the final platform executation.* As LLVM bytecode from what I understand is language independent, I would expect it does 2 pass compilation like C.
- **Windows FMX executables will be faster than other platforms' FMX executables.** I have much less confidence in this prediction as the LLVM compilers for each platform are not under Embarcadero's control so it is possible that a platform vendor will optimize LLVM code much better than Embarcadero. *However, I base this prediction on the fact that it is far easier to debug and optimize on Delphi's native platform than other platforms so I expect that Windows will be where Embarcadero has put in most of their optimization efforts.*

Methodology

For the tests in this document, I created sample applications and compiled them in Release configuration for each version of Delphi 2010 to XE6 (if applicable). All compilation tests (including for OSX, iOS, and Android targets) were performed on my Windows 7 box, and, if possible, compiled from the command line to avoid Delphi IDE overhead (this was not possible with some of the mobile tests). Windows testing was performed on this box (Microsoft Windows 7 64-bit, Intel I7 930 @ 2.8 GHz CPU, and 6 GB RAM) with all applications except Microsoft Excel (to record test results) closed. iOS applications were tested on an iPod Touch deployed directly to the device in debug mode. Android app were tested on a Nexus 7 (2013) deployed directly to the device in release mode. iOS applications by necessity were in Debug configuration in order to be able to compile and deploy them to my iOS device. *Every test was performed a minimum of 3 times. The best 3 times were averaged to produce a final value.*

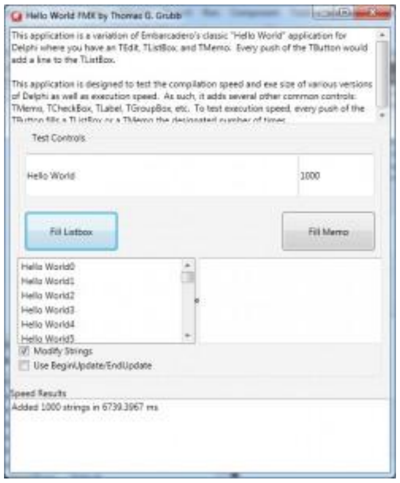
Tests

There are 3 major application types used for the performance tests. Each major application type had at least 3 applications written to explore performance including VCL, FMX, and FMX Mobile.

Hello World Tests

About the Hello World Test Applications

My first test was to create a slightly complex Hello World Delphi application for VCL, FMX, and Mobile (iOS and Android). This application is a variant of the classic Delphi application of a TEdit, TListBox, and TButton. Instead of one add of the TEdit constant every click of the button, each click would fill a TListBox or a TMemo with some number of repetitions. The TListBox.Items.BeginUpdate/EndUpdate and TMemo.Lines.BeginUpdate/EndUpdate could be turned on or off. In addition, a number could be optionally be appended to each TEdit string (to avoid any efficiencies with using the same string each addition). The Hello World source code for VCL, FMX, and Mobile can be downloaded http://riversoftavg.com/blogs/wp-content/uploads/2014/05/HelloWorldPerformance.zip.



Hello World FMX by Thomas G. Grubb

This application is a variation of Embarcadero's classic "Hello World" application for Delphi where you have an TEdit, TListBox, and TMemo. Every push of the TButton would add a line to the TListBox.

This application is designed to test the compilation speed and exe size of various versions of Delphi as well as execution speed. As such, it adds several other common controls: TMemo, TCheckBox, TLabel, TGroupBox, etc. To test execution speed, every push of the TButton fills a TListBox or a TMemo the designated number of times.

Test Controls

Hello World | 1000

Hello World0
Hello World1
Hello World2
Hello World3
Hello World4
Hello World5

[✓] Modify Strings
[ ] Use BeginUpdate/EndUpdate

Speed Results

Added 1000 strings in 6739.3967 ms

Figure 1 Hello World FMX Application Screenshot

Hello World Compilation Tests

For our first tests, I wanted to see how much Embarcadero has been improving the various compilers (Win32, Win64, OSX, iOS, and Android) for each version of Delphi, both in terms of compilation speed and final executable size. I know that this is a trivial part of the performance comparison but I thought it is important to look at every aspect of the application making process. I took the Hello World project described in the introduction and ran it through the various Delphi compilers: 2010 to XE6, Win32, Win64, OSX, iOS, and Android.

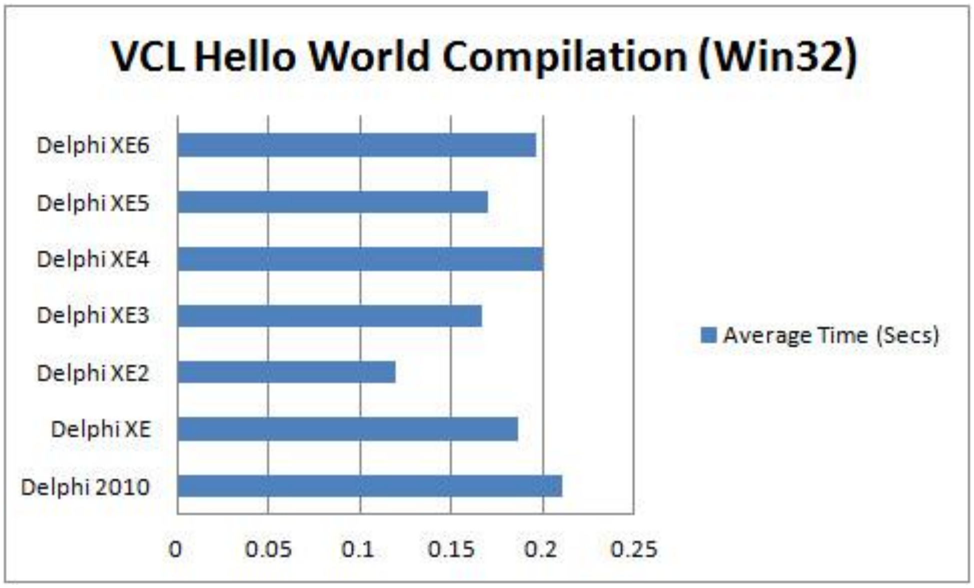


Figure 2 Comparison of compilation times for VCL Hello World Application (Win32) with Delphi 2010 to XE6

Hello World, VCL, Win32

For the Win32 test, all compilers from Delphi 2010-XE6 were used. Every compiler was blazingly fast and with such a small application (101 lines of code), **compilation was performed so fast (less than 1/4 of a second) that the difference between compilers looks like system noise.** ***It must also be noted that the Delphi 2010 and Delphi XE compilers were run from the IDE which explains their slower execution.*** The interesting result, however, is EXE size. It has been apparent for a long time that each version of Delphi has been increasing the EXE size (i.e., code bloat). The results are backed up with our tests. **Every version has increased the resulting EXE in size from 914KB in Delphi 2010 to over 2 MB (2258KB) in Delphi XE6. The biggest jumps were from XE to XE2 and XE2 to XE3.** Note that this code bloat is not unexpected as every version of Delphi has been adding to the run-time library. Whether you find this code bloat good or bad is a matter of opinion as adding features is IMO a good thing. However, it is a shame that the Delphi linker has not kept pace with development and been aggressive in discarding unused code.

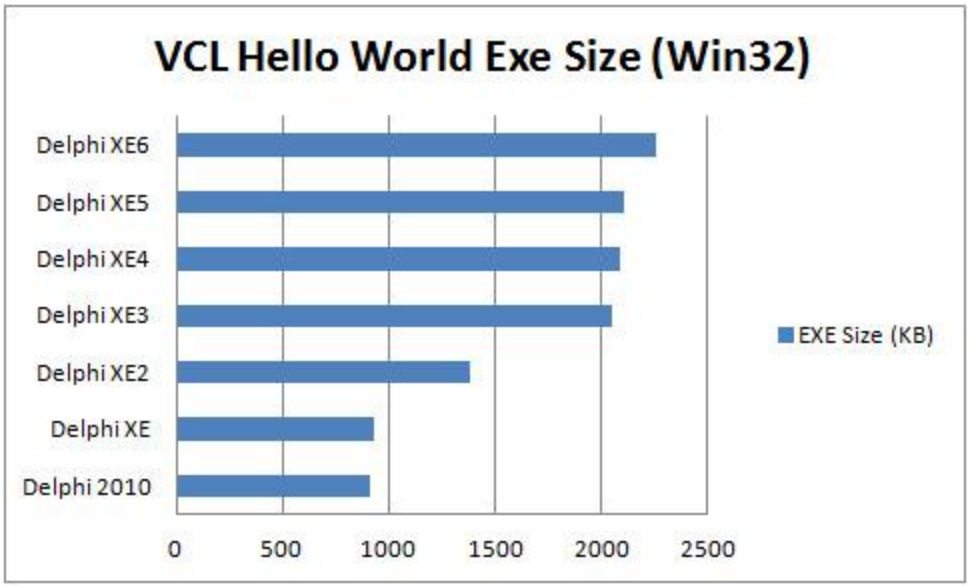


Figure 3 Comparison of EXE Sizes for VCL Hello World Application (Win32) with Delphi 2010 to XE6

Hello World, VCL, Win64

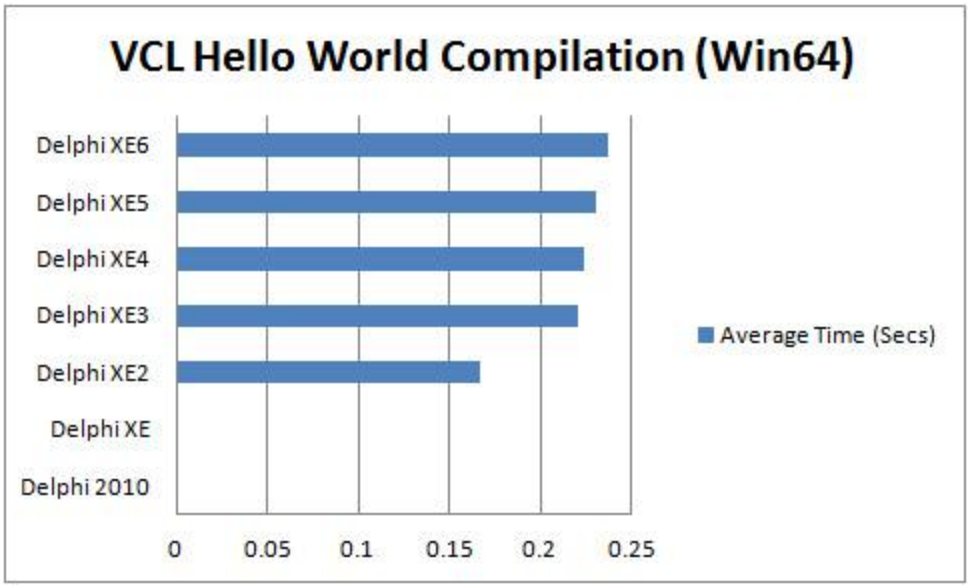


Figure 4 Comparison of compilation times for VCL Hello World Application (Win64) with Delphi 2010 to XE6

With the introduction of Delphi XE2, both Win64 and OSX support were added. For my next tests, I compiled the Hello World application with the Delphi Win64 compilers. Again, compilation was blazingly fast (less than 1/4 second) though each version of the compiler is very slightly slower than the previous one. However, EXE size has ballooned to about a MB more than the Win32 version so compilation differences are probably just hard drive limited. Again, **the EXE sizes are increasing with every version of Delphi. Delphi XE did not have a Win64 compiler so we cannot see the EXE size increase between those versions, but the jump from XE2 to XE3 is bad.**

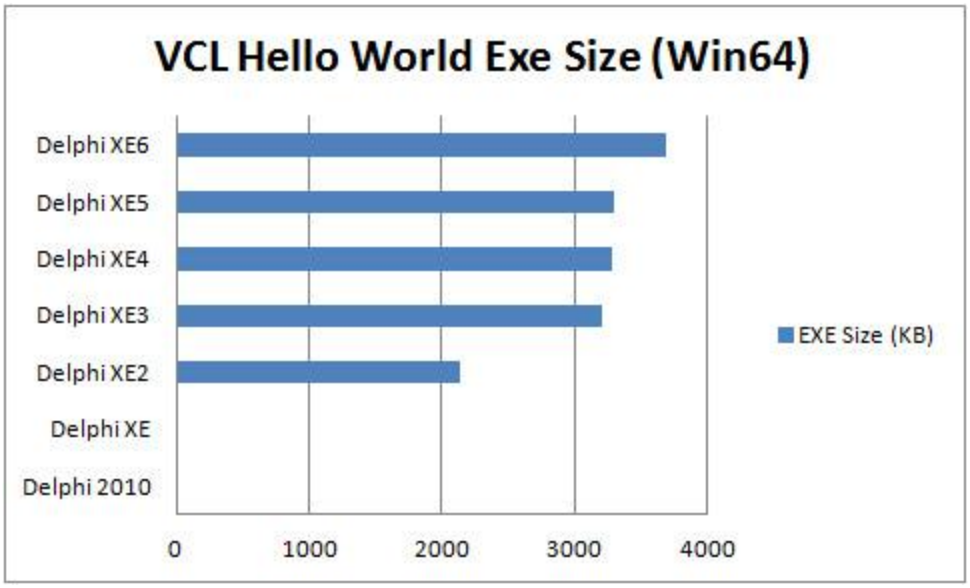


Figure 5 Comparison of EXE Sizes for VCL Hello World Application (Win64) with Delphi 2010 to XE6

Hello World, FMX, Win32

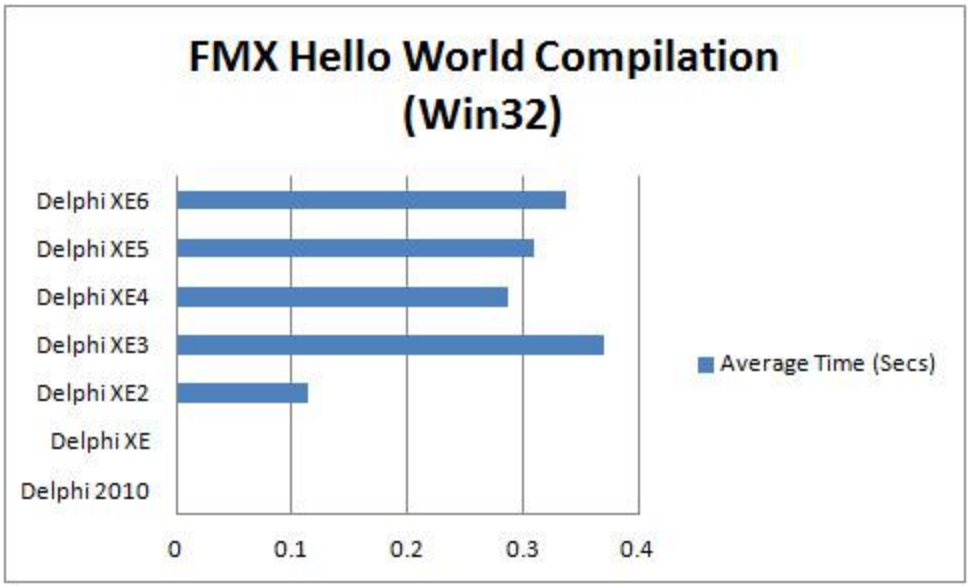


Figure 6 Comparison of compilation times for FMX Hello World Application (Win32) with Delphi XE2 to XE6

The compilers for Win32 and Win64 are the same as that used for the VCL application. The difference is the visual component library used (FMX instead of VCL). The Hello World FMX application uses TListBox, TMemo, TEdit, TCheckBox, etc, but instead of using Windows managed controls, FMX does all of the work itself. **As expected, FMX EXEs are significantly larger than their VCL counterparts. The surprise is by how much (over 2x as large) and how bad Delphi XE3 EXE sizes are (almost 4x larger). Delphi XE4 dramatically improved EXE sizes over XE3 but was not able to reduce the EXE size to XE2 levels. Since**

then, the EXE size has been slowly creeping up. Compilation speeds seem to track very closely to the EXE size so my guess is that we are seeing hard drive read/write times.

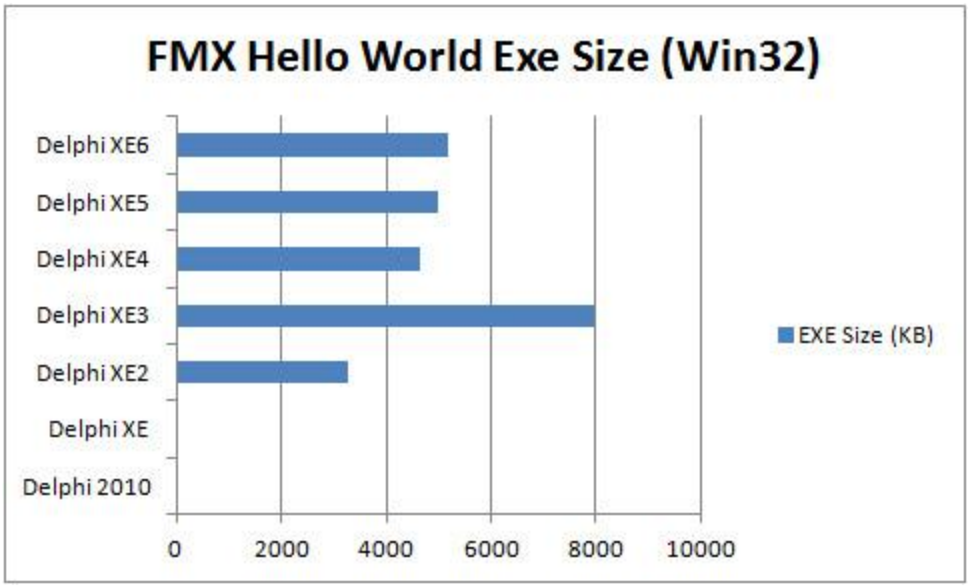


Figure 7 Comparison of EXE Sizes for FMX Hello World Application (Win32) with Delphi XE2 to XE6

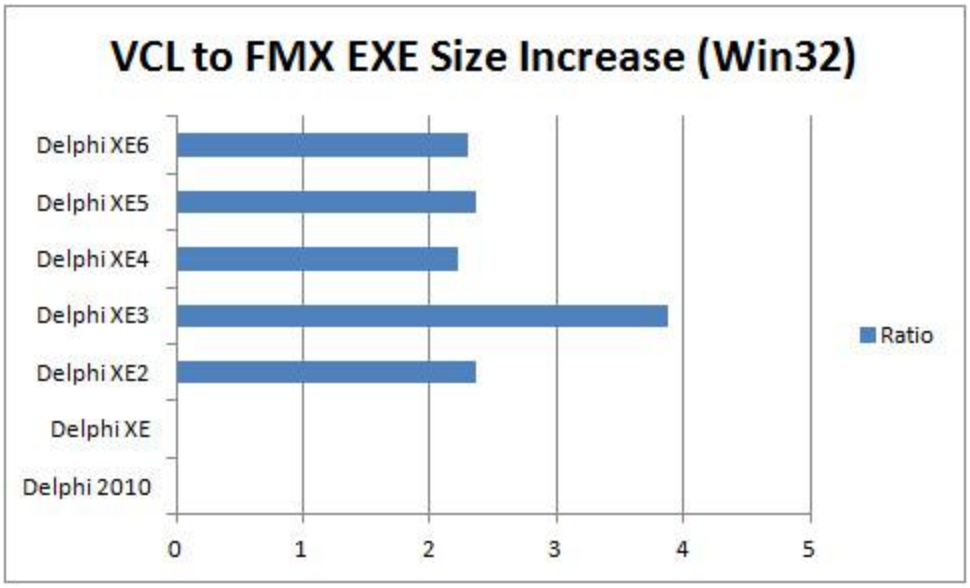


Figure 8 EXE Size increase between VCL and FMX Hello World applications (Win32) for Delphi XE2 to XE6

Hello World, FMX, Win64

The Win64 results closely mirror the Win32 results.

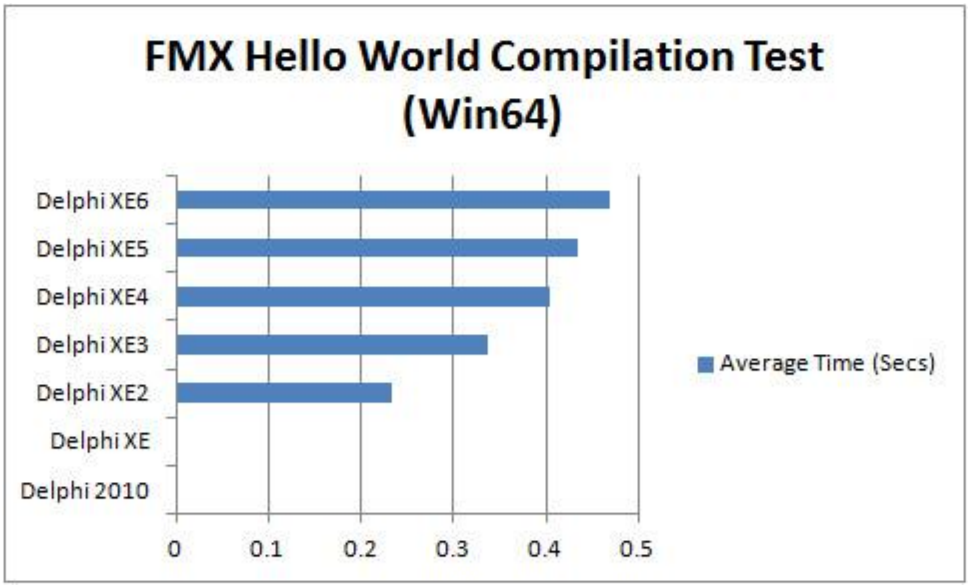


Figure 9 Comparison of compilation times for FMX Hello World Application (Win64) with Delphi XE2 to XE6

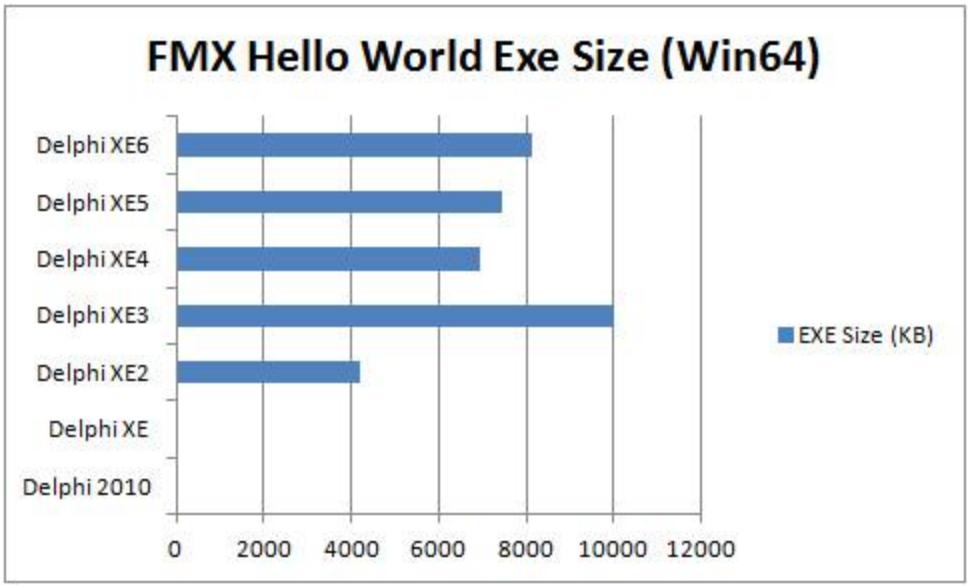


Figure 10 Comparison of EXE Sizes for FMX Hello World Application (Win64) with Delphi XE2 to XE6

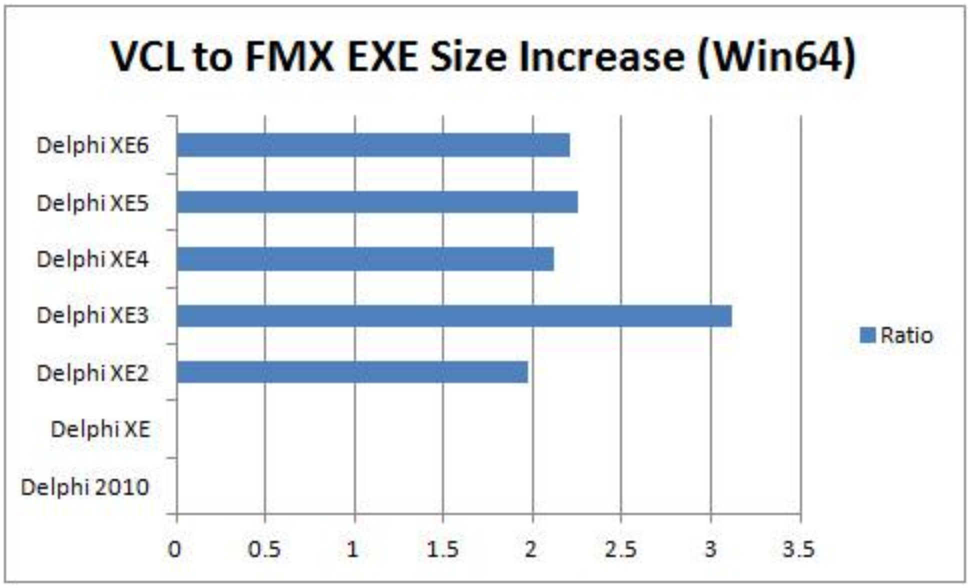


Figure 11 EXE Size increase between VCL and FMX Hello World applications (Win64) for Delphi XE2 to XE6

Hello World, FMX, OSX

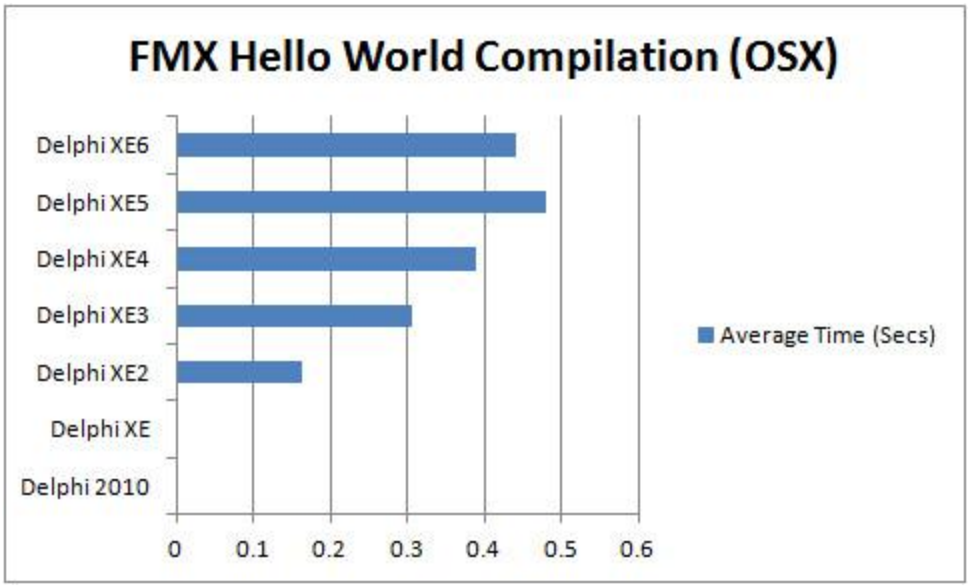


Figure 12 Comparison of compilation times for FMX Hello World Application (OSX) with Delphi XE2 to XE6

Finally, we get to compare a completely new platform! With the release of Delphi XE2, OSX support was added. **We see the same growth pattern of EXE size from Delphi XE2 to XE6, with Delphi XE3 again being the outlier. In this case, Delphi XE6 is very slightly faster than XE5 though I wouldn't read too much into it.** An interesting comparision is to compare the size of Delphi FMX applications on OSX (Win32) to their Windows counterparts (Win32). OSX applications are even larger than their Windows versions, approximately 1.5x to 2x larger. However, the OSX version from XE3 is not as comparatively huge as other Delphi versions. Also, it is interesting to see that the OSX compiler is as fast as the WinXX compilers. I believe that this is because the compiler was written by Embarcadero and does not use LLVM.

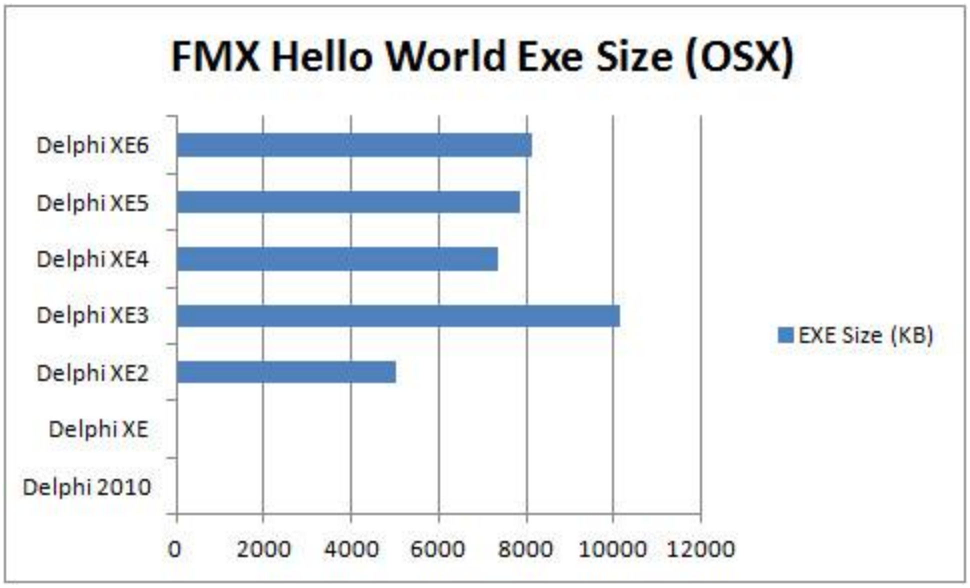


Figure 13 Comparison of EXE Sizes for FMX Hello World Application (OSX) with Delphi XE2 to XE6

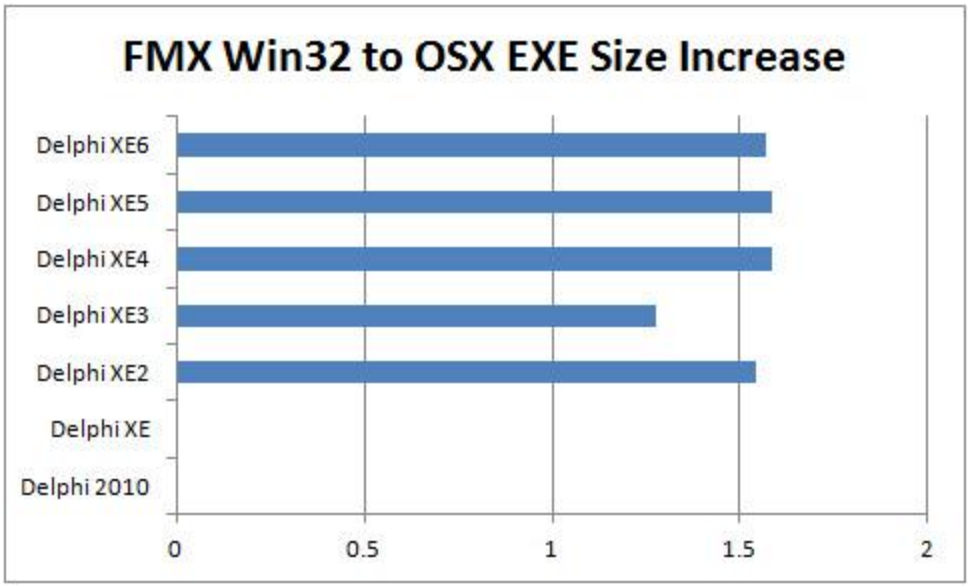


Figure 14 EXE Size increase between Win32 and OSX Hello World applications for Delphi XE2 to XE6

Hello World, Mobile, iOSSim and iOSDevice

In Delphi XE4, support was added for compiling and deploying apps to an iOS device (I am going to ignore the free pascal compiler from XE2). This next generation compiler creates ARM code and then uses XCode running on the Mac to deploy to an iOS device. I had difficultly compiling a release version of the Hello World Mobile app for my iPod Touch as I did not want to create a certificate for the app; I had to settle for compiling in debug for Delphi XE4. **Note that these tests do NOT include the XCode part of the compile and deploy equation.**

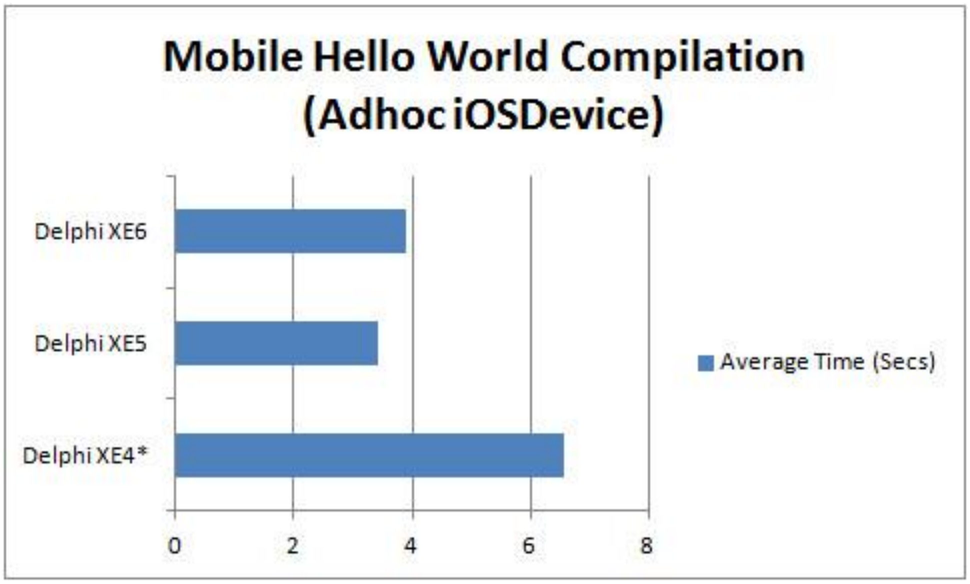


Figure 15 Compilation times for iOS Hello World apps for Delphi XE4 to XE6

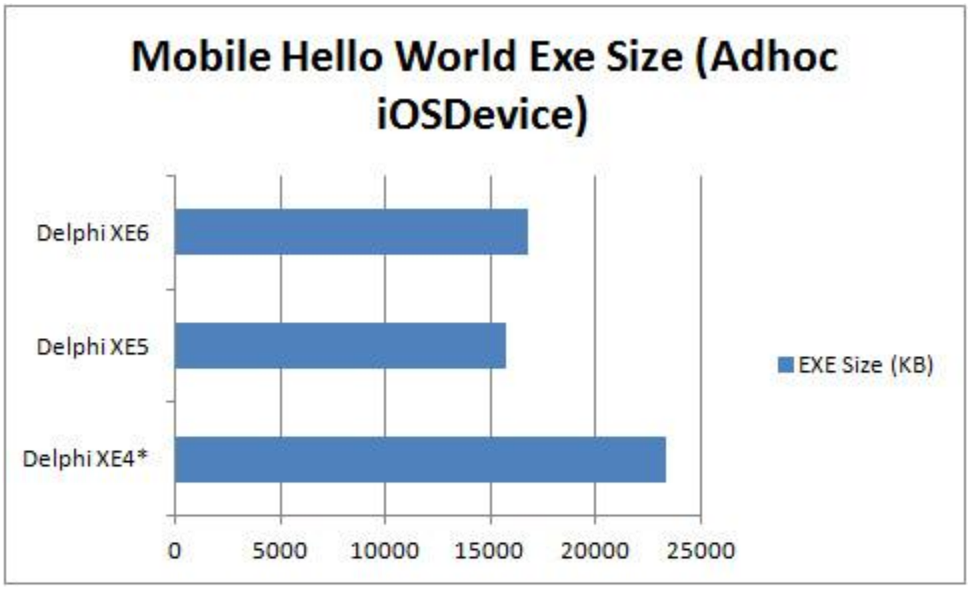


Figure 16 Comparison of EXE Sizes for Mobile FMX Hello World App (iOS) with Delphi XE4 to XE6

As predicted and even without the deployment step, iOS apps take significantly longer to compile than Win32, Win64, and OSX. While it is awesome that we can finally compile iOS apps, the compilation process is much slower (over 10x slower than compiling the same app for Win32) and much, much, MUCH slower when you finally deploy to device. **The move to XE6 (ignoring the spurious XE4 results) has not reduced compilation time or EXE size.**

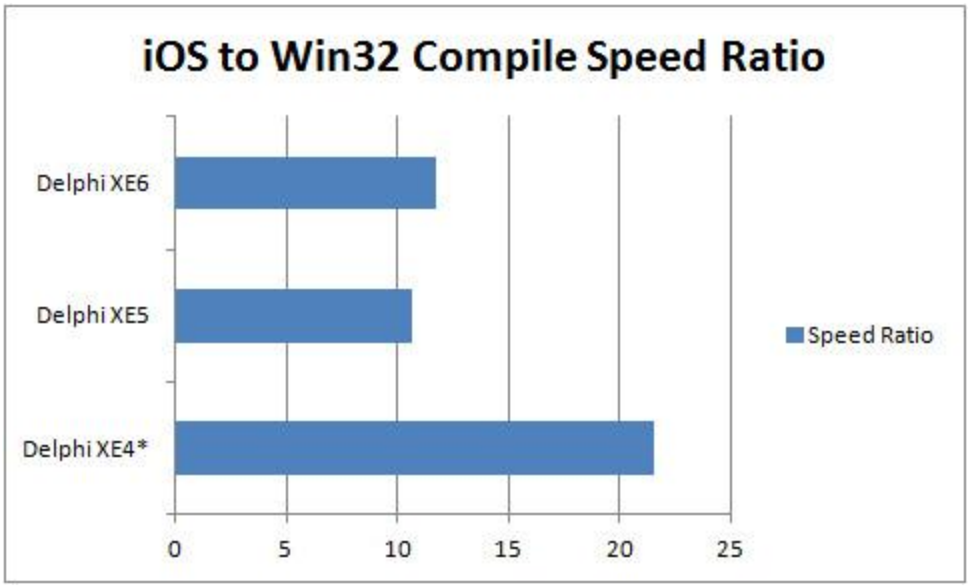


Figure 17 Comparison of compilation times for Mobile FMX Hello World App (iOS) with Delphi XE4 to XE6

Hello World, Mobile, Android

In Delphi XE5, support was finally added for compiling and deploying apps to an Android device. This next generation compiler creates ARM code and an APK file and then directly deploys it to an Android device. **Note these tests do not include the deployment times. They are also executed directly in the IDE instead of the command line.**

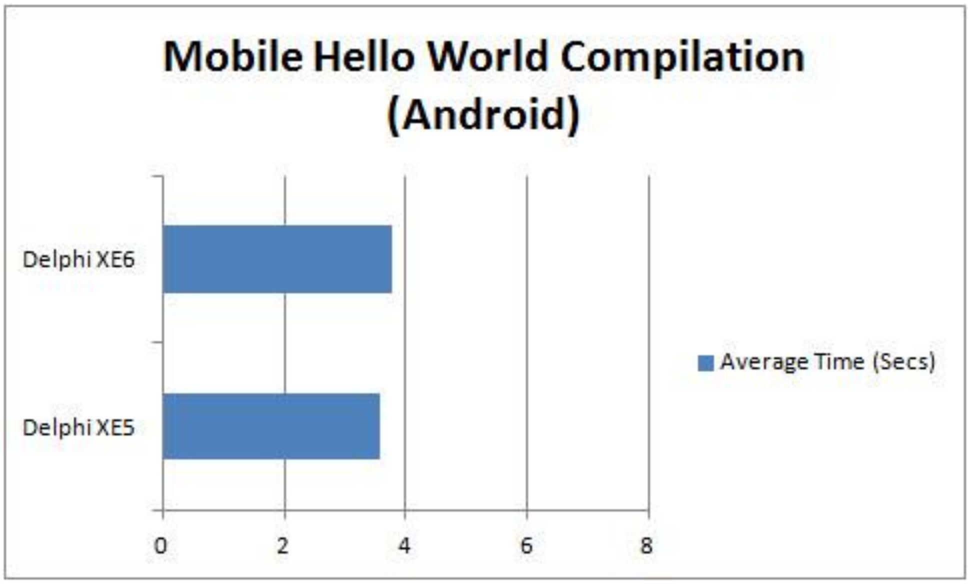


Figure 18 Comparison of compilation times for Mobile FMX Hello World App (Android) with Delphi XE5 to XE6

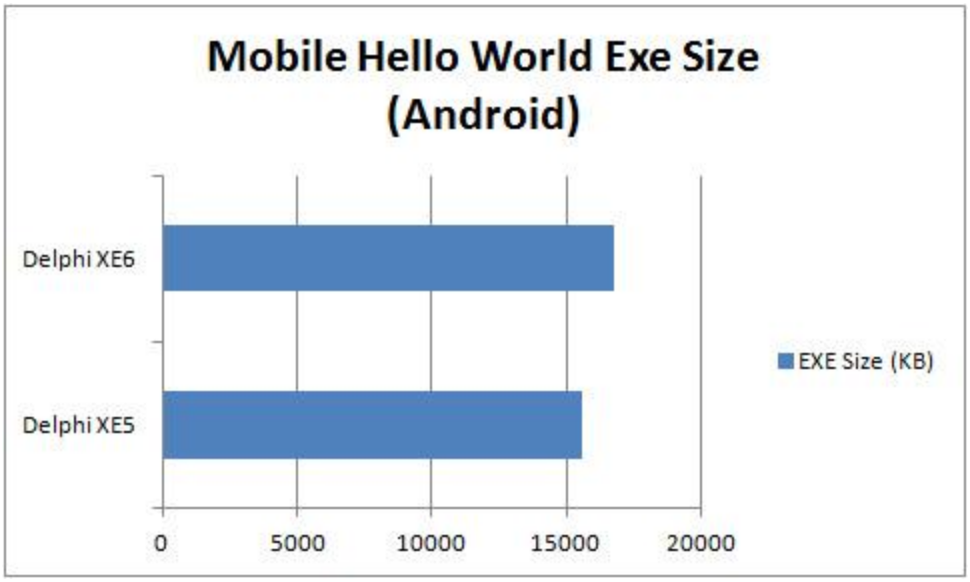


Figure 19 Comparison of EXE Sizes for Mobile FMX Hello World App (Android) with Delphi XE5 to XE6

Interestingly, without the deployment step (a big caveat admittedly :-)), iOS and Android EXE/APK sizes are comparable and compile in the same amount of time. **Like the iOS compilations, Android apps take significantly longer (almost 12x) to compile than Win32, Win64, and OSX. The move to XE6 has not reduced compilation time or EXE size over XE5.**

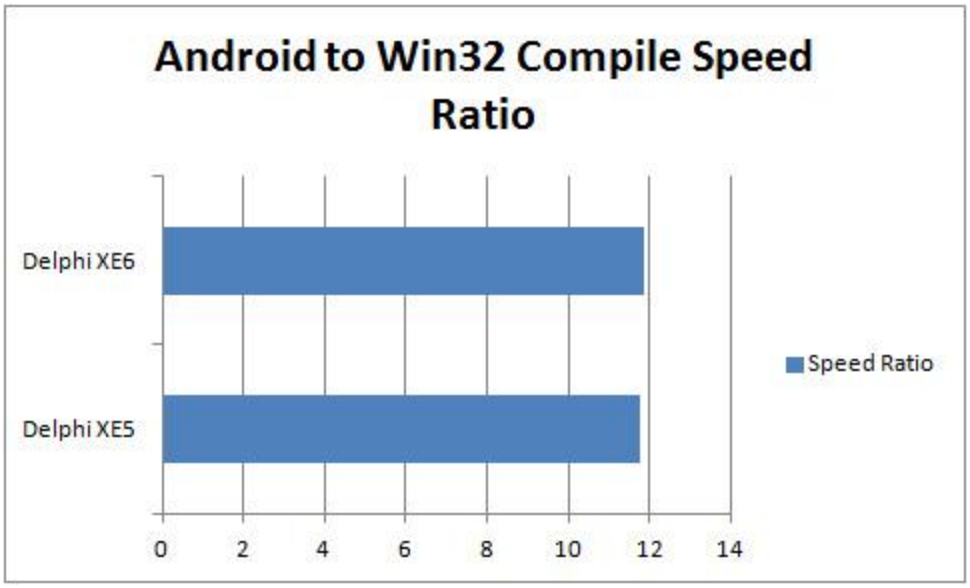


Figure 20 Compilation time increase between Win32 and Android Hello World apps for Delphi XE5 to XE6

Compilation Test Results Conclusion

Embarcadero has not been doing a lot of work in increasing compilation speed or reducing EXE size. Except for Delphi XE3 which seems an outlier, EXE sizes have been going up slowly but steadily as more is added to the RTL. The addition of FMX greatly increases EXE sizes. The ARM compilers are 10x-12x slower than the Win32 compiler even without the lengthy deployment step. Delphi XE6 does not improve either compilation speed or EXE size. However, this is not the most important of Delphi performance.

Hello World Speed Tests

Finally we get to the interesting part of these series of tests. In the previous section, we looked at compilation speed and EXE size for the Hello World project from the introduction. In this section, we are going to look at the execution speed for the Hello World project in Delphi 2010 to Delphi XE6, for Win32, Win64 (Delphi XE2-XE6 only**,** OSX (Delphi XE2-XE6 only), iOS (Delphi XE4-XE6 only), and Android (Delphi XE5-XE6 only). The Hello World project fills a TListBox or a TMemo with a number of strings with every click of a button. In our speed tests, we tested adding the string "Hello World" with a number appended after it. The button click adds 10, 100, 1000, or 10000 strings at a time. We are also going to test adding strings inside a BeginUpdate/EndUpdate, which should be significantly faster unless something has been broken. *Note that the charts display the execution time on the y-axis in milliseconds. The number of strings added as well as if BeginUpdate/EndUpdate were used are on the x-axis. Except for the VCL charts, the y-axis is logarithmic as the amount of time to add strings goes up exponentially with FMX applications.*

Hello World, VCL, Win32

The first test is with the VCL version of Hello World in Win32. Both the TListBox and TMemo pass their control handling to the underlying Windows control so unless Embarcadero broke something, we should see little change between the different versions of Delphi. And that is what we see. **Every version of Delphi is able to fill a TListBox with up to 10000 strings in under a second. Note that using BeginUpdate and EndUpdate methods speed up the execution by over 30x! Every version of Delphi is able to fill a TMemo with up to 10000 strings in just around 6 seconds with the BeginUpdate/EndUpdate tests finishing in a sixth of the time.**

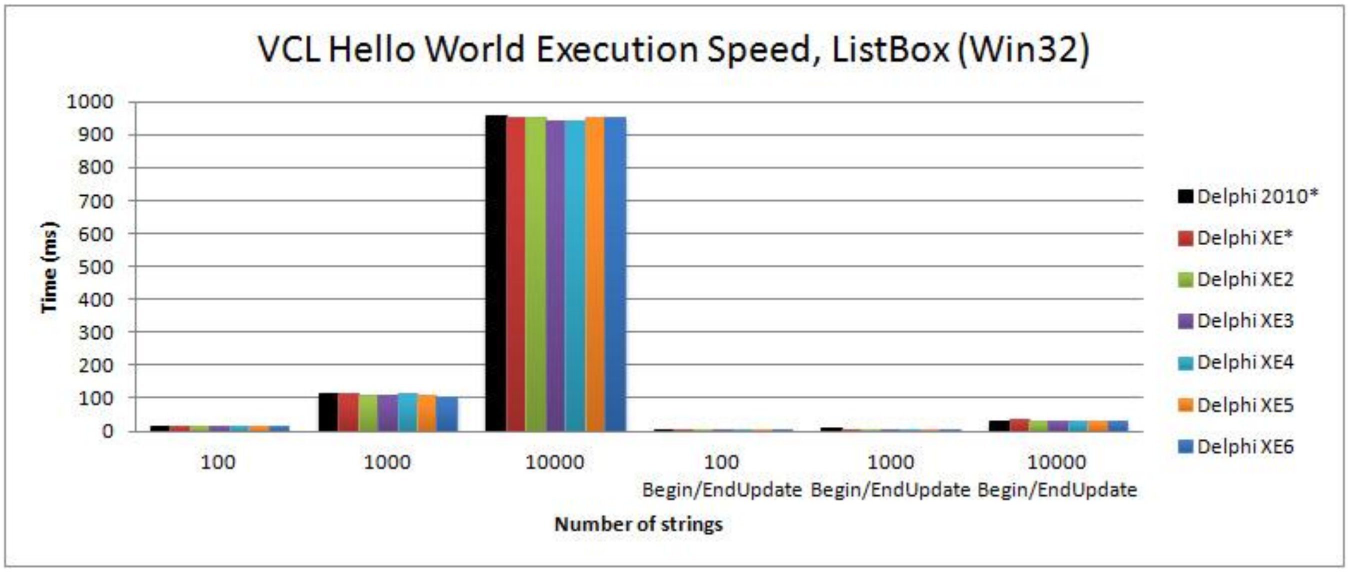


Figure 21 Comparison of execution speed for filling a TListBox in the VCL Hello World (Win32) with Delphi 2010 to XE6

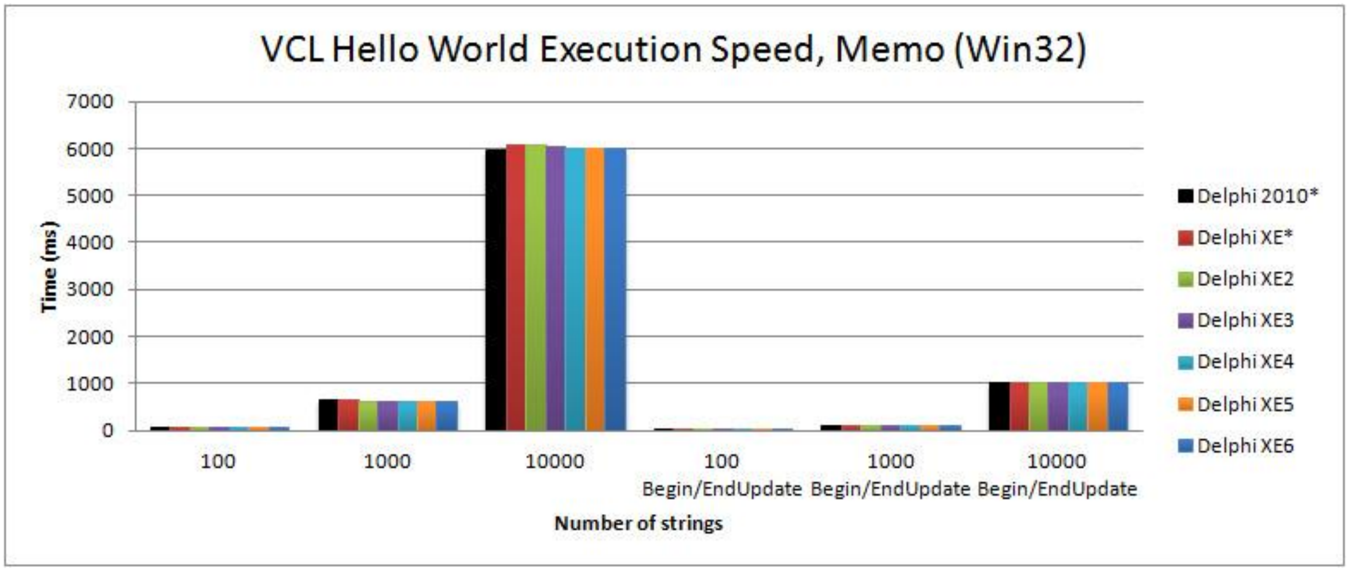


Figure 22 Comparison of execution speed for filling a TMemo in the VCL Hello World (Win32) with Delphi 2010 to XE6

Hello World, VCL, Win64

For the 64-bit Hello World application, we can only test with Delphi XE2 through XE6. **There are also no significant execution speed differences between the Delphi versions in Win64. It is, however, gratifying to see that the promise of Win64 Delphi applications running native in a Windows 64-bit OS and 64-bit CPU is realized as the Win64 versions run around 10-25% faster.** Even though 64-bit applications are larger and consume more memory, it can be worth it in order to see these speed increases.

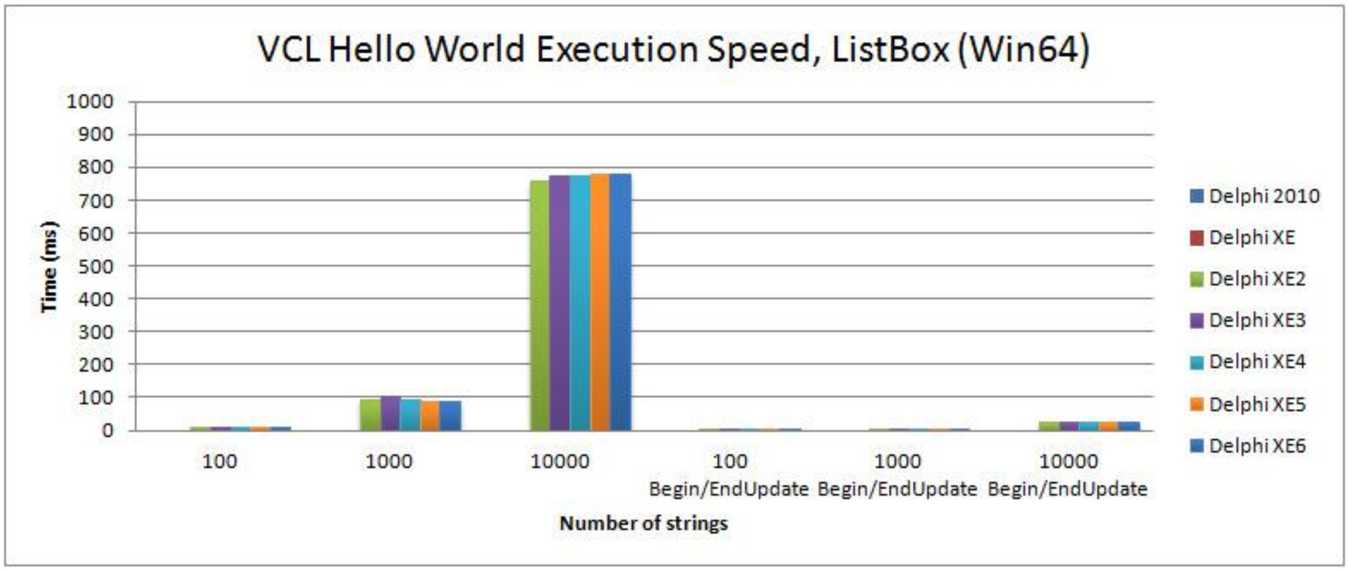


Figure 23 Comparison of execution speed for filling a TListBox in the VCL Hello World (Win64) with Delphi XE2 to XE6

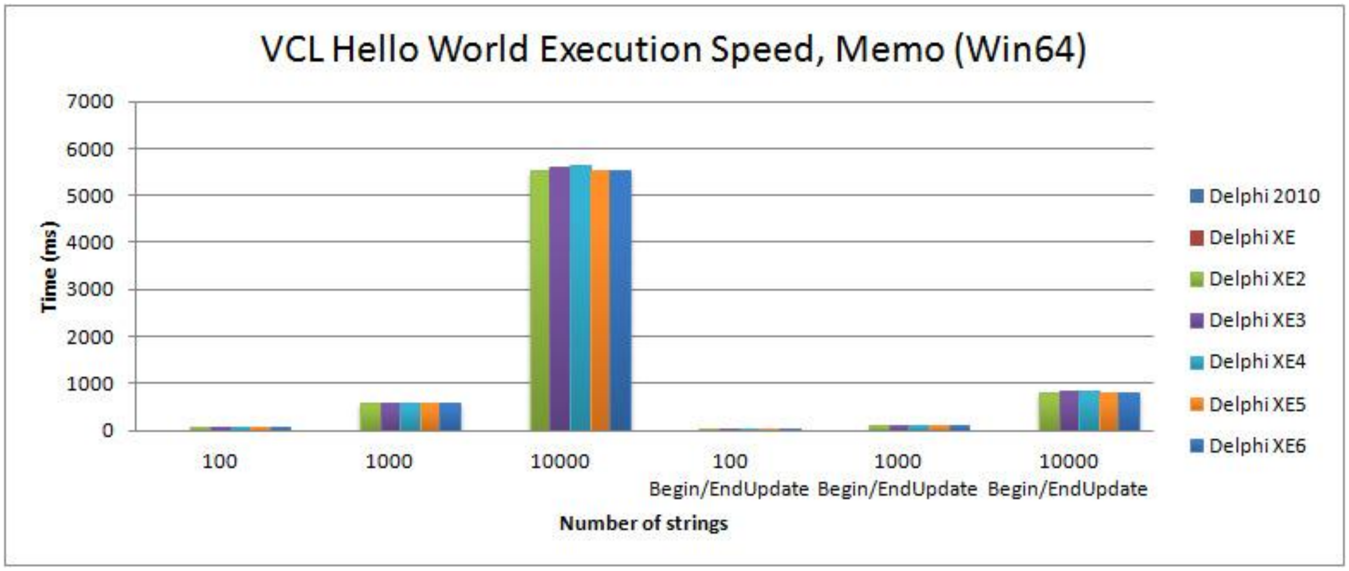


Figure 24 Comparison of execution speed for filling a TMemo in the VCL Hello World (Win64) with Delphi XE2 to XE6

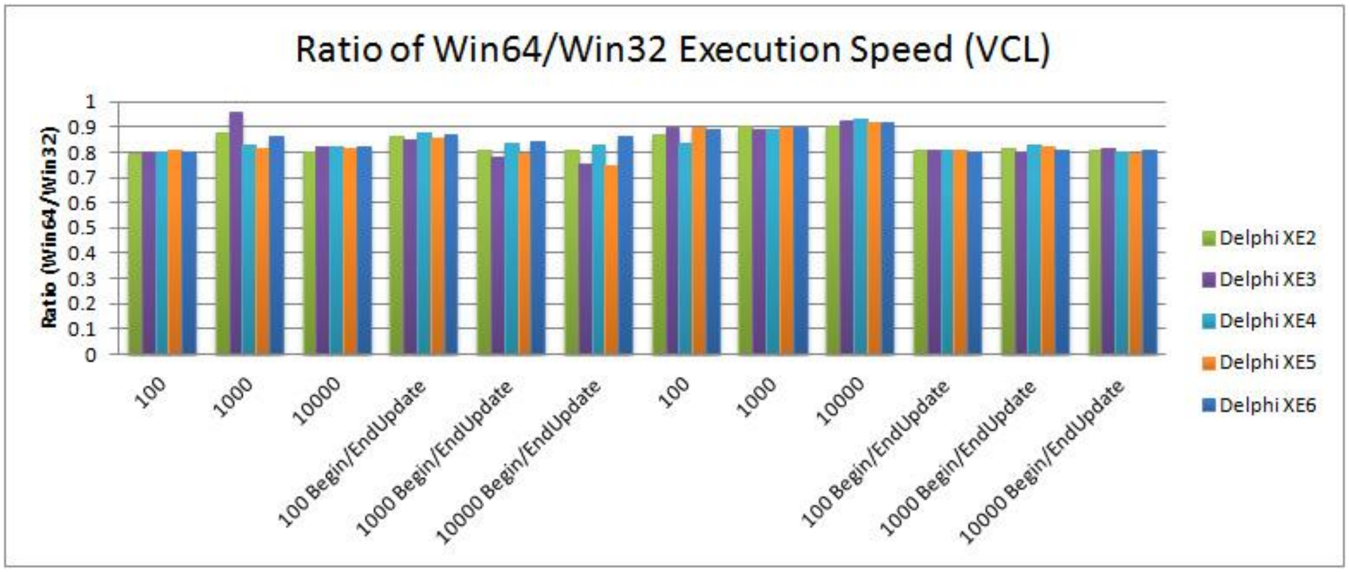


Figure 25 Ratio of VCL Hello World execution speed of Win64/Win32 running with Windows 7 64-bit and 64-bit CPU

Hello World, FMX, Win32

From here on out, we are going to be testing FMX applications. Everything... every pixel, every string, every button... is drawn by the FMX library and does not use OS controls. It is expected that the FMX applications will be, not only bigger (as we saw in the last post as all the drawing and interaction code must be compiled into the application), but may also draw GUI controls slower. Not only can Delphi applications not count on using OS native controls, but in general FMX controls are much, much richer than their VCL counterparts. The FMX TListBox implementation will be slower than the VCL implementation for this reason. However, this section is about exploring the execution differences between the various Delphi versions. What we hope to see is that the execution speed will improve with each new version of Delphi, especially with the latest Delphi XE6 and its mantra of "Quality, Performance, and Stability."

Note the FMX charts have a logarithmic y-axis for execution time. We used this because the execution times vary so greatly between a small number of items and a large number of items. Filling a TListBox or a TMemo with 10000 items would swamp out the other results and make them appear tiny and indistinguishable.

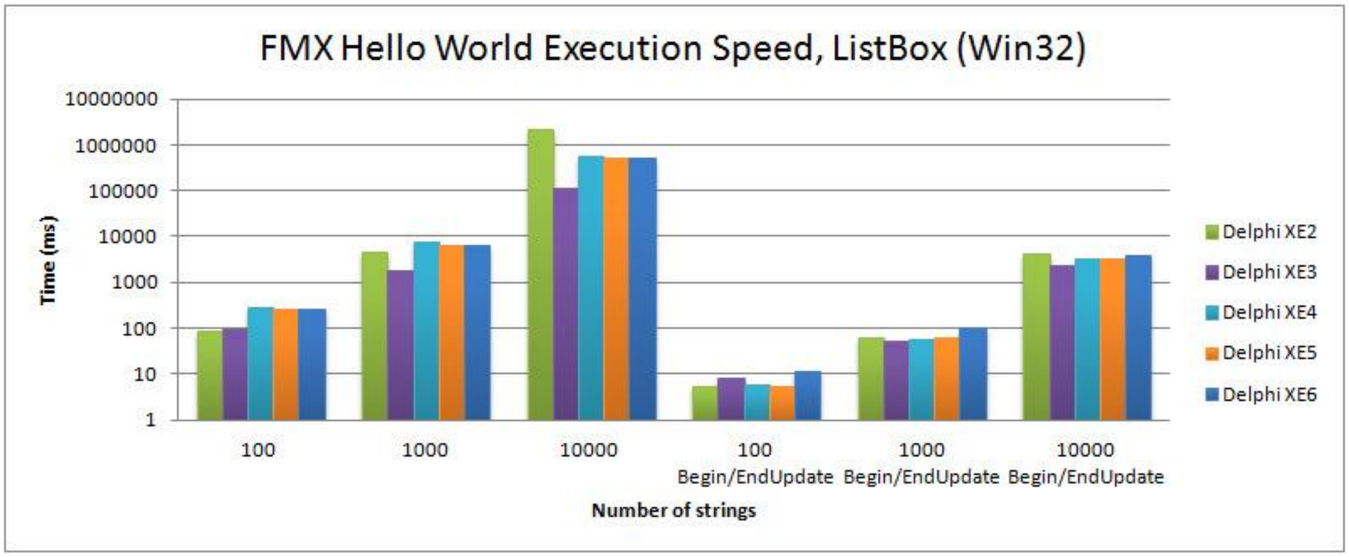


Figure 26 Comparison of execution speed for filling a TListBox in the FMX Hello World (Win32) with Delphi XE2 to XE6

With the Win32 version of the FMX Hello World application, *at small number of items,* the XE2 version is fastest using the TListBox. Surprisingly, *at higher numbers of items*, Delphi XE3 is fastest. **Delphi XE6 manages to be faster than Delphi XE4 and XE5 when not using the BeginUpdate/EndUpdate methods. Using BeginUpdate and EndUpdate, Delphi XE6 is slower than every prior version until we get up to 10000 items where Delphi XE2 is the slowest.** *(Update: I forgot to mention an important point about Delphi XE2 TListBox performance: the Clear method is broken in XE2. It doesn't seem to use BeginUpdate/EndUpdate internally so emptying the list goes exponential as well causing any use of Delphi XE2's TListBox to be very painful)*

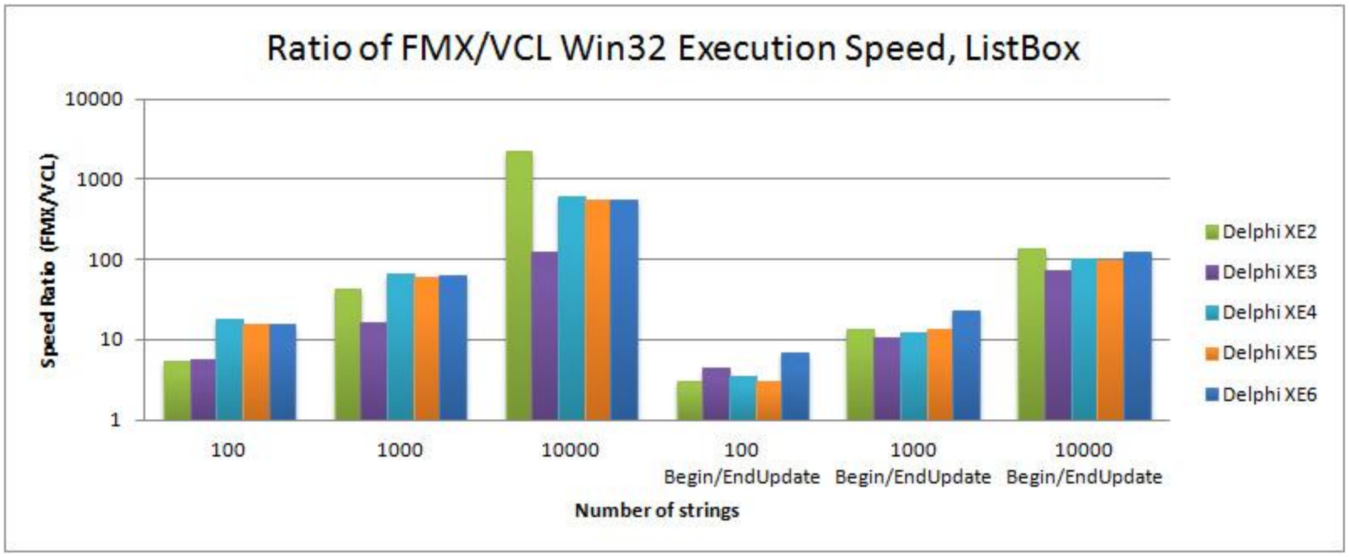


Figure 27 Ratio of FMX to VCL execution speed for TListBox in Delphi XE2 to XE6

Comparing FMX execution time to their VCL counterparts, the TListBox is much slower (there is a reason that Embarcadero recommends using TListView for lots of items). **At its best (100 items using BeginUpdate and EndUpdate), the FMX TListBox is 3x (XE5) to 6.76x (XE6) times slower. Even using the BeginUpdate and EndUpdate methods, the FMX TListBox is much slower when adding 10000 items, from 72x (XE3) to a staggering 2258x (XE2, without BeginUpdate/EndUpdate) slower.**

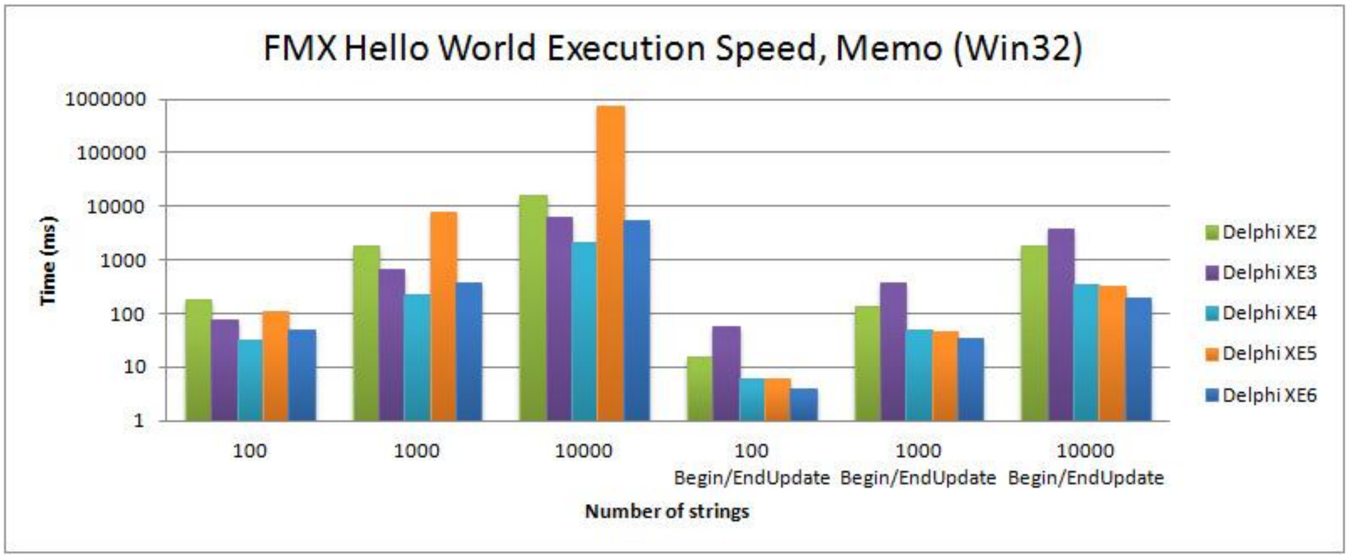


Figure 28 Comparison of execution speed for filling a TMemo in the FMX Hello World (Win32) with Delphi XE2 to XE6

Filling a TMemo, the story is completely different: **Delphi XE6 is faster than every other version of Delphi except XE4 when not using BeginUpdate/EndUpdate, and faster than all other versions of Delphi when using BeginUpdate/EndUpdate (even the VCL versions)!** Delphi XE5 is the big loser when the BeginUpdate/EndUpdate methods are not used. If they are used, Delphi XE3 has the slowest FMX TListBox.

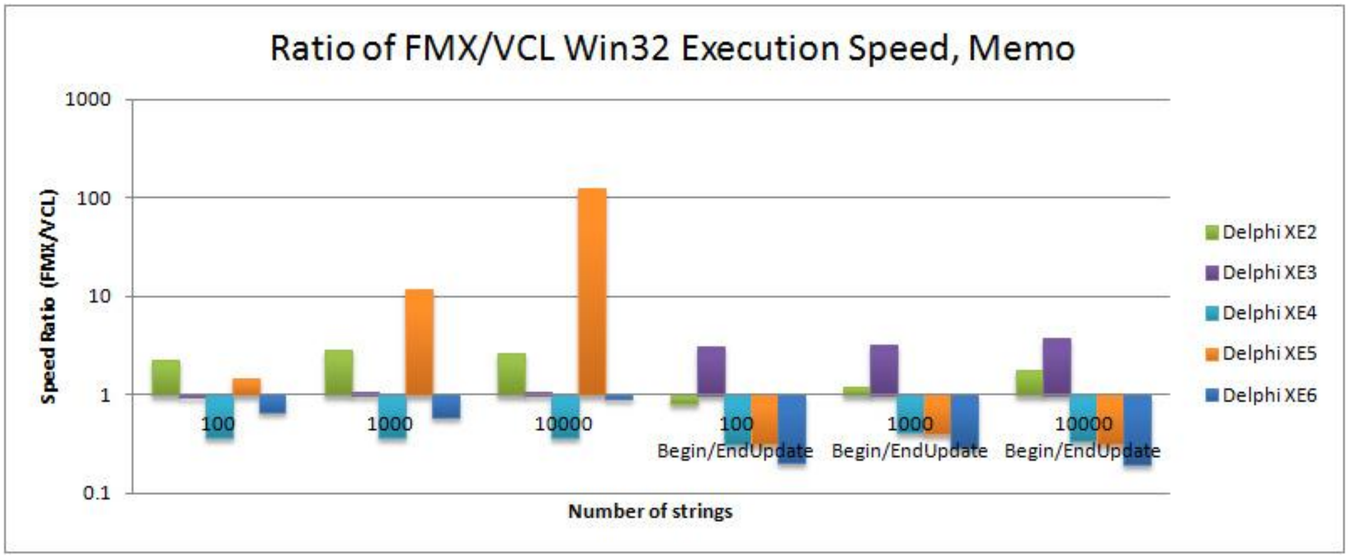


Figure 29 Ratio of FMX to VCL execution speed for TMemo in Delphi XE2 to XE6

As stated, Delphi XE6 FMX TMemo is even faster than the VCL TMemo, even without using BeginUpdate/EndUpdate; **however, the comparison is not fair as the VCL TMemo draws every string as it**

is added while the FMX TMemo does not. As anyone who has used the FMX TMemo can attest, the FMX TMemo draws very slowly, especially compared to the VCL TMemo.

Hello World, FMX, Win64

The FMX Win64 speed tests for filling a TListBox hold no new surprises and closely mirror the Win32 tests. With the FMX Hello World application, at small number of items, the XE2 version is fastest using the TListBox. **However, as mentioned in the update to last week's post, the Delphi XE2 TListBox is crippled when clearing items (which is not reflected here) and shows exponential times in clearing the list box.** At higher numbers of items, Delphi XE3 is fastest. Delphi XE6 manages to be insignificantly faster than Delphi XE4 and XE5 when not using the BeginUpdate/EndUpdate methods. Using BeginUpdate and EndUpdate, Delphi XE6 is slower than every prior version until we get up to 10000 items where Delphi XE2 is the slowest.

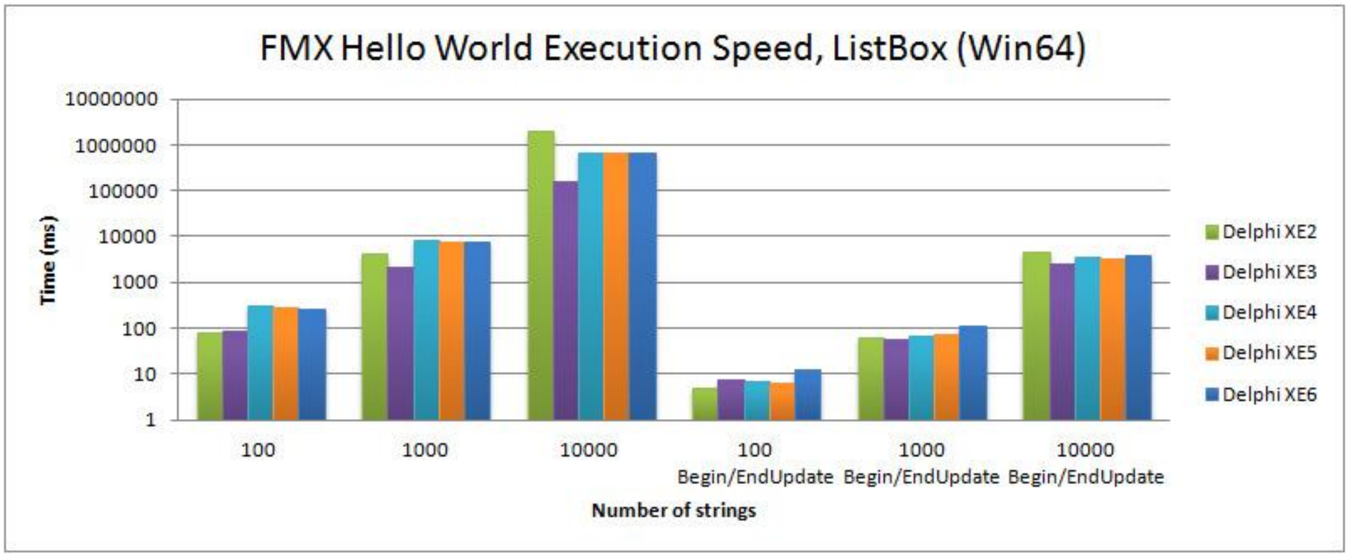


Figure 30 Comparison of execution speed for filling a TListBox in the FMX Hello World (Win64) with Delphi XE2 to XE6

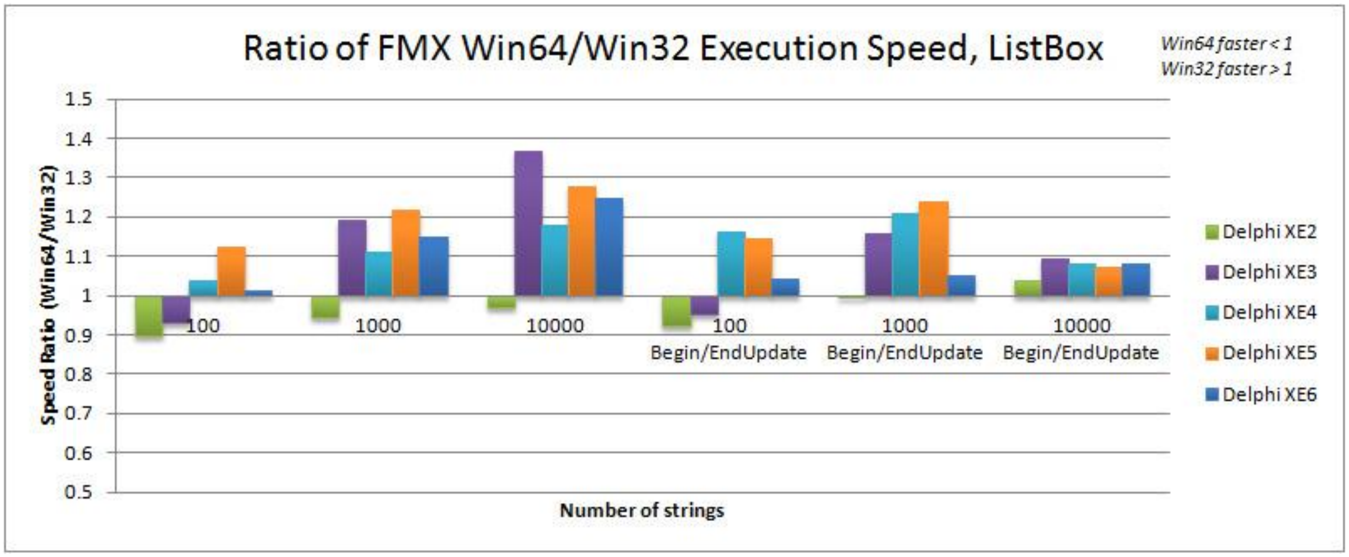


Figure 31 Ratio of Win64 to Win32 execution speed for TListBox in Delphi XE2 to XE6 (FMX)

Comparing Win64 FMX to Win32 FMX execution, surprisingly only Delphi XE2 manages to be faster in Win64 on a Windows 64-bit OS and machine (though even it cannot do it when using BeginUpdate/Endupdate methods at 10000 points). Delphi XE3, at a small number of items is faster filling a TListBox in Win64 than Win32. **Delphi XE4, XE5, and XE6 for some reason are all slower filling a TListBox in Win64 vs Win32 when executed on a Windows 64-bit OS and machine.**

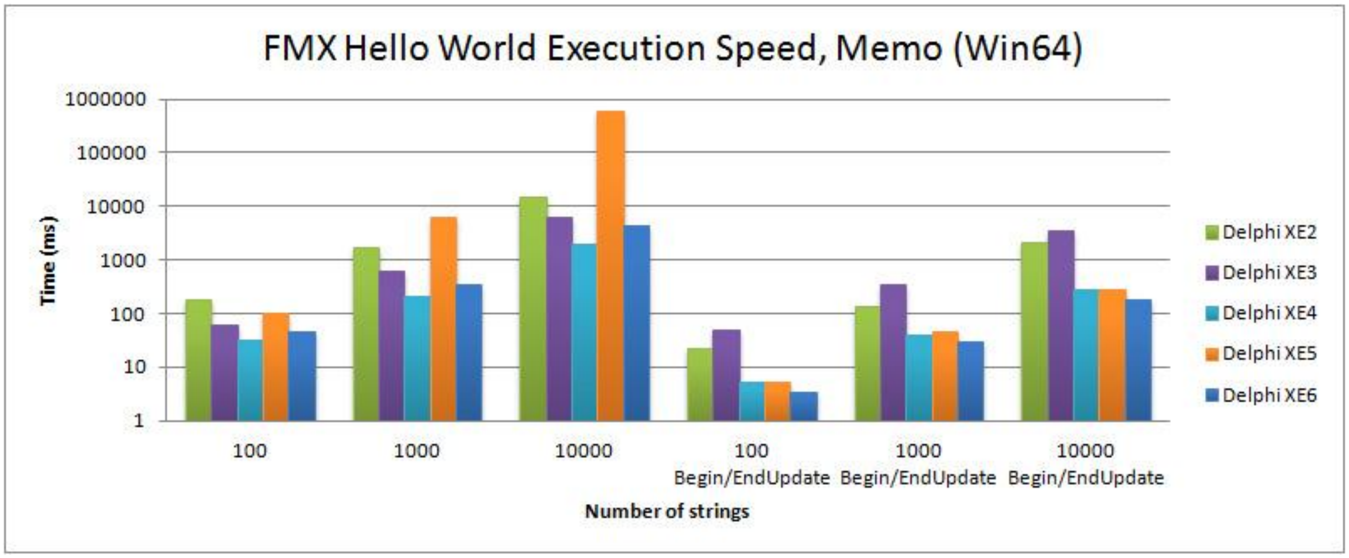


Figure 32 Comparison of execution speed for filling a TMemo in the FMX Hello World (Win64) with Delphi XE2 to XE6

Filling a TMemo, the picture changes. Because the FMX version of TMemo does not draw each line as it is added, the Delphi versions manage to maintain a speedy pace. **Delphi XE4 is fastest when filling a TMemo with no BeginUpdate/EndUpdate method calls. Delphi XE6 is strong again and is the fastest when using the BeginUpdate/EndUpdate methods.**

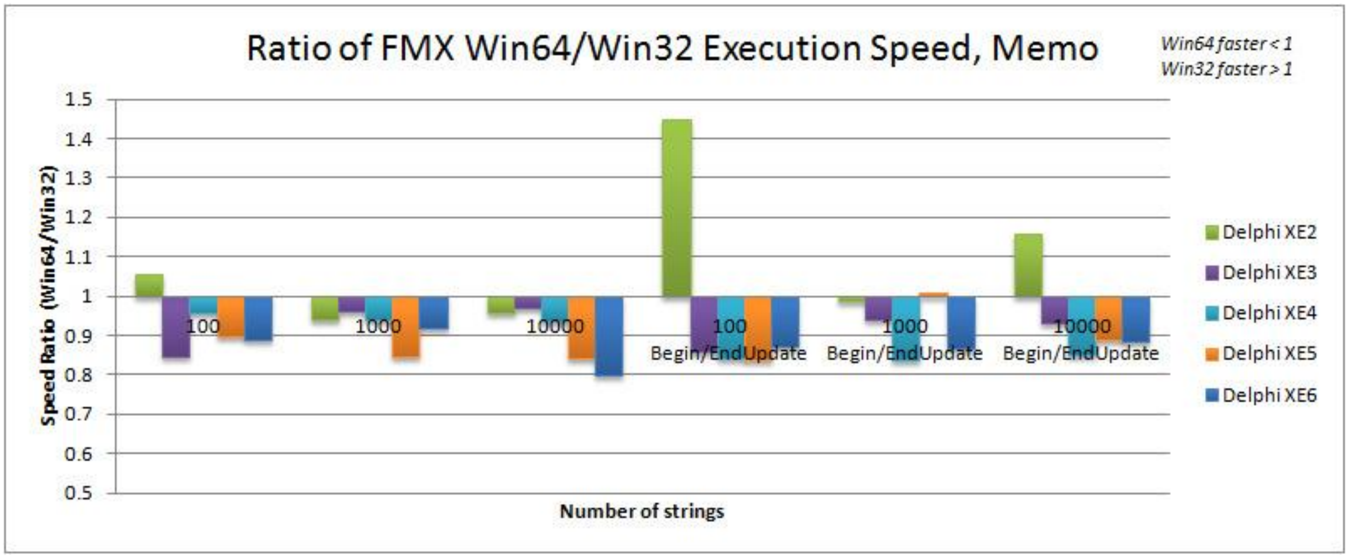


Figure 33 Ratio of Win64 to Win32 execution speed for TMemo in Delphi XE2 to XE6 (FMX)

For all versions of Delphi except for XE2 in some cases, the 64-bit version of the FMX Hello World application is faster than its 32-bit FMX counterpart when filling a TMemo.

Hello World, FMX, OSX

Now for the really exciting tests, how do the different versions of Delphi do on other platforms? Our first tests will be the Hello World application running on OSX. Since Delphi XE2, there has been support for creating applications for the Mac. For our tests, we use a 2.3 GHz Intel Core I7 Mac Mini with 4GB 1600 MHz DDR3 RAM running OSX 10.9.2.

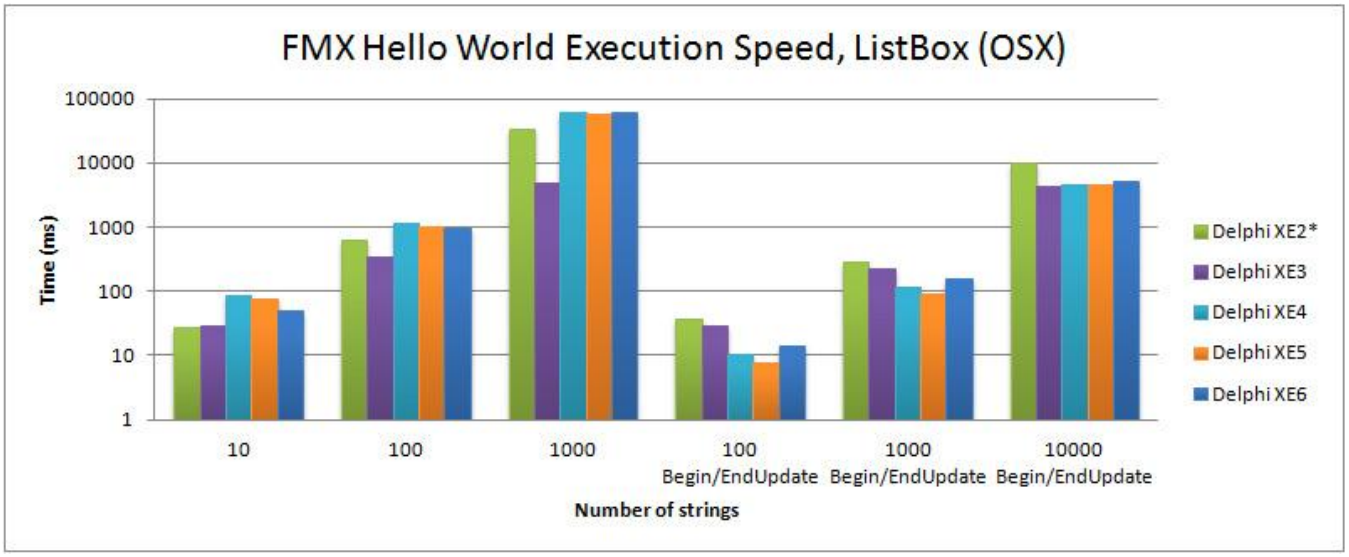


Figure 34 Comparison of execution speed for filling a TListBox in the FMX Hello World (OSX) with Delphi XE2 to XE6

The results for filling a TListBox in OSX is interesting. I**n general, Delphi XE3 is the fastest when BeginUpdate/EndUpdate are not called. When BeginUpdate and EndUpdate are called, Delphi XE5 has a very strong showing**, beating out all other versions except where Delphi XE3 manages to eke out a win by 1/10 of a second for 10000 items.

Delphi XE6 gives competent but not winning numbers. It is faster or practically equal to Delphi XE4 and XE5 when not using the BeginUpdate/EndUpdate methods. If BeginUpdate and EndUpdate are called, Delphi XE6 is faster than XE2 and XE3 for 1000 items and under, and almost tied with XE3-XE5 at 10000 items.

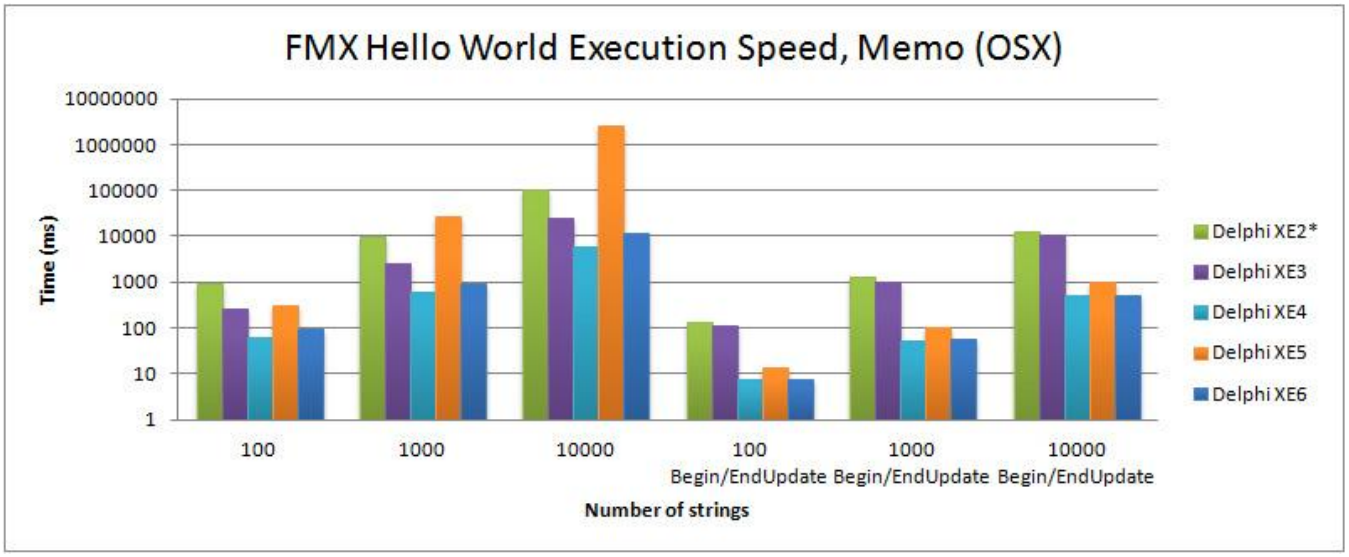


Figure 35 Comparison of execution speed for filling a TMemo in the FMX Hello World (OSX) with Delphi XE2 to XE6

When filling a TMemo, Delphi XE4 is particularly strong on OSX. Delphi XE4 is faster than every other version. Delphi XE6 gives an excellent showing and is in second place for all TMemo tests, and actually equalling XE4 with 100 items and BeginUpdate/EndUpdate calls.

Hello World, FMX, Win32 vs OSX Win32

Unfortunately, the Windows machine used for testing is not exactly comparable to the Mac Mini. However, the Windows Machine specs (Windows 7 64-bit Intel I7 930 @ 2.8 GHz CPU, and 6 GB RAM) are *close* to the Mac Mini specs (OSX 64-bit Intel I7 @ 2.3 GHz with 4GB RAM), so it will be very interesting to do comparisons of the FMX Win32 vs FMX OSX Hello World.

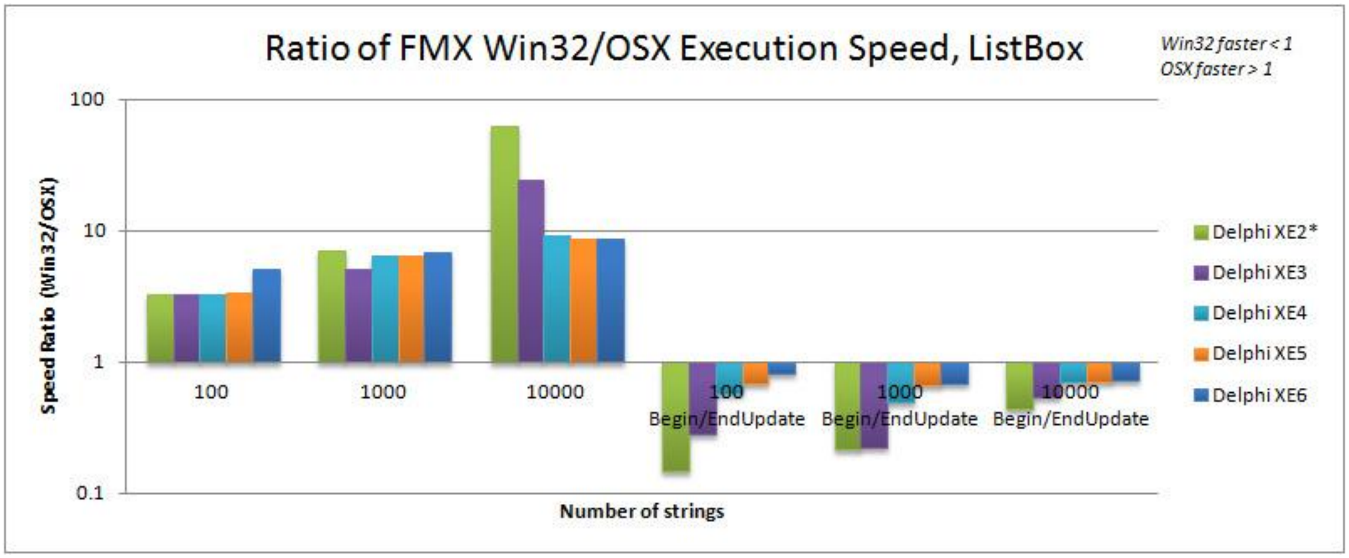


Figure 36 Ratio of Win32 to OSX execution speed for TListBox in Delphi XE2 to XE6 (FMX). Note that test machines are not exactly comparable.

Surprisingly, adding items to a TListBox is faster on a Mac when the BeginUpdate and EndUpdate methods are ***not* called.** Once, BeginUpdate and EndUpdate are called, we get the expected results and the Windows box wins.

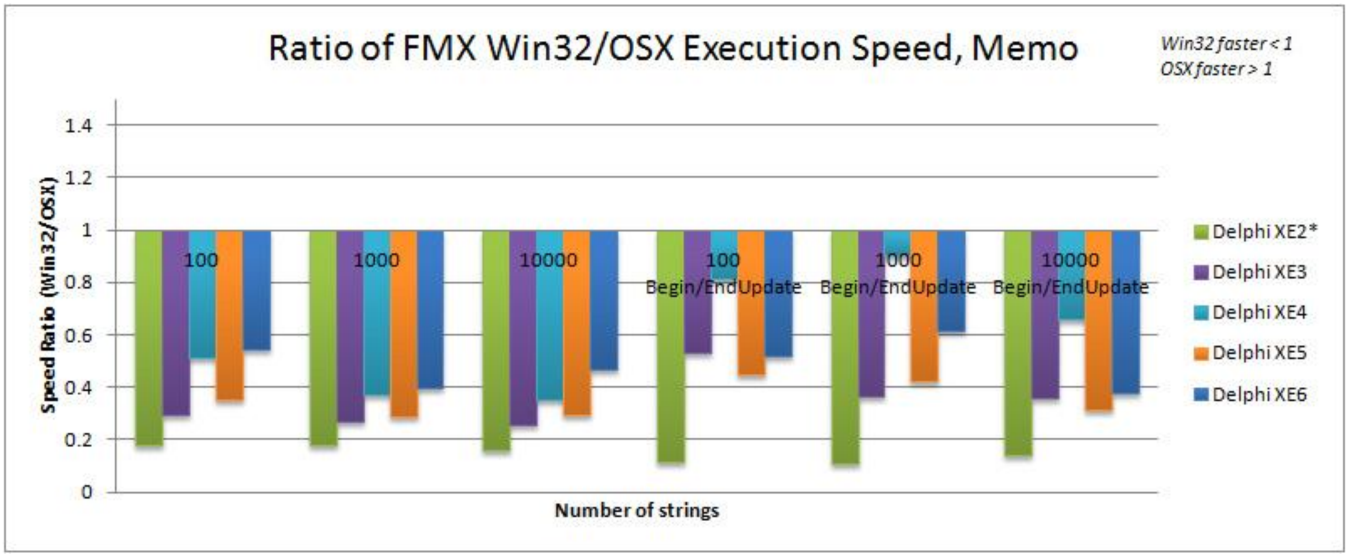


Figure 37 Ratio of Win32 to OSX execution speed for TMemo in Delphi XE2 to XE6 (FMX). Note that test machines are not exactly comparable.

Adding lines to a TMemo is a rout and the Windows application wins across the board and for all versions of Delphi.

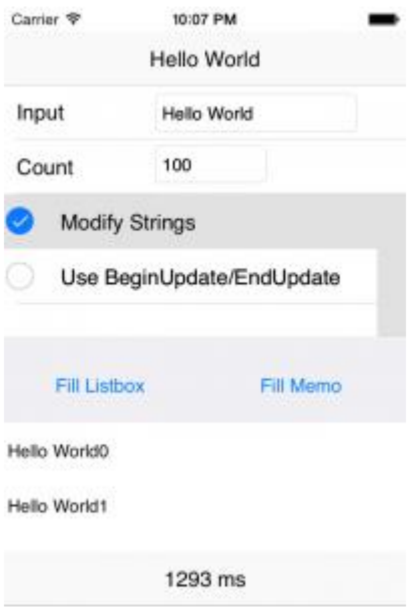
Hello World Mobile Speed Tests

Now that we have finished comparing execution on Win32, Win64, and OSX, we are finally going to compare performance on mobile devices! We created a mobile app version of the Hello World project. It is directly comparable to the desktop version, using TMemo, TListBox, TButtons, and TLabels (see Figure)

For testing iOS apps, we used an iPod Touch with 32GB. For testing Android apps, we used a Nexus 7 (2013) with 32GB.

Because the speed of the mobile devices is so much slower than our desktop machines, we tested adding 10, 100, and 1000 items or lines instead of 100, 1000, and 10000 like the desktop versions. This was only necessary when the BeginUpdate and EndUpdate methods were not called.

Please notice that the FMX charts have a logarithmic y-axis for execution time. With FMX, execution times vary so greatly between a small number of items and a large number of items that filling a TListBox or a TMemo with 1000 items would swamp out the other results and make them appear tiny and indistinguishable.



Input: Hello World
Count: 100
✓ Modify Strings
Use BeginUpdate/EndUpdate
Hello World0
Hello World1
1293 ms

Figure 38 Hello World Mobile App

Hello World, Mobile FMX, iOS

Our first tests are with filling the TListBox on an iOS device. Since Delphi XE4, Delphi has supported deploying to iOS devices. Note that the Delphi XE4 app was compiled with **debug**. Delphi XE5 and XE6 can deploy **release** versions directly to the device without requiring certificates. However, this was not true with Delphi XE4 but as we will see, it doesn't seem to have slowed the app down.

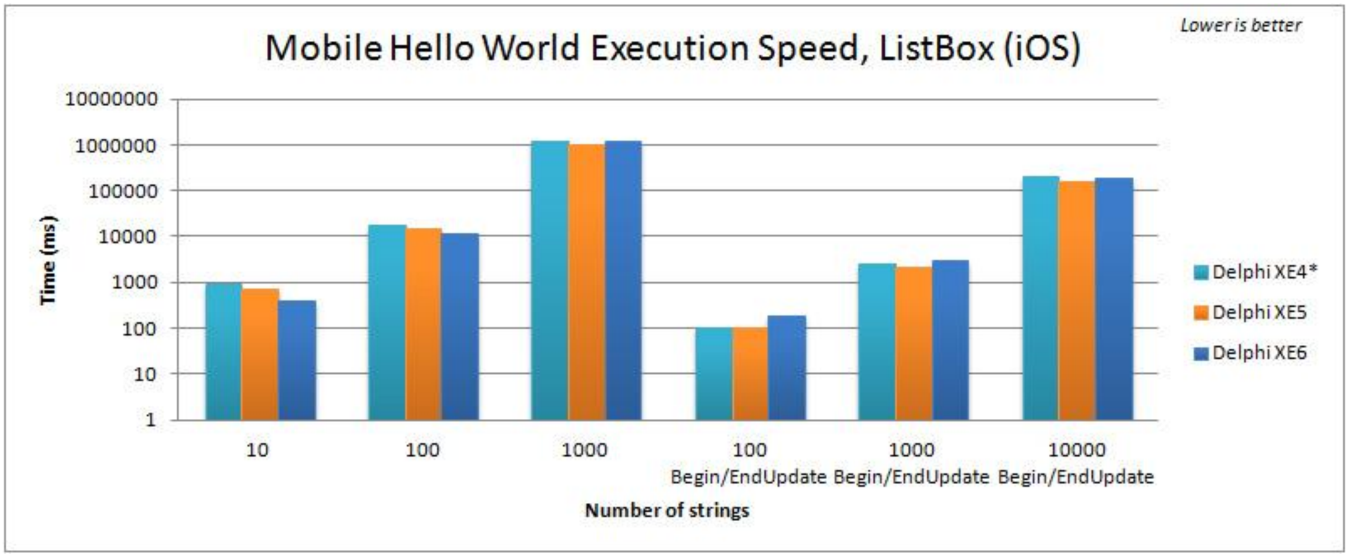


Figure 39 Comparison of execution speed for filling a TListBox in the Mobile FMX Hello World (iOS) with Delphi XE4 to XE6

Delphi XE6 does well when filling a TListBox on an iOS device. For 10 items, it manages to be over 2x faster (416 ms) than the slowest version, Delphi XE4, when BeginUpdate/EndUpdate are not called. At 100 items, it is still 1.5x faster (11.9 secs) than XE4. Only at 1000 items is it slowest, however, "slowest" is relative here as Delphi XE5 which is fastest still took over 16 minutes. When BeginUpdate/EndUpdate are used while filling a TListBox, Delphi XE5 is the fastest. Delphi XE6 is last in this case, but the differences are minor as it fills a TListBox with 100 items in 189 ms compared to XE5's 102.3, with 1000 items in 3.1 secs compared to XE5's 2.2 seconds, and with 10000 all of the Delphi's are slow, taking 2.8 minutes for XE5 and 3.19 minutes for XE6.

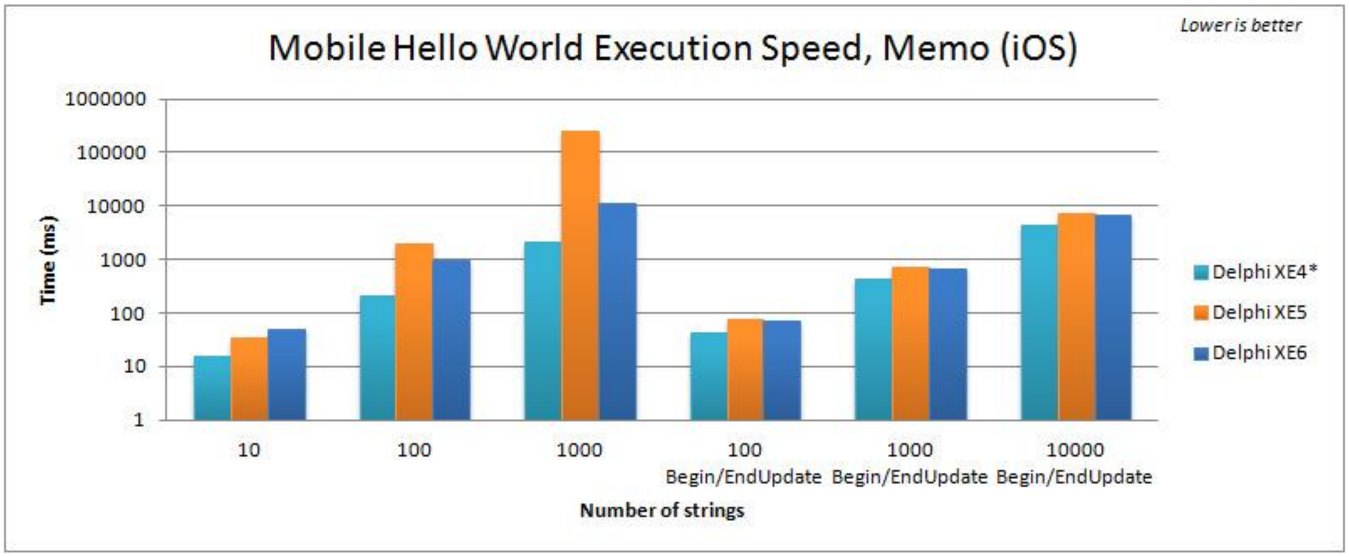


Figure 40 Comparison of execution speed for filling a TMemo in the Mobile FMX Hello World (iOS) with Delphi XE4 to XE6

It must be mentioned that Delphi XE4 did well filling a TListBox when the BeginUpdate and EndUpdate methods are called, nearly equaling XE5. **This is important as Delphi XE4 completely dominates when filling a TMemo, with or without calls to BeginUpdate/EndUpdate.** Delphi XE6 is in second managing to beat XE5 in every case except when filling a TMemo with 10 lines (51 ms vs 33 ms).

Hello World, Mobile FMX, Android

Our last platform is Android. Note that only Delphi XE5 and XE6 can target this platform.

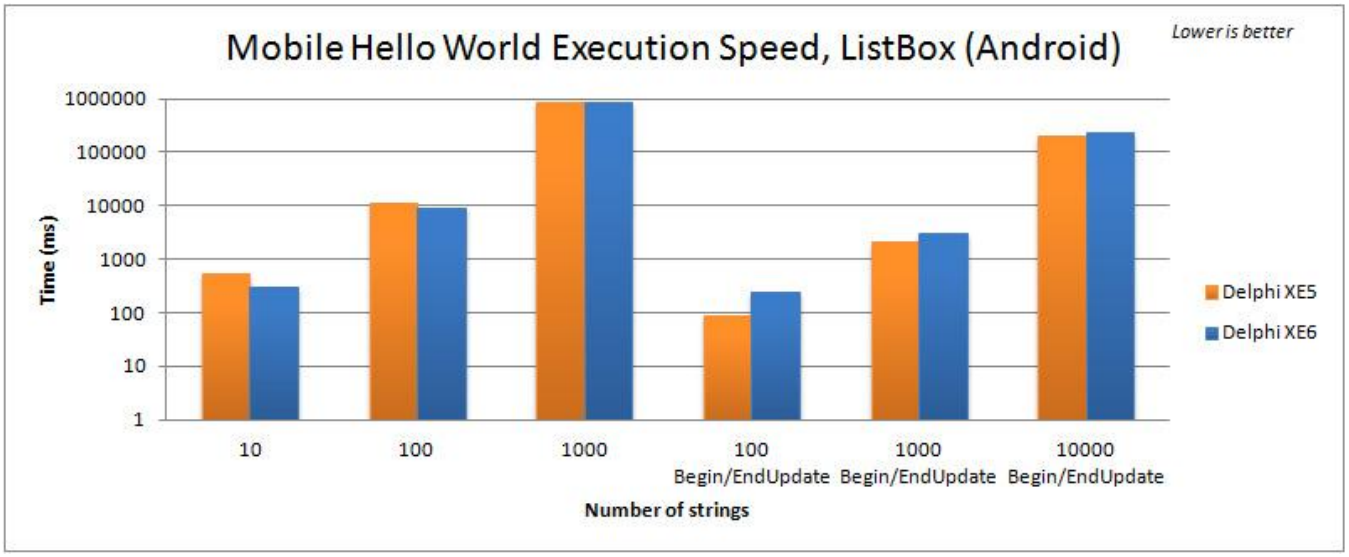


Figure 41 Comparison of execution speed for filling a TListBox in the Mobile FMX Hello World (Android) between Delphi XE5 and XE6

No clear winner emerges for the Android platform, though Delphi XE6 is the all-around better Delphi for Android. If you look at the filling TListBox chart, Delphi XE6 is faster at filling a TListBox with less than 1000 items and no BeginUpdate/EndUpdate. However, when the BeginUpdate and EndUpdate methods are called, Delphi XE5 is faster (157 ms at 100 items, 941 ms at 1000 items, and **41 seconds** at 10000 items.

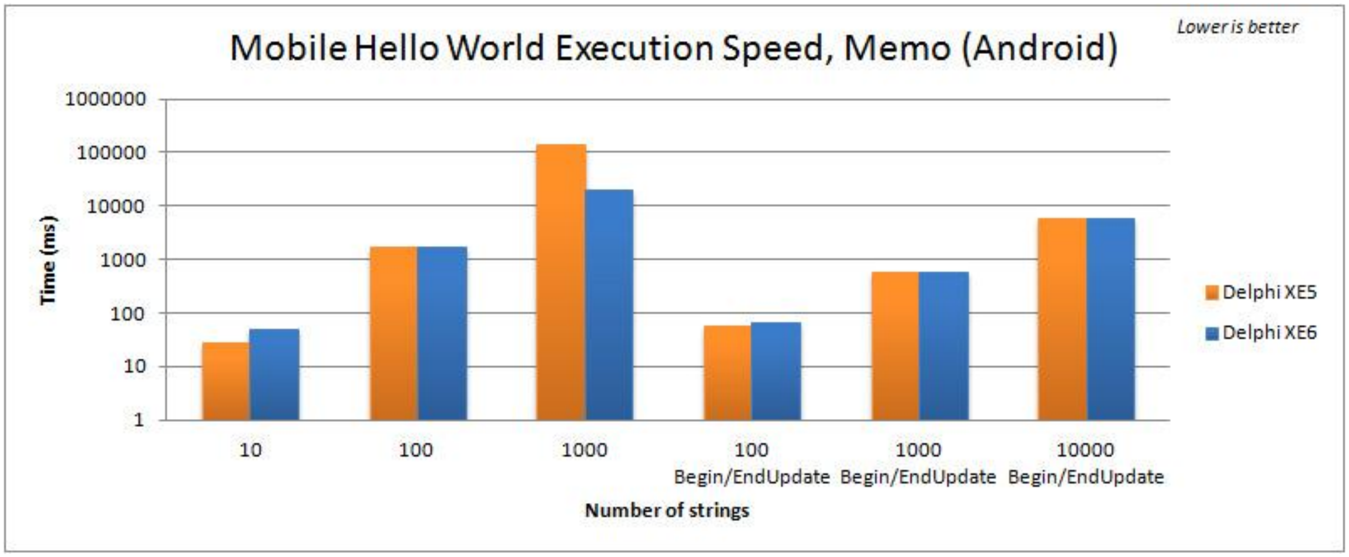


Figure 42 Comparison of execution speed for filling a TMemo in the Mobile FMX Hello World (Android) between Delphi XE5 and XE6

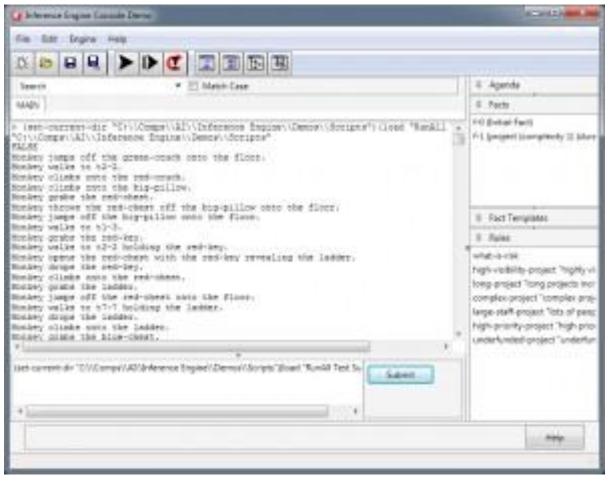
When filling a TMemo, Delphi XE5 and Delphi XE6 are essentially tied except for one important caveat. Delphi XE5 is almost twice as fast as XE6 at adding 10 lines (and no BeginUpdate/EndUpdate); however, that does not mean much as it is only a difference of 22 ms. **The glaring discrepancy is that Delphi XE6 is over 7x faster than XE5 in filling a TMemo with 1000 lines (and no BeginUpdate/EndUpdate). It takes 20 seconds for XE6; over 2 minutes later XE5 finishes.**

IECS Speed Tests

So far, we have been testing the various Delphi versions using a simple Hello World application (described in the Introduction). The Hello World project is a simple application/app that provided some interesting comparisons between the different Delphi versions on various platforms. It showed the work that needs to be done to improve the FMX versions. However, the Hello World project is limited and truly only tested a couple "standard" controls (TListBox and TMemo). This week, we are going to show speed results from sample applications using the Inference Engine Component Suite (IECS). The IECS is a large component library (96K lines of engine code and another 48K LOC for dialogs) for developing expert systems with Delphi. It uses lots of interfaces, generics, parsing/string manipulation, and 100s of classes. It works from Delphi 2010 through XE6 and for all platforms that Delphi supports. It should provide very interesting **non-visual** execution results.

About the IECS Test Applications

The IECS test applications use the IECS components to solve expert systems, both with Fuzzy Logic and without. We are actually testing 3 demo projects that come with the IECS: IECS Advanced Console application (both VCL and FMX), IECS Basic Console application (VCL and FMX), and an IECS Mobile app (FMX for iOS and Android).



```
> (set-current-dir "C:\\Comps\\AI\\Inference Engine\\Demos\\Scripts")(load "RunAll
"C:\\Comps\\AI\\Inference Engine\\Demos\\Scripts"
FALSE
Monkey jumps off the green-couch onto the floor.
Monkey walks to t2-2.
Monkey climbs onto the red-couch.
Monkey climbs onto the big-pillow.
Monkey grabs the red-chest.
Monkey throws the red-chest off the big-pillow onto the floor.
Monkey jumps off the big-pillow onto the floor.
Monkey walks to t1-3.
Monkey grabs the red-key.
Monkey walks to t2-2 holding the red-key.
Monkey opens the red-chest with the red-key revealing the ladder.
Monkey drops the red-key.
Monkey climbs onto the red-chest.
Monkey grabs the ladder.
Monkey jumps off the red-chest onto the floor.
Monkey walks to t7-7 holding the ladder.
Monkey drops the ladder.
Monkey climbs onto the ladder.
Monkey grabs the blue-chest.
```

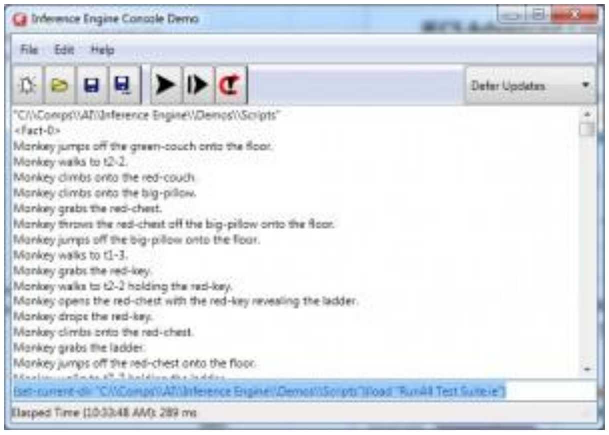
```
f-0 (initial-fact)
f-1 (project (complexity 1) (dura
```

```
what-is-risk
high-visibility-project "highly vi
long-project "long projects inc
complex-project "complex proj
large-staff-project "lots of peo
high-priority-project "high prio
underfunded-project "underfun
```

```
(set-current-dir "C:\\Comps\\AI\\Inference Engine\\Demos\\Scripts")(load "RunAll Test Su
```

Figure 43 Inference Engine Component Suite Advanced Console Screenshot

The **IECS Advanced Console Application** provides a GUI console for interacting and building expert systems. It contains input and output windows and listboxes that display the list of facts, fact templates, rules, and agenda items currently active in the inference engine. This application should provide a good mix of non-visual benchmarks with updates to TListBoxes and output to a TMemo and tests Win32, Win64, and OSX. *Note that this is not a console program in the Delphi sense, but a console for interacting with the expert system.* The **IECS Basic Console Application** provides a very simple GUI console for interacting and building expert systems. There is an input TEdit and an output TMemo, but no TListBox. The output can be controlled from a combobox: Live Updates print all output immediately as it occurs, Defer Updates defers all output until the current command is finished by surrounding output in BeginUpdate/EndUpdate, and No Updates suppresses all output and will just write execution time to a status bar.



```
"C:\\Comps\\AI\\Inference Engine\\Demos\\Scripts"
<Fact-0>
Monkey jumps off the green-couch onto the floor.
Monkey walks to t2-2.
Monkey climbs onto the red-couch.
Monkey climbs onto the big-pillow.
Monkey grabs the red-chest.
Monkey throws the red-chest off the big-pillow onto the floor.
Monkey jumps off the big-pillow onto the floor.
Monkey walks to t1-3.
Monkey grabs the red-key.
Monkey walks to t2-2 holding the red-key.
Monkey opens the red-chest with the red-key revealing the ladder.
Monkey drops the red-key.
Monkey climbs onto the red-chest.
Monkey grabs the ladder.
Monkey jumps off the red-chest onto the floor.
Monkey walks to t7-7 holding the ladder.
```

```
(set-current-dir "C:\\Comps\\AI\\Inference Engine\\Demos\\Scripts")(load "RunAll Test Suite.ie")
```

```
Elapsed Time (10:33:48 AM): 289 ms
```

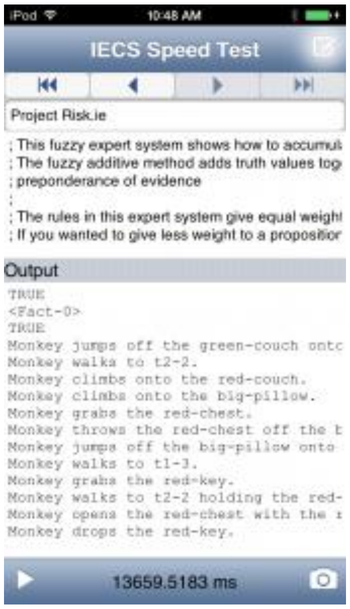
Figure 44 Inference Engine Component Suite Basic Console Screenshot

This application is about getting the GUI out of the way and benchmarking non-visual performance with minimal GUI updates. However, as we will see, even little output to a TMemo wrapped in BeginUpdate/EndUpdate calls can have significant performance penalties, especially in FMX. This application is for Win32, Win64, and OSX. *Note that this is not a console program in the Delphi sense, but a console for interacting with the expert system.* The **IECS Mobile app** provides the simple GUI console for executing expert systems on iOS and Android. This app executes expert systems and can optionally send output to a TMemo.

Upon execution, all 3 applications execute a battery of expert system problems:

- Monkey and Banana Problem - Determine steps for monkey to eat a banana based on moving around a room, stacking furniture, etc to get to banana
- Word Game problem (GERALD + DONALD = ROBERT)
- Stacking Problem - Determine plan for stacking one block onto another
- Sticks Game - Win the game by removing 1 to 3 sticks per turn (from a starting pile of sticks) to leave opposing player with 1 stick at end (only in desktop simulations)
- Home Recommendation - return scores for different cities based on customer choices for average temperature, population, etc using fuzzy logic
- Product Design Match - returns scores for different products based on customer choices for cost and size using fuzzy logic
- Backorders - Simple calculation of backorders based on fuzzy customer order amounts (low, high, medium)
- Game Combs - Decide aggressiveness based on enemy health, our health, and distance
- Project Risk - Determine how risky a project is based on accumulation of evidence of duration, staffing, funding, complexity, priority, and visibility

  using fuzzy logic



Project Risk.ie

; This fuzzy expert system shows how to accumul
; The fuzzy additive method adds truth values tog
; preponderance of evidence
;
; The rules in this expert system give equal weight
; If you wanted to give less weight to a propositior

```
TRUE
<Fact-0>
TRUE
Monkey jumps off the green-couch ontc
Monkey walks to t2-2.
Monkey climbs onto the red-couch.
Monkey climbs onto the big-pillow.
Monkey grabs the red-chest.
Monkey throws the red-chest off the t
Monkey jumps off the big-pillow onto
Monkey walks to t1-3.
Monkey grabs the red-key.
Monkey walks to t2-2 holding the red-
Monkey opens the red-chest with the r
Monkey drops the red-key.
```

13659.5183 ms

Figure 45 Inference Engine Component Suite Mobile App Screenshot

The IECS Advanced and Basic Console applications load the expert systems from the file system. The IECS Mobile App has the expert systems embedded in the app in a TClientDataSet.

IECS Advanced Console Speed Tests

The first Inference Engine Component Suite (IECS) application we will start with is the IECS Advanced Console application. As mentioned, this application should provide a good mix of non-visual benchmarks with updates to TListBoxes and output to a TMemo and tests Win32, Win64, and OSX.

IECS Advanced Console, VCL, Win32

The IECS Advanced Console Win32 VCL speed tests closely mirror the Hello World VCL Win32 speed results. All versions complete the tests in around 1.2 seconds with only 0.03 second difference between the winner (Delphi XE) and the loser (Delphi 2010). The differences are essentially meaningless. According to these test results, no significant compiler works has probably occurred between Delphi 2010 and XE6 (at least for Win32).

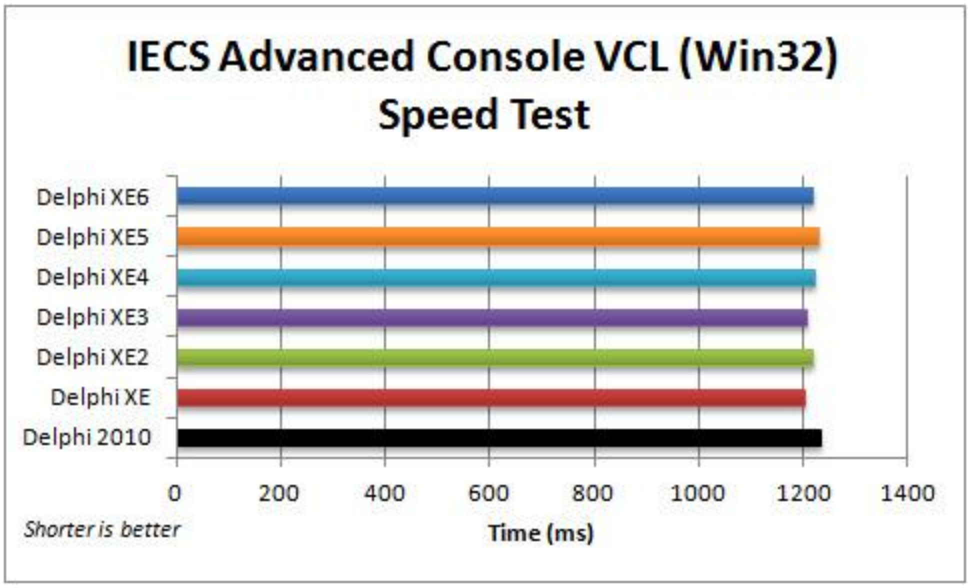


Figure 46 Comparison of execution speed for executing expert systems in the VCL IECS Advanced Console application (Win32) with Delphi 2010 to XE6

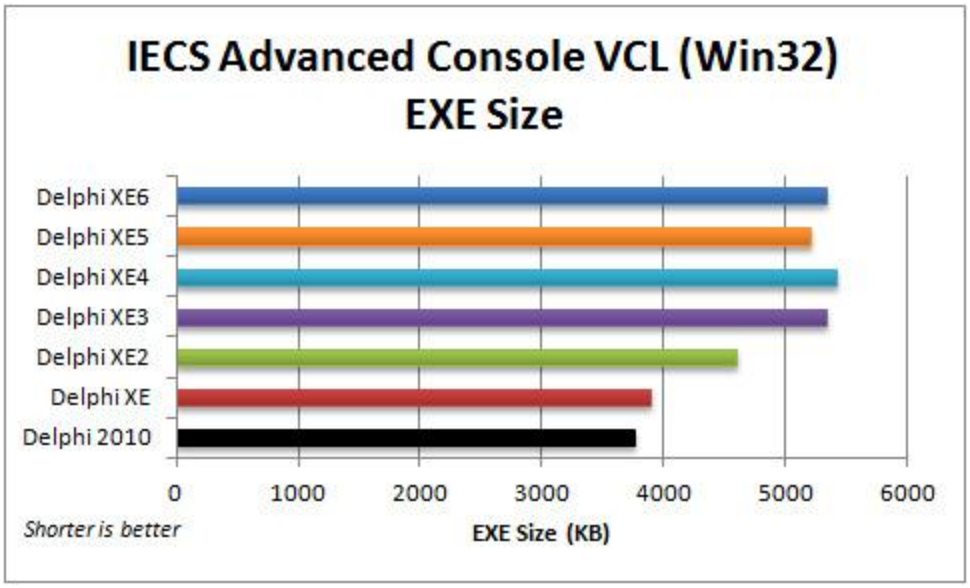


Figure 47 Comparison of EXE size for the VCL IECS Advanced Console application (Win32) with Delphi 2010 to XE6

Of more interest, the IECS application EXE sizes show the same ballooning of application size as the Hello World application.

IECS Advanced Console, FMX, Win32

Since the Win32 VCL results were so similar, we decided to skip doing Win64 VCL results and go directly to examining IECS Advanced Console Win32 FMX speed tests. With the IECS Advanced Console running all of those expert system consecutively, we would expect the Delphi versions that did well with small number of items in TListBox (e.g., XE2 and XE3) and with a moderate number of lines added to a TMemo without BeginUpdate/EndUpdate calls (e.g., XE4 and XE6) should do best. Since there are 4 listboxes (one each for Agenda, Facts, Fact Templates, and Rules) versus 1 memo, XE2 and XE3 should be the winner. This is what we see, **Delphi XE3 is fastest. Delphi XE2 is hurt by its poor TMemo performance and almost loses second place to Delphi XE6**. **Delphi XE5 is the clear loser with its abysmal TMemo performance making it 2.5x slower than XE3.**

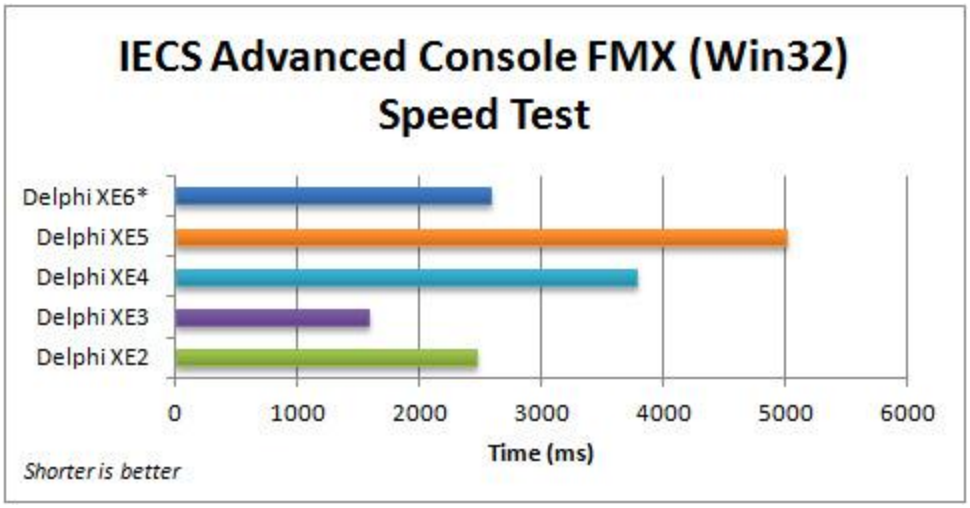


Figure 48 Comparison of execution speed for the FMX IECS Advanced Console application (Win32) with Delphi XE2 to XE6

**Note the Delphi XE6 is "helped" because we had to disable the Memo1.GoToTextEnd statement after every addition to the TMemo, which causes a guaranteed access violation in XE6 (without updates). This operation incurs some cost and by not having it, Delphi XE6 manages to beat XE4.*

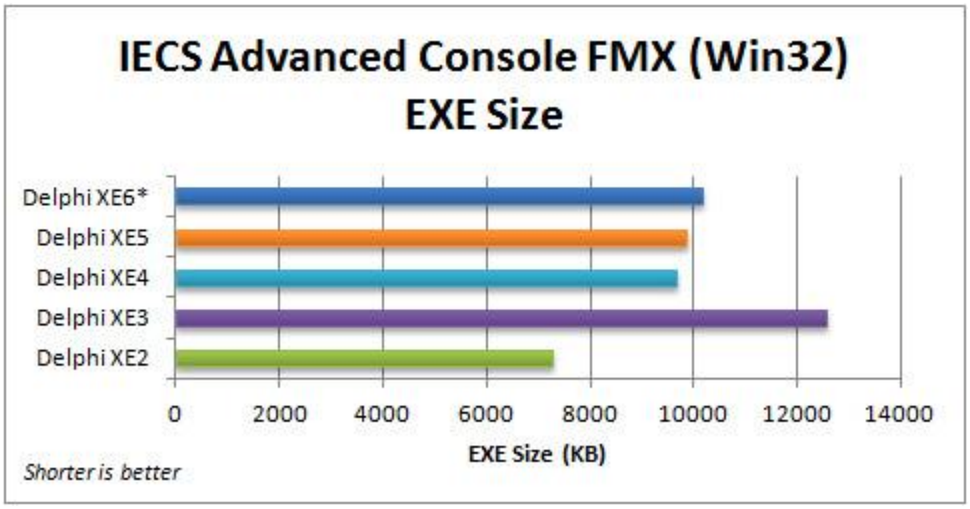


Figure 49 Comparison of EXE Sizes for the FMX IECS Advanced Console application (Win32) with Delphi XE2 to XE6

The EXE size closely tracks Hello World FMX Win32 sizes, with every version getting bigger except for the oversized EXE sizes produced by Delphi XE3. These differences hold true for all applications. **For the rest of this document, we are not going to compare EXE size anymore.**

IECS Advanced Console, FMX, Win64

The same pattern of results holds for the 64-bit version of the IECS Advanced Console application running on a Windows 7 64-bit box. **Delphi XE3 is fastest. Delphi XE2 is hurt by its poor TMemo performance and**

almost loses second place to Delphi XE6. **Delphi XE5 is the clear loser with its abysmal TMemo performance making it 2.5x slower than XE3. If we compare the ratio of Win64 execution speed to Win32 execution speed, we see that Delphi XE2, XE5 and XE6 all improve their speed in Win64.** For some reason, Delphi XE3 and XE4 are slower in Win64 than their Win32 counterparts.

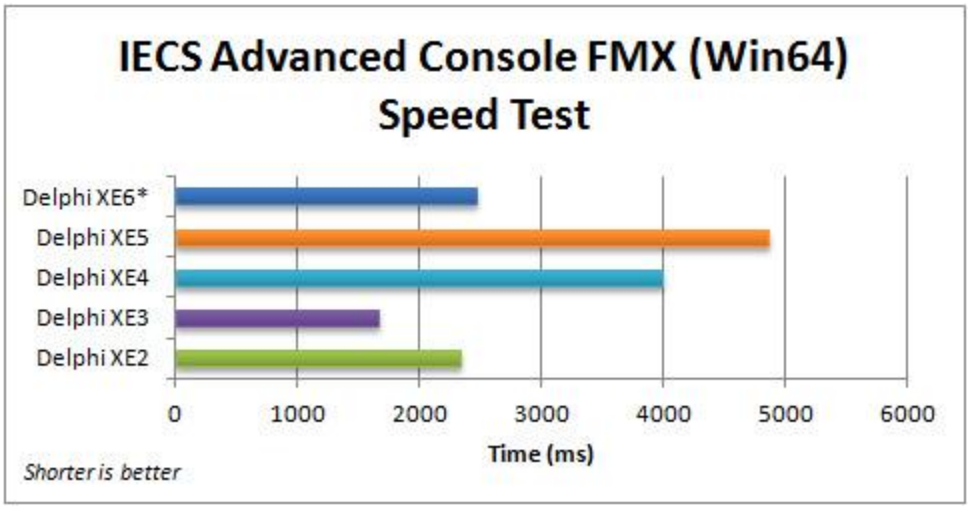


Figure 50 Comparison of execution speed for the FMX IECS Advanced Console application (Win64) with Delphi XE2 to XE6

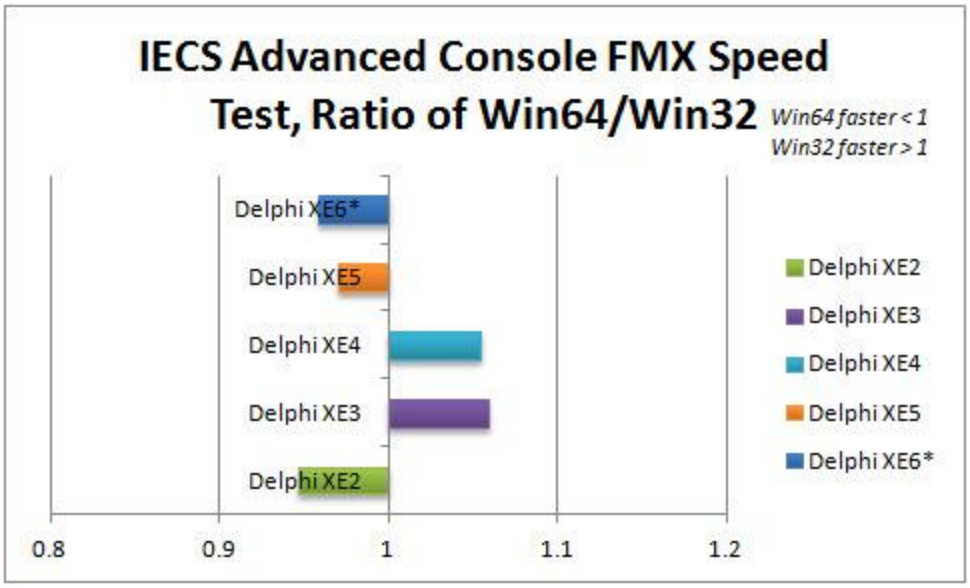


Figure 51 Ratio of Win64 to Win32 execution speed for IECS Advanced Console in Delphi XE2 to XE6 (FMX)

IECS Advanced Console, FMX, OSX

Interestingly, when we test the IECS Advanced Console on OSX, the pattern is broken. **Delphi XE3 is still fastest. However, Delphi XE6 is now second and XE4 third.**

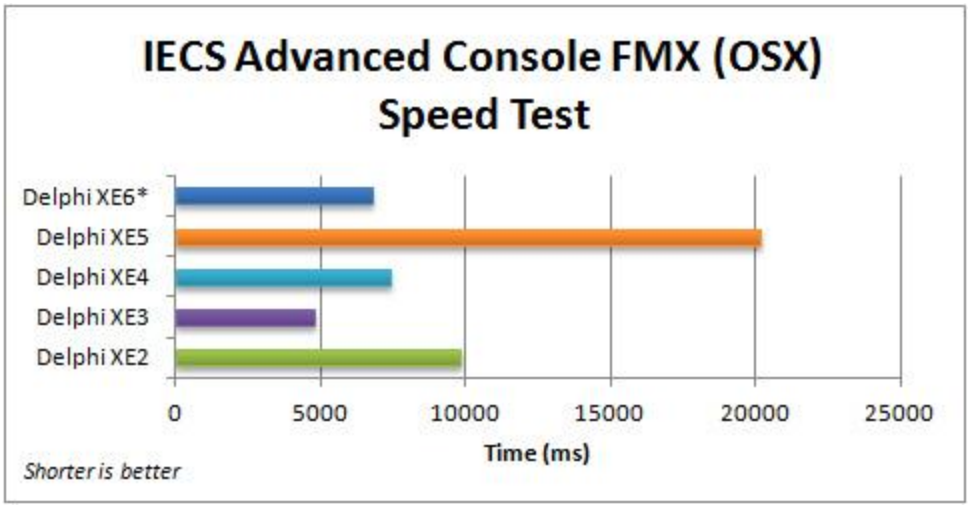


Figure 52 Comparison of execution speed for the FMX IECS Advanced Console application (OSX) with Delphi XE2 to XE6

As we mentioned previously, the Windows machine used for testing is not exactly comparable to the Mac Mini. However, the Windows Machine specs (Windows 7 64-bit Intel I7 930 @ 2.8 GHz CPU, and 6 GB RAM) are *close* to the Mac Mini specs (OSX 64-bit Intel I7 @ 2.3 GHz with 4GB RAM), so it will be interesting to do comparisons of the FMX Win32 vs FMX OSX IECS.

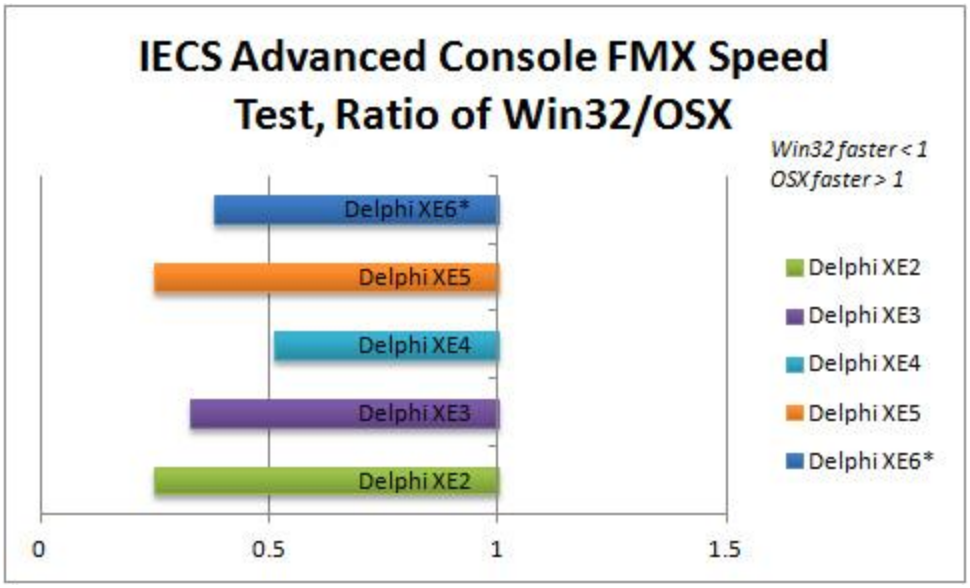


Figure 53 Ratio of Win32 to OSX execution speed for IECS Advanced Console in Delphi XE2 to XE6 (FMX)

The Win32 versions absolutely destroy the OSX versions and are 2x-4x faster. The poor TMemo performance on OSX significantly hurts the OSX applications. On the next page, we examine the IECS Basic Console speed tests...

IECS Basic Console Speed Tests

The second IECS test application is the IECS Basic Console application. As mentioned in the IECS description, the output in the Basic Console can be controlled from a combobox: *Live Updates print all output immediately as it occurs, Defer Updates defers all output until the current command is finished by surrounding* *output in BeginUpdate/EndUpdate, and No Updates suppresses all output and will just write execution time to a status bar.* This application is about getting the GUI out of the way and benchmarking non-visual performance with minimal GUI updates. However, as we will see, even a little output to a TMemo wrapped in BeginUpdate/EndUpdate calls can have significant performance penalties, especially in FMX.

IECS Basic Console, VCL, Win32

As expected, our first tests in Win32 VCL revealed that No Updates executions are fastest, deferring updates are next, and live updates are the slowest. However, with the VCL TMemo, the differences are relatively minor with no updates being only about twice as fast. **There is one surprise though: Delphi XE6 is slightly but consistently faster than all other versions of Delphi tested.** There is a 16-27 ms difference between Delphi XE6 (fastest) and Delphi 2010 (slowest). Since the IECS uses generics extensively and generics were still relatively immature in Delphi 2010 (only being introduced in Delphi 2009), perhaps this explains the differences we see.

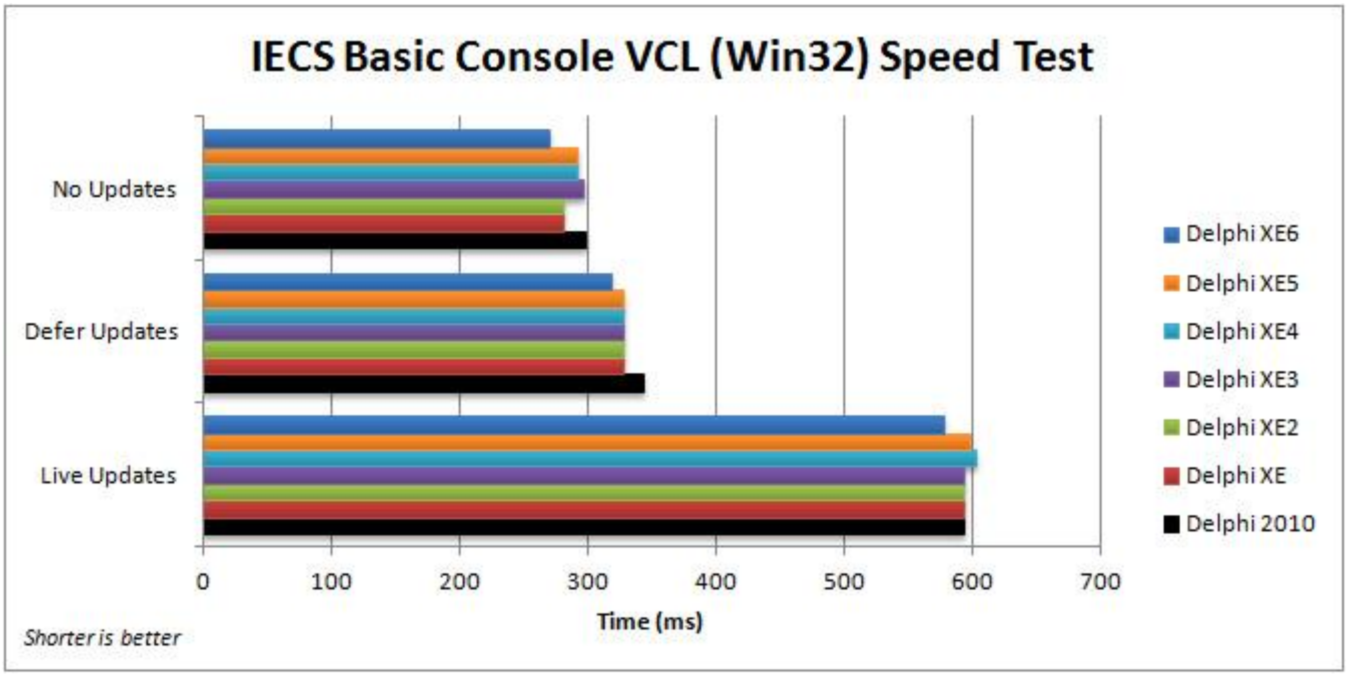


Figure 54 Comparison of execution speed for executing expert systems in the VCL IECS Basic Console application (Win32) with Delphi 2010 to XE6

IECS Basic Console, VCL, Win64

When we move up to 64 bits, **Delphi XE6 keeps its lead and is faster than all other Delphi versions in all modes (except for a tie with Delphi XE2 when deferring updates).**

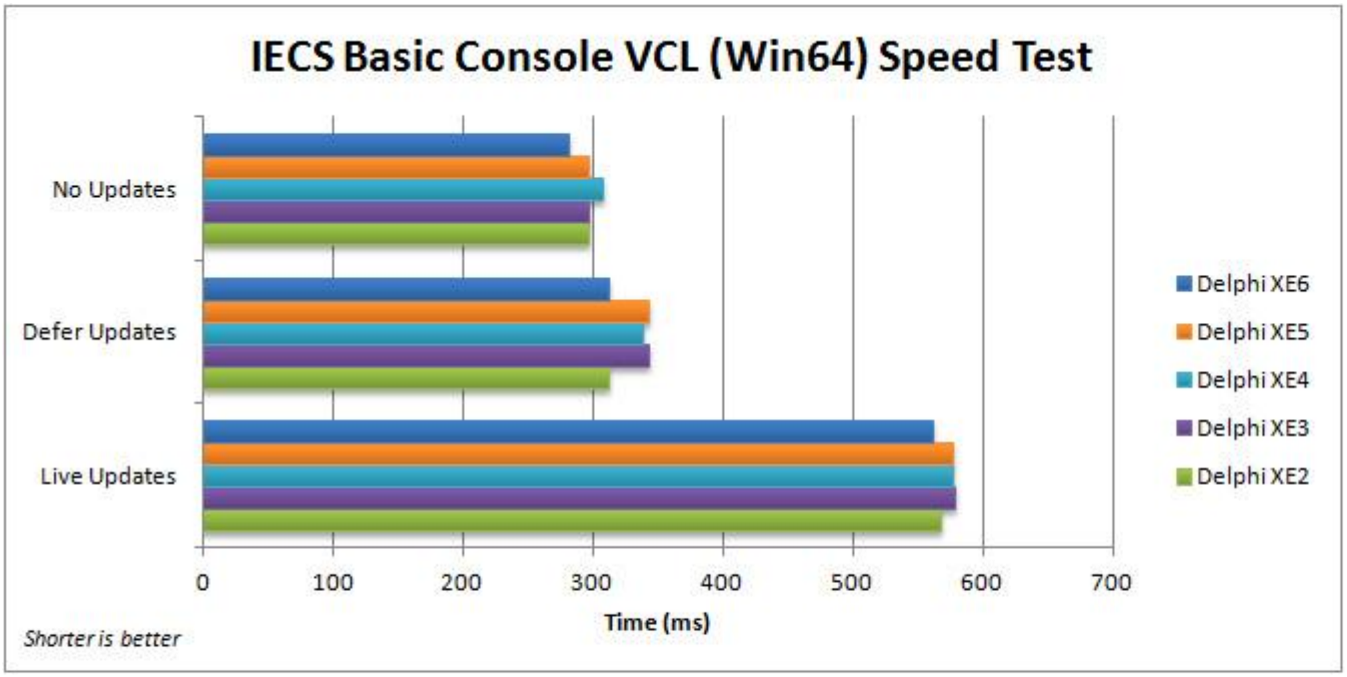


Figure 55 Comparison of execution speed for executing expert systems in the VCL IECS Basic Console application (Win64) with Delphi 2010 to XE6

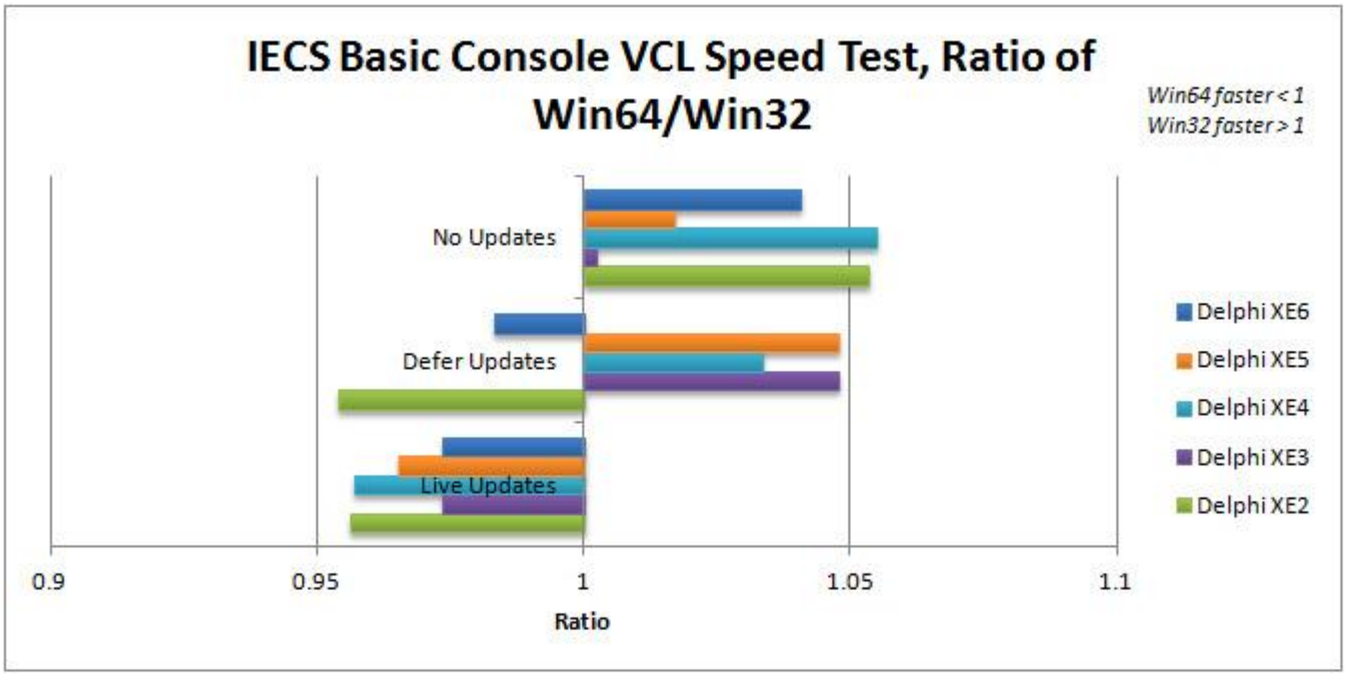


Figure 56 Ratio of Win64 to Win32 execution speed for IECS Basic Console in Delphi XE2 to XE6 (VCL)

Examining the ratio of Win64/Win32 doesn't reveal much. Win64 is marginally faster than Win32 with live updates and marginally slower with no updates.

IECS Basic Console, FMX, Win32

With the FMX version of the IECS Basic Console, things get more interesting. In Win32, t**he Delphi XE4 version scales the best from no updates to defer updates to live updates. Delphi XE6 has a strong showing being second fastest (only 6 ms behind XE2) with no updates, fastest with defer updates, and second fastest with live updates.** As usual, Delphi XE5 does well if we don't have to actually output anything. As soon as live updates are turned on, Delphi XE4 is over 4x faster.

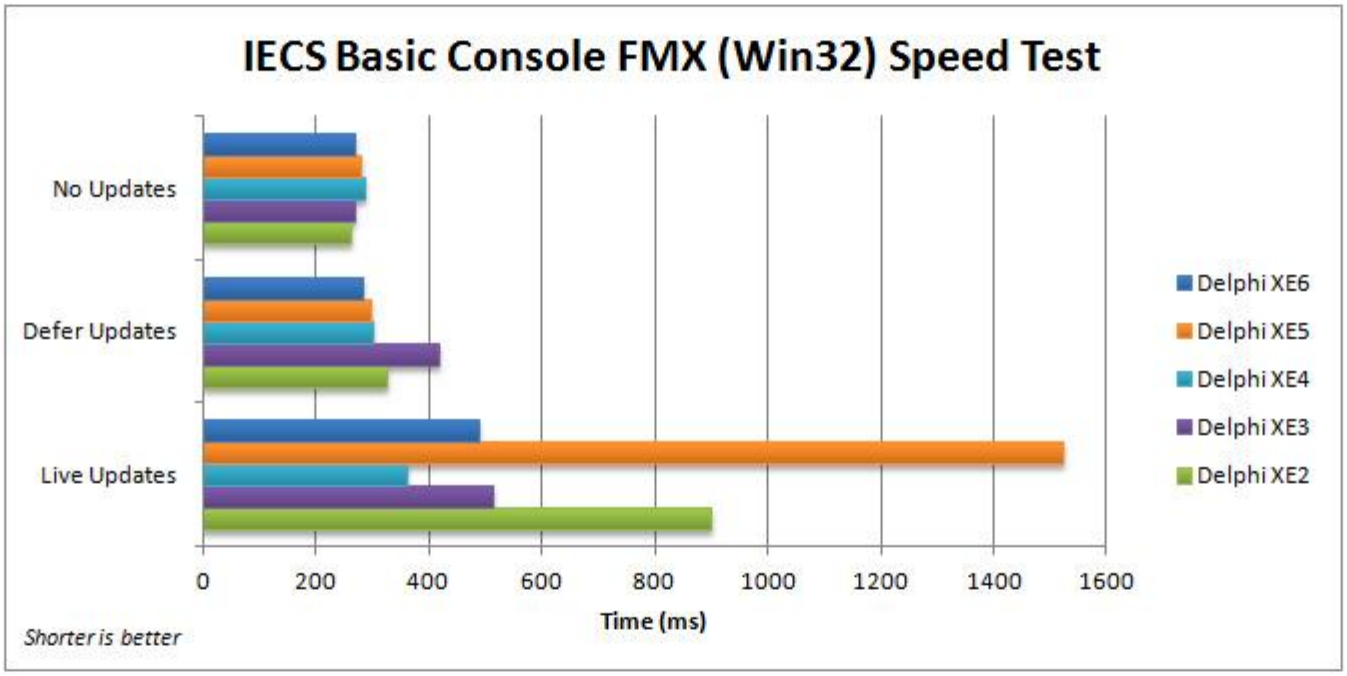


Figure 57 Comparison of execution speed for the FMX IECS Basic Console application (Win32) with Delphi XE2 to XE6

IECS Basic Console, FMX, Win64

Moving up to 64-bit FMX, Delphi XE6 continues to do well. It is only 2.6 ms behind XE2 (within error margins) when there are no updates, a solid 23 ms ahead when deferring updates, and second fastest (101 ms behind) when live updates are on. Delphi XE4 is commanding when live updates are on and very strong otherwise. Delphi XE5 is dragged down by its slow TMemo output.

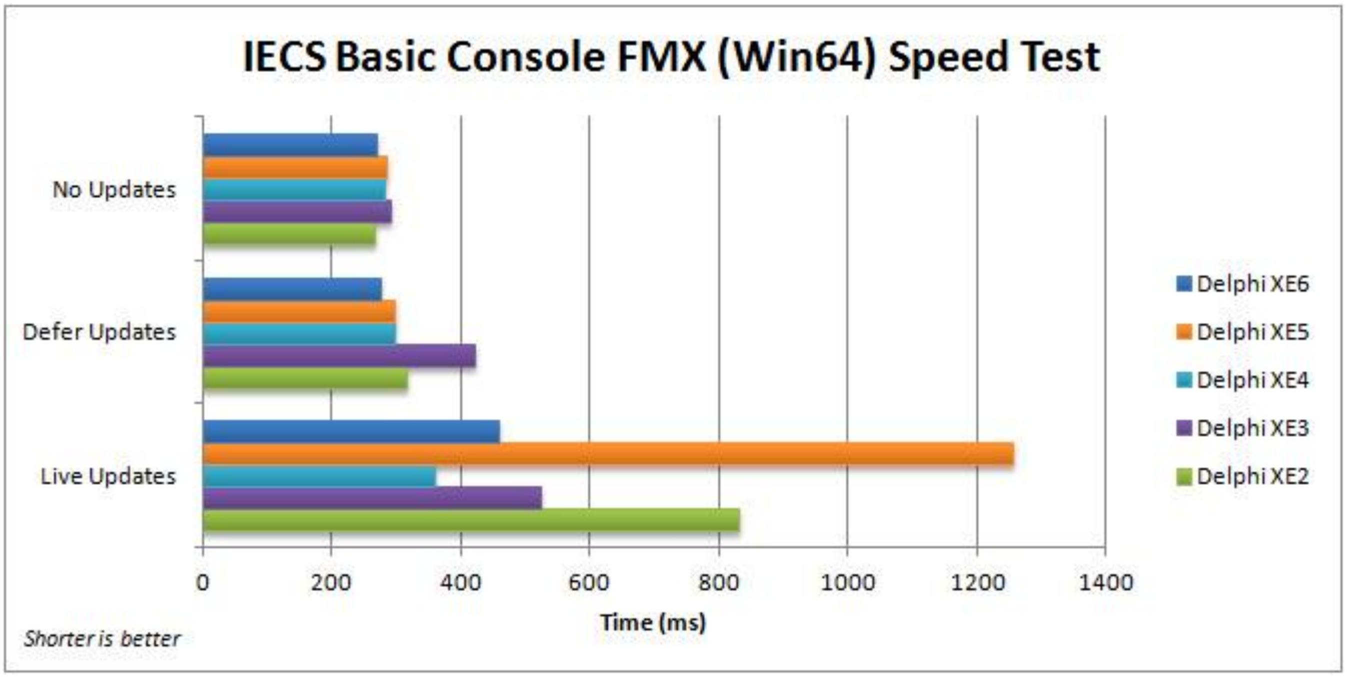


Figure 58 Comparison of execution speed for the FMX IECS Basic Console application (Win64) with Delphi XE2 to XE6

Looking at the ratio of Win64/Win32, there are no clear patterns shown. Delphi XE3 is always faster in Win32 versus Win64. Both Delphi XE2 and XE6 manage to gain speed when outputting to the memo, whether live or deferred. Delphi XE5 manages to do significantly better in Win64 when doing live updates to the memo.

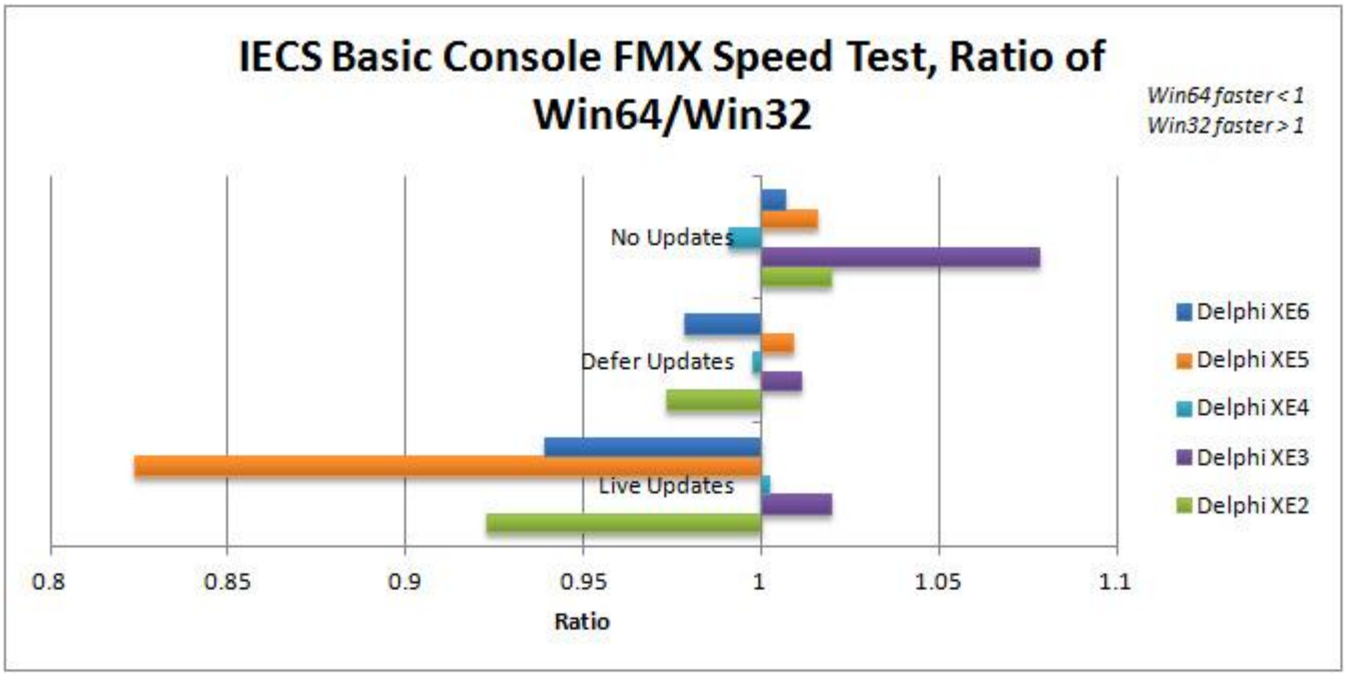


Figure 59 Ratio of Win64 to Win32 execution speed for IECS Basic Console in Delphi XE2 to XE6 (FMX)

IECS Basic Console, FMX, OSX

Moving to the Mac, D**elphi XE4 is our winner, managing strong showings in all modes.** Without updates all versions of Delphi are essentially tied (XE6 ekes out 4.5 ms victory). When we start deferring updates, XE2 and XE3 fall behind. The surprising result is that when we turn on live updates, Delphi XE2 performs as poorly as XE5.

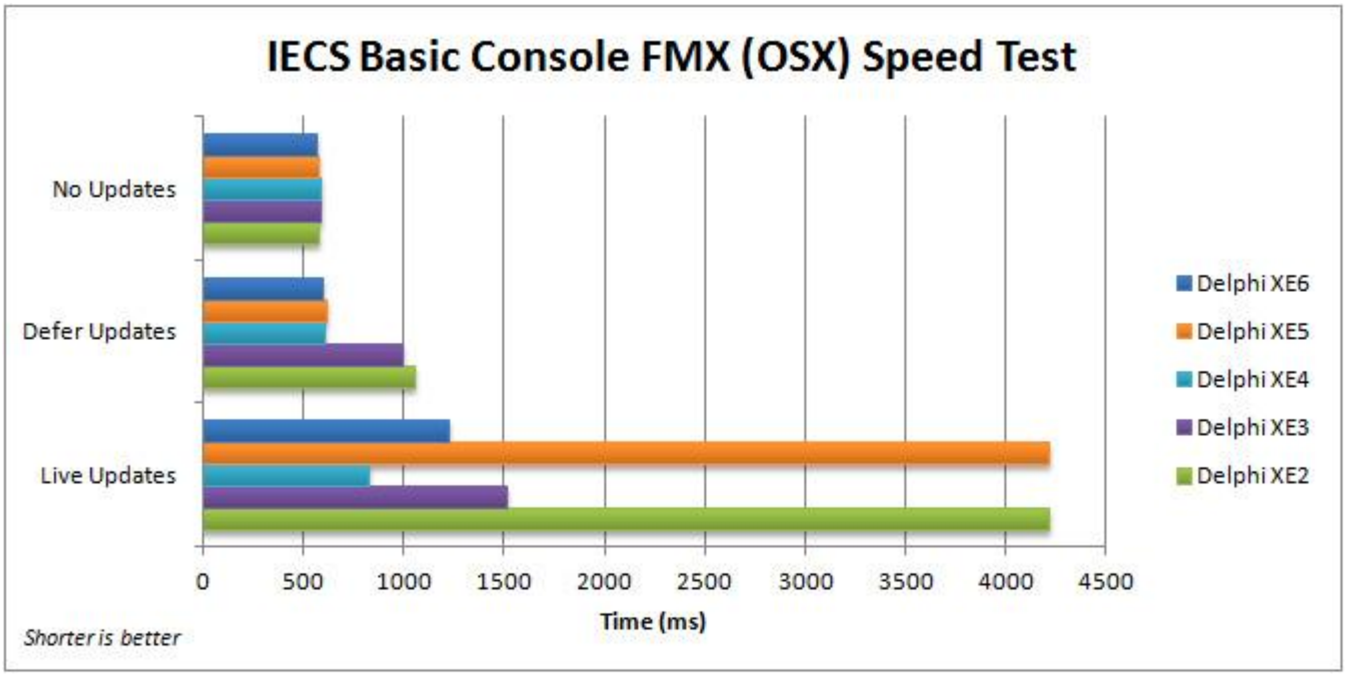


Figure 60 Comparison of execution speed for the FMX IECS Basic Console application (OSX) with Delphi XE2 to XE6

Like with the IECS Advanced Console, **when we look at the ratio of Win32/OSX, the Win32 versions absolutely destroy the OSX versions and are 2x-5x faster.** Delphi XE2 shows the largest difference. Delphi XE4 manages the "best" parity between the Win32 and OSX version, with the Win32 version being "only" a little more than twice the speed of the OSX version.

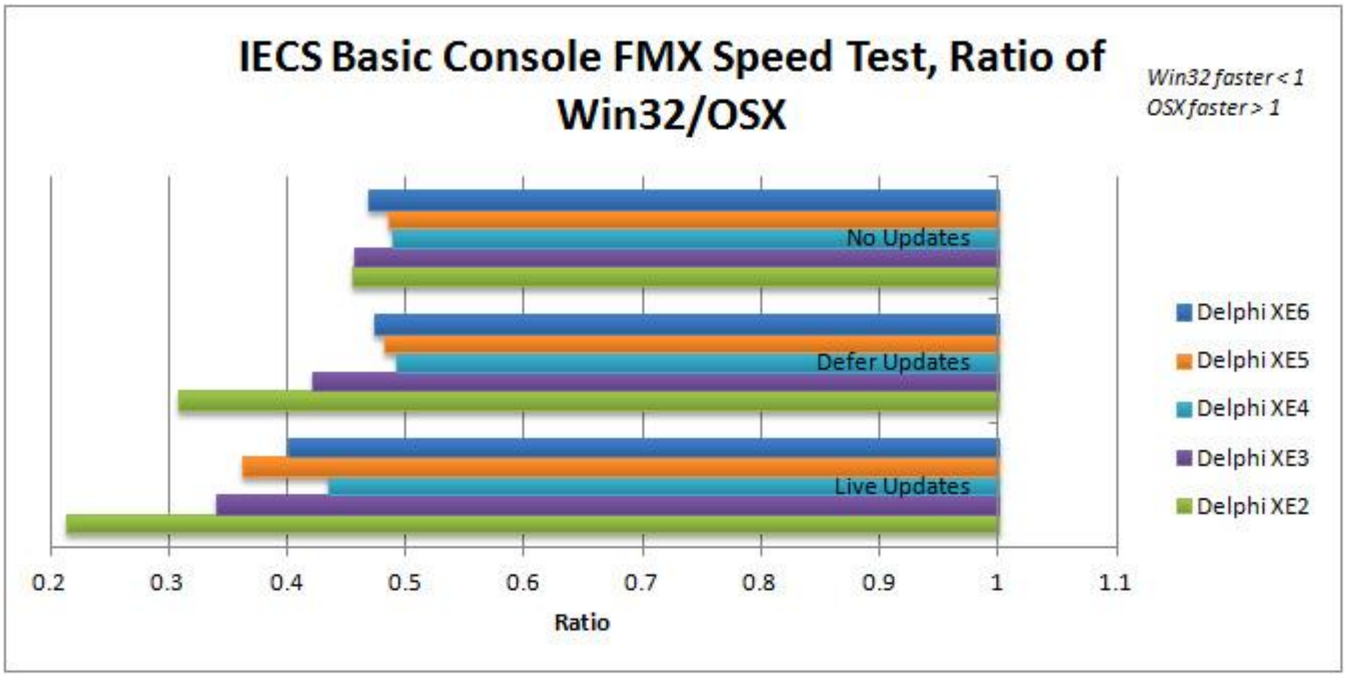


Figure 61 Ratio of Win32 to OSX execution speed for IECS Basic Console in Delphi XE2 to XE6 (FMX)

IECS Mobile App Speed Tests

In the previous two subsections, we looked at the Inference Engine Component Suite (IECS) Advanced Console speed tests and the IECS Basic Console speed tests on Win32, Win64, and OSX. Our final IECS Speed tests are with the IECS Mobile app running on iOS and Android. The **IECS Mobile app** provides the simple GUI console for executing expert systems on iOS and Android. This app executes expert systems and can optionally send output to a TMemo. It is much closer to the Basic Console than to the Advanced Console so we expect to see similar performance patterns to the IECS Basic Console.

IECS Mobile App, FMX, iOS

Since Delphi XE4, Delphi has supported deploying to iOS devices. Note that the Delphi XE4 app was compiled with **debug**. Delphi XE5 and XE6 can deploy **release** versions directly to the device without requiring certificates. However, this was not true with Delphi XE4 but as we will see, it doesn't seem to have slowed the app down. As with the Hello World Mobile App, we tested our mobile app on an iPod Touch.

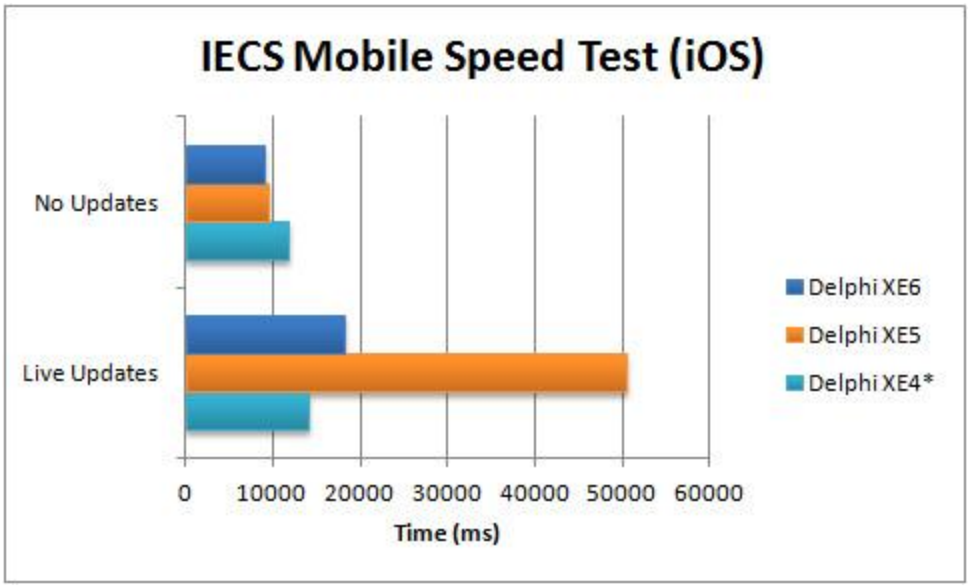


Figure 62 Comparison of execution speed for IECS Mobile App (iOS) from Delphi XE4 to XE6

Delphi XE6 makes a very good showing on iOS. It manages the fastest non-visual speed (no updates), being a 1/4 second faster than XE5 and over 2.5 seconds faster than XE4. With live updates turned on, Delphi XE6 puts in a respectable second place finish. Delphi XE4 dominates with live updates and finishes in 14.1 seconds compared to 18.2 seconds with XE6 and compared to the **embarrassingly awful 50.3 seconds with XE5**.

IECS Mobile App, FMX, Android

Since Delphi XE5, Delphi has supported deploying to Android devices. In Android, Delphi XE5 has to plead no contest again. While Delphi XE6 is only approximately 9% faster with no updates, once the TMemo is turned on, Delphi XE6 is twice as fast as Delphi XE5.

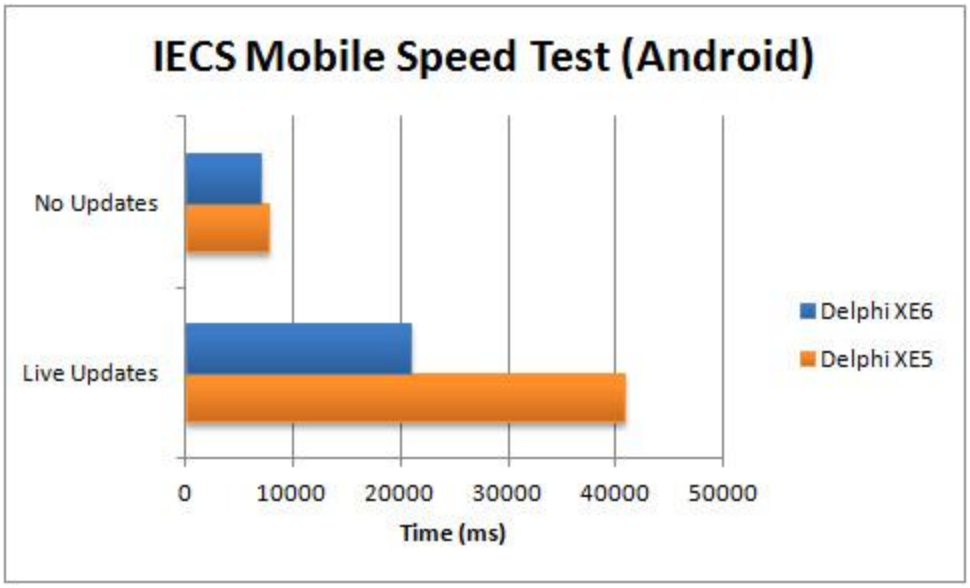
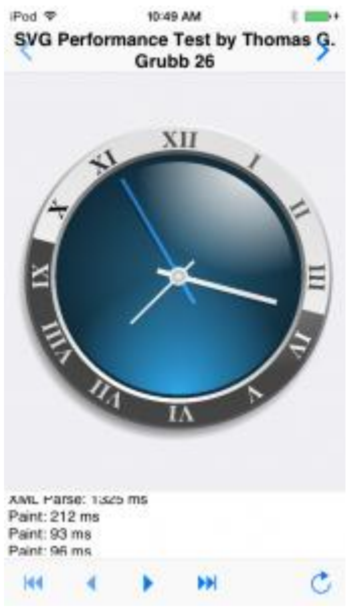


Figure 63 Comparison of execution speed for IECS Mobile App (Android) between Delphi XE5 and XE6

Well, that is it for the IECS test applications. The three IECS test applications showed the strength of Delphi XE4 and Delphi XE6. Next, we are going to perform speed tests using the RiverSoftAVG SVG Component Library (RSCL). The RSCL will **only test Delphi XE2 through XE6 and will concentrate on FMX**. However, it renders SVG files using low-level canvas operations (gradients, paths, text, etc) and is available for VCL and FMX. It uses GDI+ for VCL (which is a software renderer with comparable features to the FMX TCanvas) so it should provide a **good test of the FMX DirectX and OpenGL hardware-based rendering.** The RSCL is also able to build SVGs using the FMX shape primitives (TRectangle, TPath, TText, etc); it should reveal any optimizations (or lack thereof) Embarcadero has been doing with the **low-level primitives from which most FMX controls are built.**

RiverSoftAVG SVG Component Library Speed Tests



```
XML Parse: 1325 ms
Paint: 212 ms
Paint: 93 ms
Paint: 96 ms
```

Figure 64 Display of the SVG clock using FMX TCanvas operations in the iOS test app

In the previous section, we tested the various Delphi versions with sample applications using the Inference Engine Component Suite (IECS). It provided interesting details for non-visual performance as well as a mix of non-visual and visual performance. Our final tests are going to use the RiverSoftAVG SVG Component Library (RSCL) to test the graphics performance of the FMX Delphi versions, e.g., **Delphi XE2 through XE6**. The FMX TCanvas is graphics hardware accelerated, using DirectX on Windows and OpenGL on OSX, iOS, and Android. In addition, Delphi XE6 added DirectX 11 support so it will be interesting to see if that makes a difference.

About the RSCL Test Applications

There are actually two types of RSCL applications we are going to test. The first type of RSCL applications will test the drawing speed of the FMX TCanvas routines by drawing SVGs. They render SVG files using low-level canvas operations (gradients, paths, text, etc) and are available for VCL and FMX. The VCL versions use GDI+ (which is a software renderer with comparable features to the FMX TCanvas) so they should provide a **good baseline of the FMX DirectX and OpenGL hardware-based rendering.**

Figure 65 Display of the SVG watch using FMX TCanvas operations in the iOS test app

The RSCL is also able to build SVGs using the FMX shape primitives (TRectangle, TPath, TText, etc); it should reveal any optimizations (or lack thereof) Embarcadero has been doing with the **low-level primitives from which most FMX controls are built.**

For simplicity with deploying mobile apps, all of the projects store the test SVGs inside a TClientDataSet in the app/application. The SVG applications are only testing **drawing** using low-level canvas operations or shape primitives; the loading, parsing, and building of the RSCL SVG elements and primitives are not tested. There are four SVGs used in the tests:

1. Clock
2. Car
3. Watch
4. Flag

The following table breaks down the SVG elements that make up each SVG:

```
Paint: 715 ms
Paint: 795 ms
Paint: 713 ms
```

Figure 67 Display of the SVG car using FMX TCanvas operations in the iOS test app

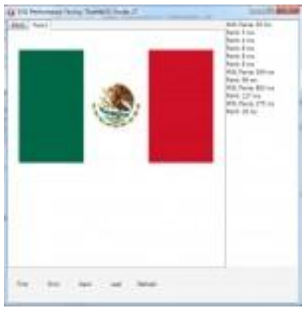
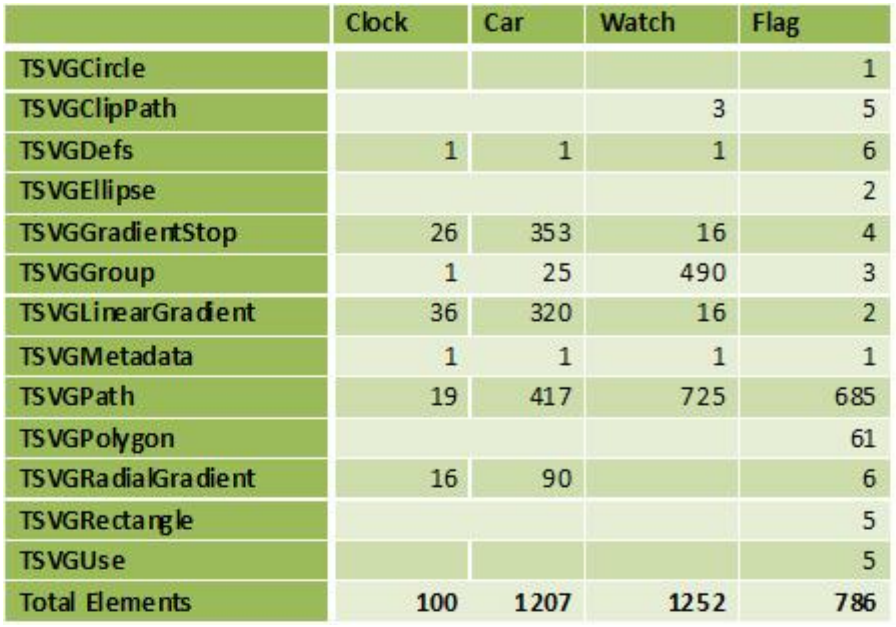


Figure 66 Display of the SVG flag using FMX shape primitives in the iOS test app



	Clock	Car	Watch	Flag
TSVGCircle				1
TSVGClipPath			3	5
TSVGDefs	1	1	1	6
TSVGEllipse				2
TSVGGradientStop	26	353	16	4
TSVGGroup	1	25	490	3
TSVGLinearGradient	36	320	16	2
TSVGMetadata	1	1	1	1
TSVGPath	19	417	725	685
TSVGPolygon				61
TSVGRadialGradient	16	90		6
TSVGRectangle				5
TSVGUse				5
Total Elements	**100**	**1207**	**1252**	**786**

Table 1 Listing of SVG Elements for each test SVG

Something to note is that these test applications should not be taken as an indicator of the pure TCanvas speed you can get in FMX. The SVGs are incredibly complicated, with lots of paths, gradients, transparencies, CSS styles, etc. The RSCL is saving canvas state before drawing each element and restoring canvas state after drawing each element. Each SVG element has its own brush and pen (or Fill and Stroke in FMX terminology). There are a lot of canvas operations occurring besides the pure filling of a graphics primitive (rectangle, ellipse, path, etc).

*Another thing to note is that these drawing tests are not representative of the RSCL TRSSVGImage component performance. The TRSSVGImage component draws an SVG **once** to a backscreen bitmap and then just draws the backscreen bitmap as needed when painting is called for. These drawing tests deliberately use a TPaintBox and draws an SVG each time the TPaintBox is refreshed. Even then, we removed the first drawing call from the performance times as the RSCL has to build the Brushes and Pens (Fills and Strokes) in the first drawing call but caches them for later drawing calls.*

RSCL Drawing Application, FMX, Win32

Our first series of tests with the RiverSoftAVG SVG Component Library (RSCL) will be the RSCL Drawing application for Win32. As mentioned earlier in the series, our test machine for Windows is a Windows 7 64-bit Intel I7 930 @ 2.8 GHz CPU, and 6 GB RAM. The graphics card is an ATI HD5700 (not a top performer) using the latest AMD 14.1 drivers and with DirectX 11. *Note that no special effort was made to choose the TCanvas implementation; we let each Delphi version pick its default.*

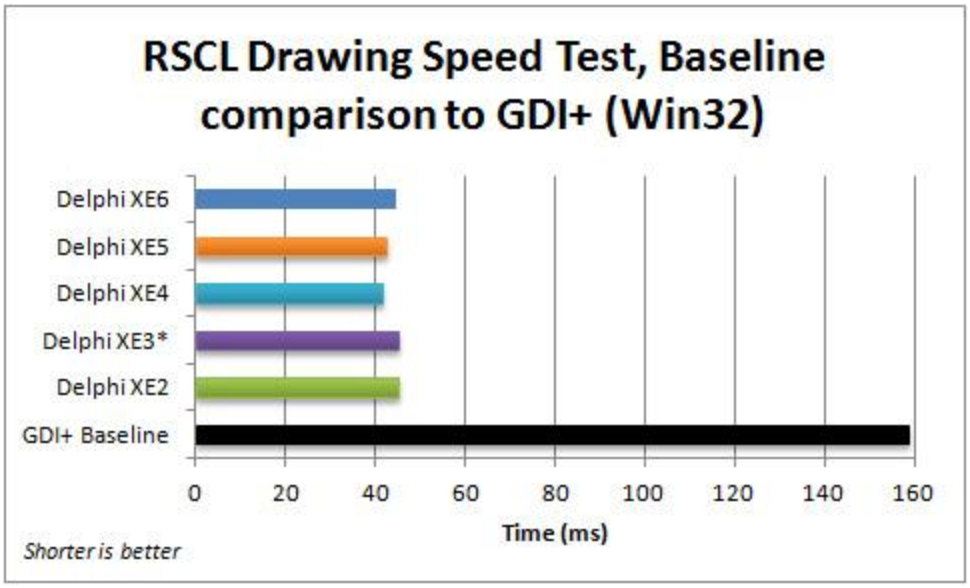


Figure 68 Comparison of averaged execution speed for the FMX RSCL Drawing application (Win32) with the baseline GDI+ VCL application, for Delphi XE2 to XE6

As a sanity test, we will first compare the FMX version to the VCL version. Since the VCL version uses GDI+, which is a software graphics library, the hardware-optimized FMX TCanvas operations should be significantly faster. Thankfully, that is what we see. **Every Delphi version of the FMX application was almost 4x faster on average than the VCL application (XE6).**

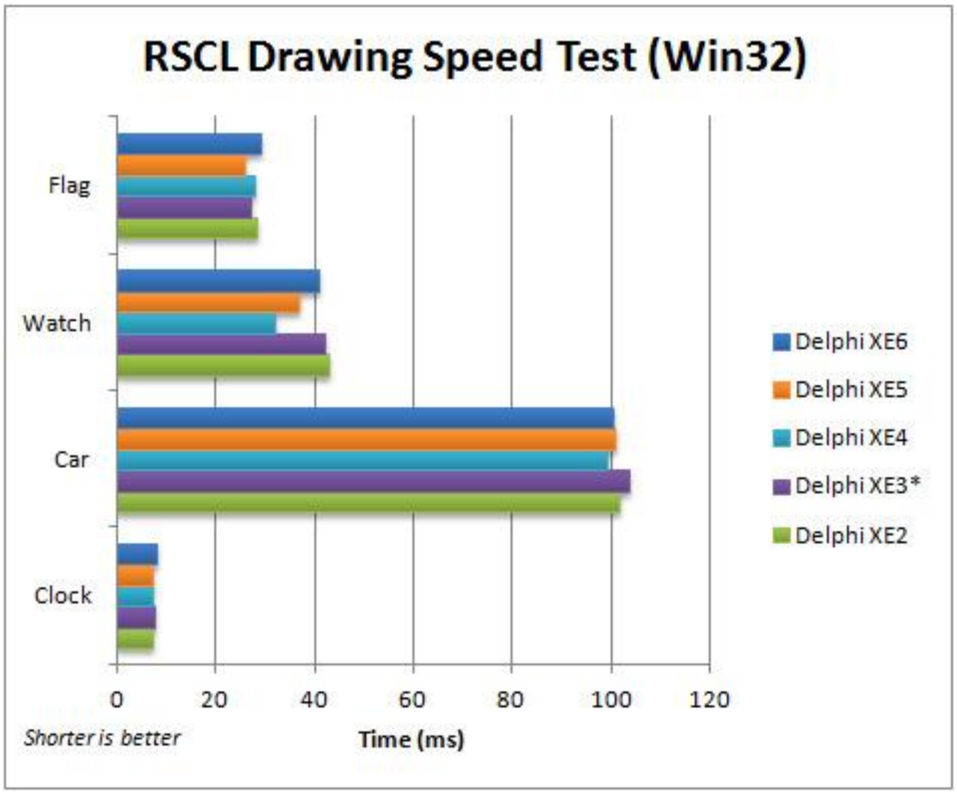


Figure 69 Comparison of execution speed for the FMX RSCL Drawing application (Win32) with Delphi XE2 to XE6

Zooming in on the drawing performance for each individual SVG, we see that Delphi XE5 drew the Flag fastest, but that Delphi XE4 drew the Watch and Car the fastest. Every version of Delphi drew the Clock in 7-8 ms (11x-12x faster than the GDI+ performance). **Overall, Delphi XE4 was the fastest in Win32.**

**Note that Delphi XE3 had visual artifacts drawing the Car and the Watch.*

RSCL Drawing Application, FMX, Win64

Moving to 64-bit, every version of Delphi was faster than its 32-bit counterpart (running on Windows 7 64-bit). This time, Delphi XE2 drew the Flag fastest. Delphi XE6 drew the Watch fastest. Delphi XE4 was fastest drawing the Car. We have a 5-way tie for drawing the Clock at 7 ms. **Averaging the results, Delphi XE4 is again the winner for Win64.**

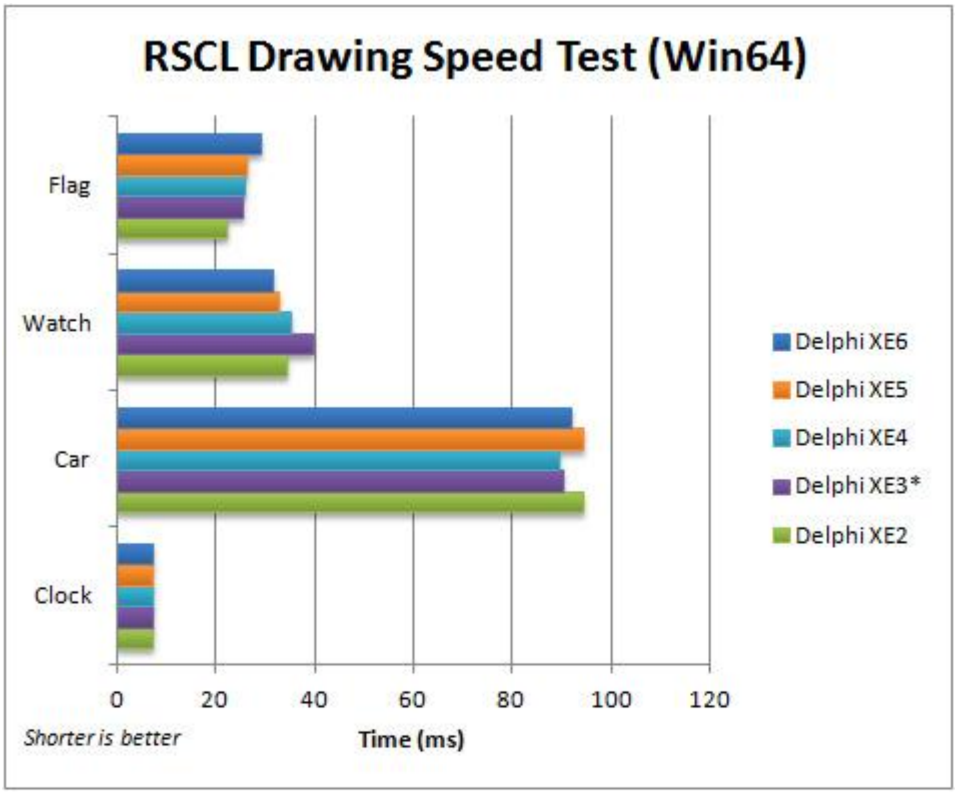


Figure 70 Comparison of execution speed for the FMX RSCL Drawing application (Win64) with Delphi XE2 to XE6

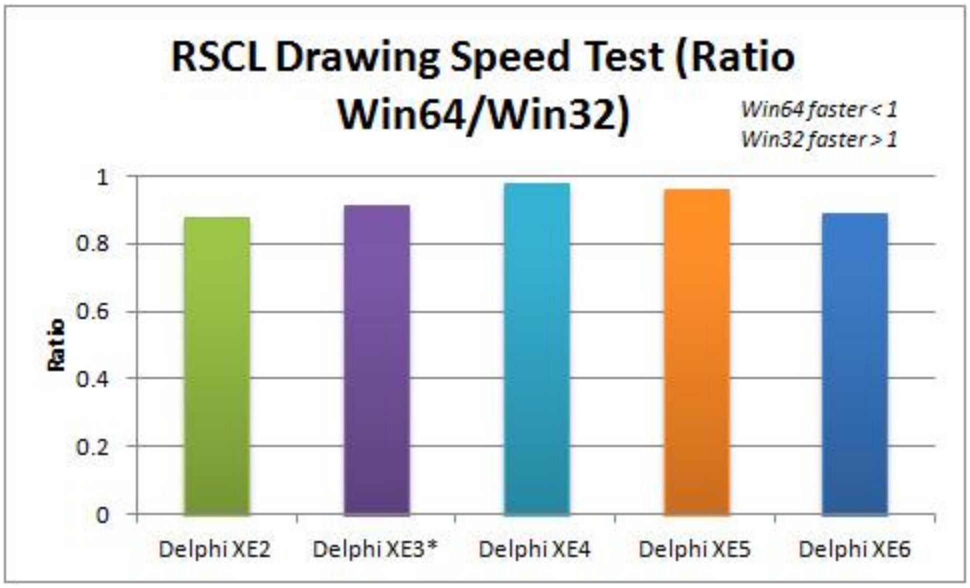


Figure 71 Ratio of Win64 to Win32 execution speed for RSCL Drawing apps in Delphi XE2 to XE6 (FMX)

Comparing the ratio of performance of Win64/Win32, Delphi XE2 and XE6 saw the largest performance gain when moving to Win64. It is difficult to see if Delphi XE6 gained anything with its DirectX 11 compatibility.

**Note that Delphi XE3 had visual artifacts drawing the Car and the Watch.*

RSCL Drawing Application, FMX, OSX

Things get much more interesting when we move to Mac OSX. The FMX programs are using OpenGL instead of DirectX. **Delphi XE2 is fastest drawing the Car and ties with XE3 for drawing the Clock. Delphi XE5 is fastest drawing the Watch; XE3 gets another tie when drawing the Flag.** Delphi XE6's showing in OSX is not so impressive at first glance, as it is dead last with the two least complicated SVG objects (Clock and Flag) and in the back of the pack with the two more complicated SVG objects. However, it should be noted that the XE6 is only 5.5 ms (Clock), 3 ms (Car), 4 ms (Watch), and 4 ms (Flag). **Averaging the results, Delphi XE2 is the winner for OSX.**

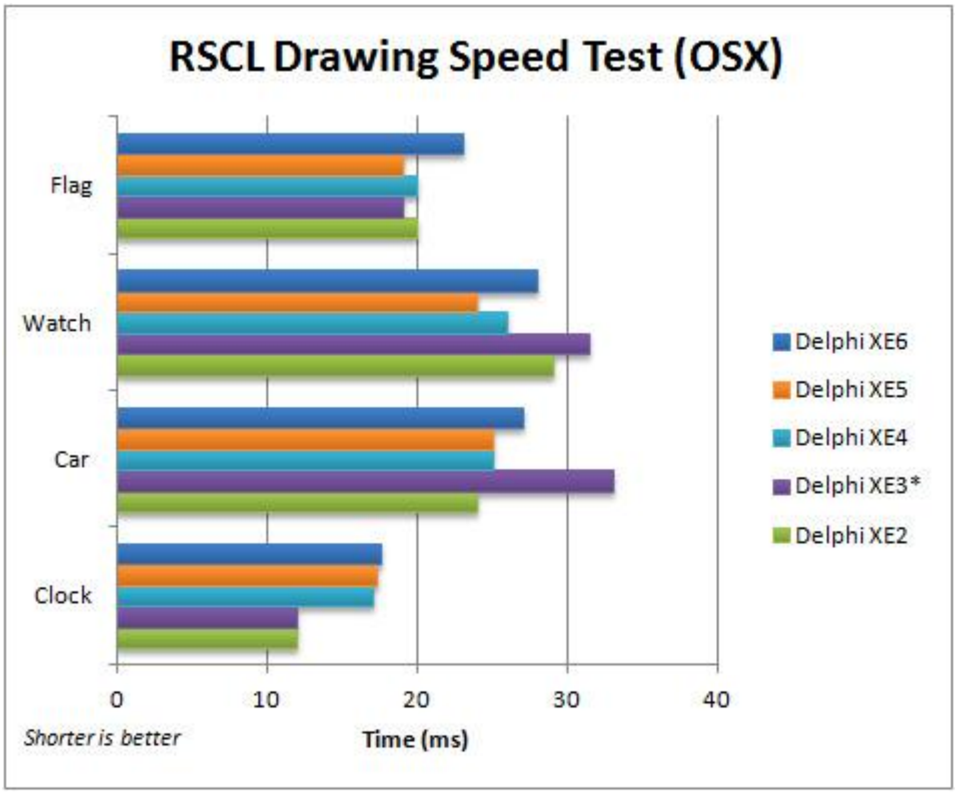


Figure 72 Comparison of execution speed for the FMX RSCL Drawing application (OSX) with Delphi XE2 to XE6

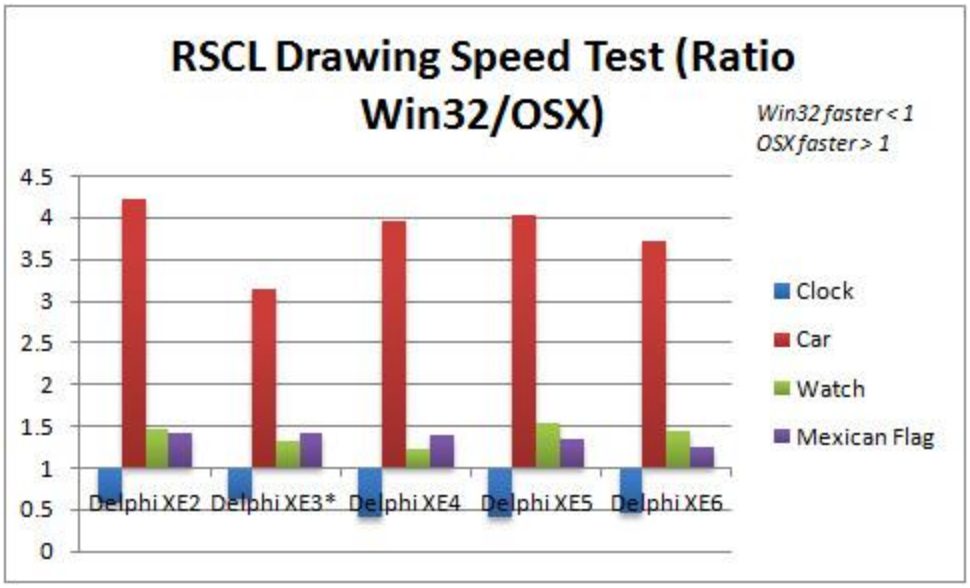


Figure 73 Ratio of Win32 to OSX execution speed for RSCL Drawing apps in Delphi XE2 to XE6 (FMX)

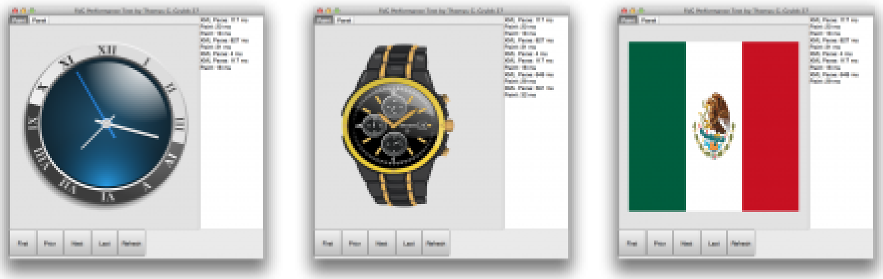
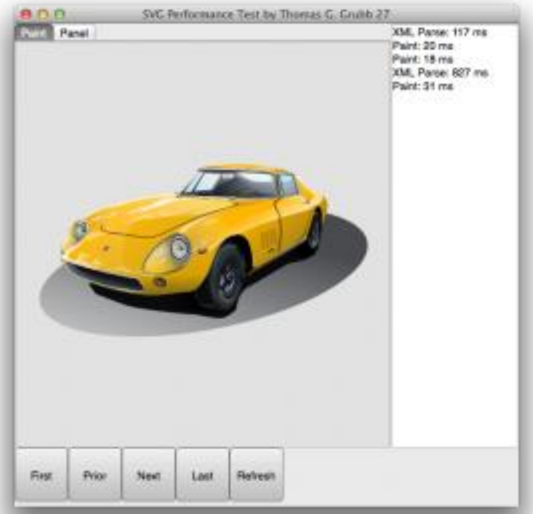


Figure 74 Display of the SVG Clock, Watch, and Flag using FMX TCanvas operations in the OSX test application



```
XML Parse: 117 ms
Paint: 20 ms
Paint: 18 ms
XML Parse: 827 ms
Paint: 31 ms
```

Figure 75 Display of the SVG Flag using FMX TCanvas operations in the OSX test application

The really interesting thing is how fast the OSX implementation is compared to the Win32 implementation. With the exception of the Clock, the OSX versions clean the Win32 versions clocks (pun unintended :-)), being anywhere from 1.2x-4.2x faster. It must be noted that the Windows box and the Mac Mini are not exactly comparable. The Mac Mini is an OSX 64-bit Intel I7 @ 2.3 GHz with 4GB RAM. Admittedly, the ATI HD 5700 is almost 5 years old now, but the Mac Mini uses integrated graphics. Examining the outputs closely, there are slight differences in the output (see the dark lines in the Car). Generally, however, the output from the OSX versions are excellent.

**Note that Delphi XE3 had visual artifacts drawing the Car and the Watch.*

RSCL Drawing Application, FMX, iOS

Moving to mobile, our drawing times get much slower. This is expected as the mobile hardware is much slower than the desktop hardware. For testing iOS apps, we used an iPod Touch with 32GB. For testing Android apps, we used a Nexus 7 (2013) with 32GB.

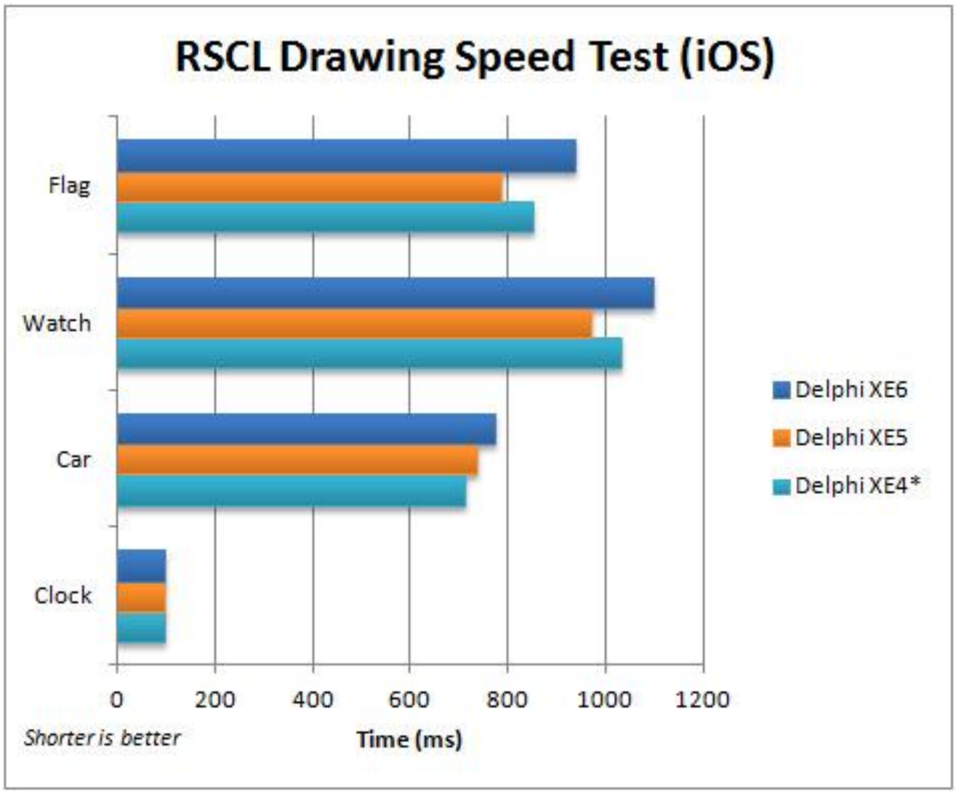


Figure 76 Comparison of execution speed for RSCL Drawing App (iOS) from Delphi XE4 to XE6

The interesting thing is that Delphi XE5 is the clear winner on iOS in drawing the SVGS. It is fastest drawing three of the SVGs and only 26 ms (approx. 4%) slower than XE4 when drawing the Car. Delphi XE6 is the slowest in drawing in iOS.

**Note that Delphi XE4 had minor visual artifacts (gradients) drawing on iOS.*

RSCL Drawing Application, FMX, Android

On an Android device, Delphi XE5 maintains its speed lead over Delphi XE6. Delphi XE5 is 4-10% faster than Delphi XE6 drawing the SVGs.

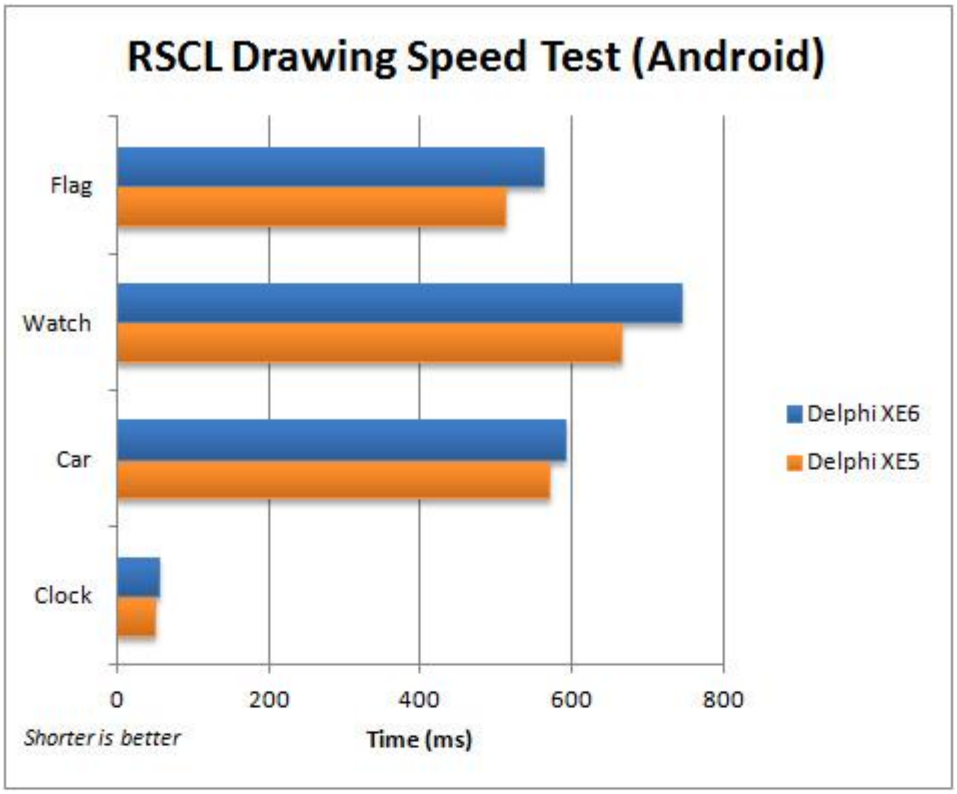


Figure 77 Comparison of execution speed for RSCL Drawing App (Android) between Delphi XE5 and XE6

RSCL Primitives Application, FMX, Win32

In the previous section, we compared SVG drawing performance for the different versions of Delphi using low-level canvas routines. However, the RiverSoftAVG SVG Component Library (RSCL) also contains a TRSSVGPanel component, which is able to generate FMX shape controls (TLayout and TPath mostly) descendants from an SVG to allow GUI interaction. It was our thought that this could be useful for performance testing FMX primitives speed as these primitives are what controls are made of in the FMX library. The RSCL converts the SVG elements listed in the last post into the following primitives:

	Clock	Car	Watch	Flag
TRSSVGLayout	2	26	491	4
TRSSVGPath	19	417	722	749
Total Elements	21	443	1213	753

Table 2 Listing of SVG Primitives created for each SVG

Note that the loading, parsing, and building of the RSCL SVG elements and primitives are not tested in these tables. Rather, the speed that the FMX library performs in refreshing the display of all the primitives that make up the SVG representation.

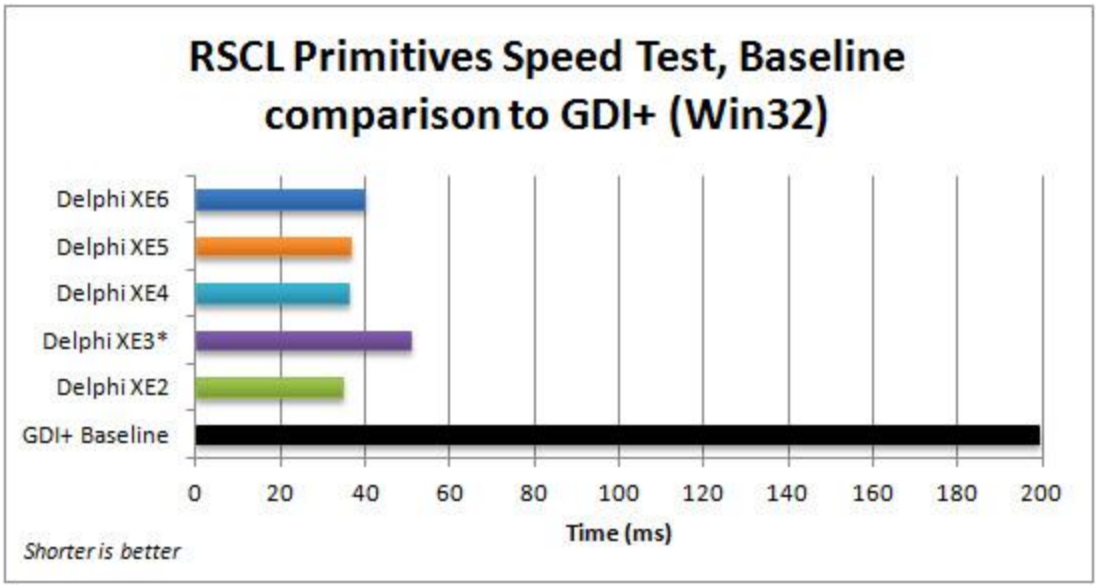


Figure 78 Comparison of averaged execution speed for the FMX RSCL Primitives application (Win32) with the baseline GDI+ VCL application, for Delphi XE2 to XE6

Before starting, we again did a sanity test to compare the FMX version to the VCL version. The VCL version uses GDI+, which is a software graphics library, so the hardware-optimized FMX TCanvas operations should be significantly faster. Thankfully, that is what we see. **Every Delphi version of the FMX application was 4x-5x faster on average than the VCL application (XE6).**

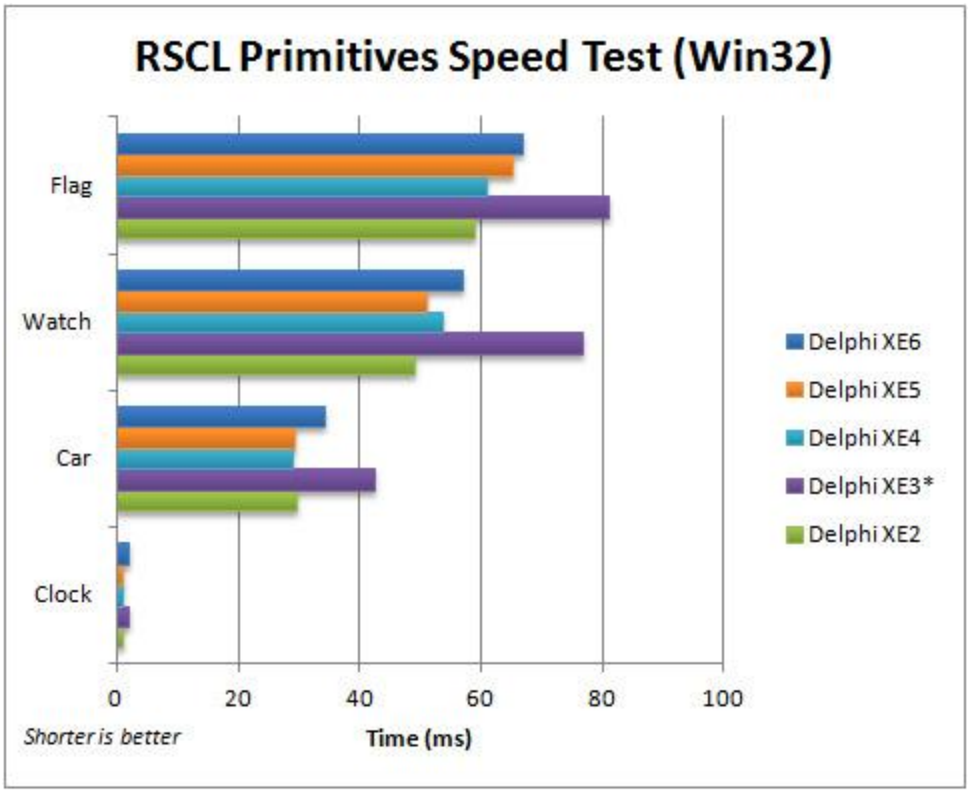


Figure 79 Comparison of execution speed for the FMX RSCL Primitives application (Win32) with Delphi XE2 to XE6

Zooming in on the primitives drawing performance for each individual SVG, **Delphi XE2 dominates,** drawing 2 SVGs fastest outright, tieing in drawing the Clock, and only 2/3 ms behind the leader drawing the Car SVG. Delphi XE3 does especially poorly in performance. Delphi XE6 comes in 4th place trailing XE2 by approximately 14%.

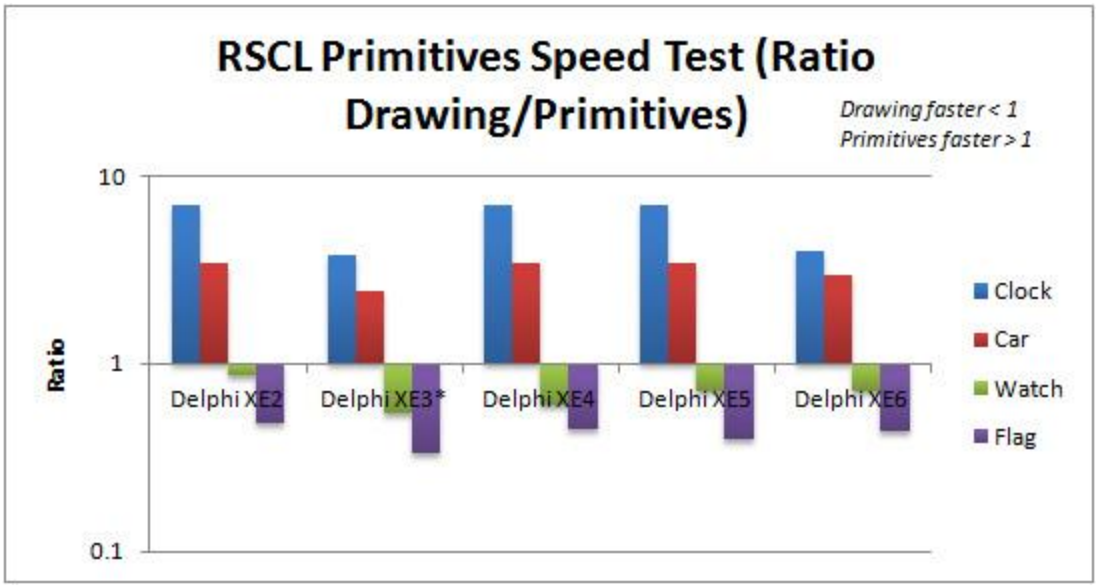


Figure 80 Ratio of Drawing vs FMX Primitives execution speed for RSCL applications in Delphi XE2 to XE6 (Win32)

Comparing performance when drawing directly to the canvas versus using primitives, we see a mix of results. The Clock and the Car are faster when using primitives, and the Watch and the Flag are faster when using the canvas directly. Without more extensive profiling, it is difficult to say exactly why this is so. If you remember the SVG element breakdown from the last post, the Clock and the Car have significantly more gradients and gradients stops and a lot fewer paths than the Watch and the Flag. Since the RSCL caches Brushes and Pens after the first drawing call, the RSCL primitives do not have an advantage from the greater number of gradients. **It is probable that the FMX library bogs down as the number of primitives goes up.**

**Note that Delphi XE3 had visual artifacts drawing the Car and the Watch.*

RSCL Primitives Application, FMX, Win64

Moving to 64-bits, Delphi XE2 again wins and is the only version to actually improve its average score from 32-bits. It renders the Watch and Flag fastest, ties for the Clock, and trails the leader by 1.66 ms for the Car. Delphi XE3 continues its poor performance. Delphi XE6 is approximately 16% slower than Delphi XE2.

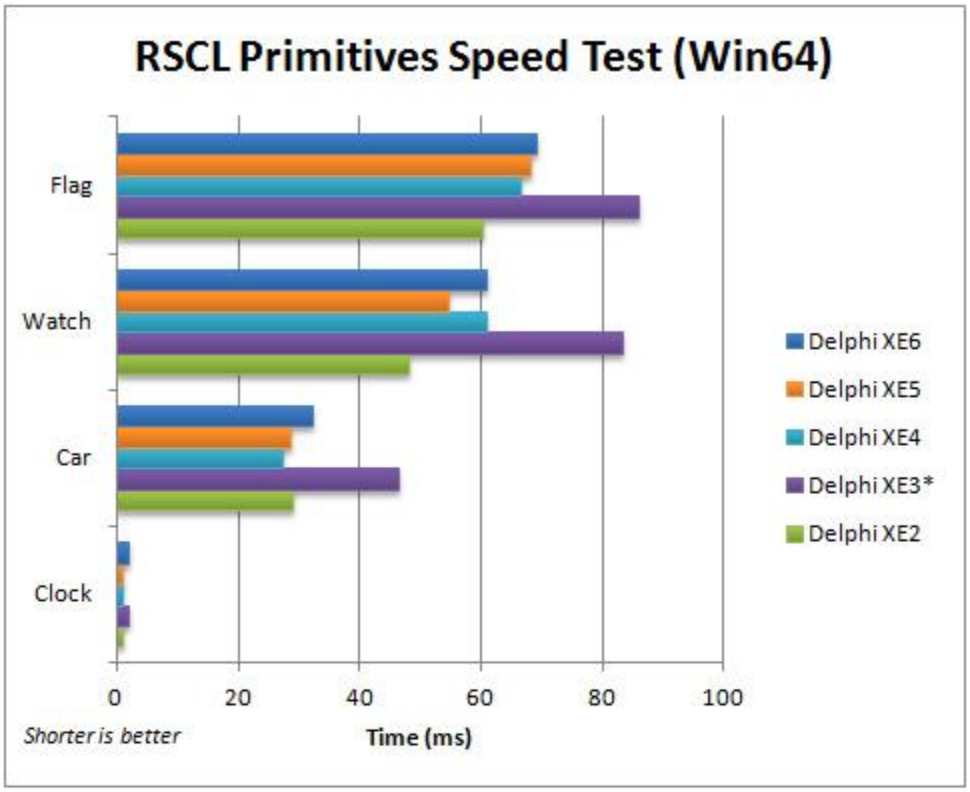


Figure 81 Comparison of execution speed for the FMX RSCL Primitives application (Win64) with Delphi XE2 to XE6

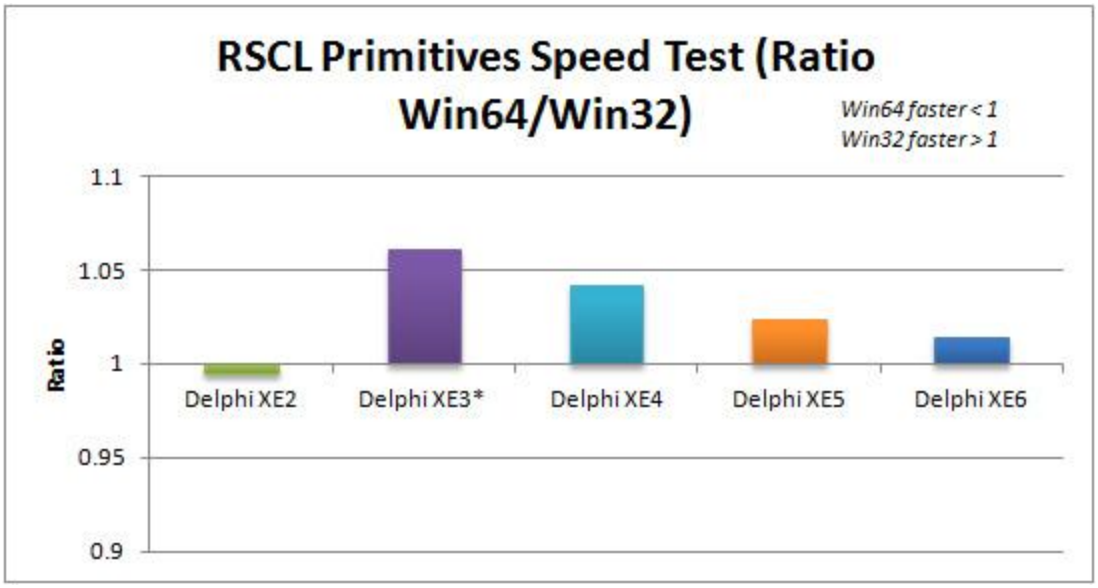


Figure 82 Ratio of Win64 to Win32 execution speed for RSCL Primitives applications in Delphi XE2 to XE6 (FMX)

**Note that Delphi XE3 had visual artifacts drawing the Car and the Watch.*

RSCL Primitives Application, FMX, OSX

When we start testing using the OSX platform, the performance of all versions of Delphi suffer dramatically. **Unlike drawing directly to the canvas, the FMX primitives on OSX are very slow… for Delphi XE3, it is even slower than the non-hardware accelerated GDI+ version!**

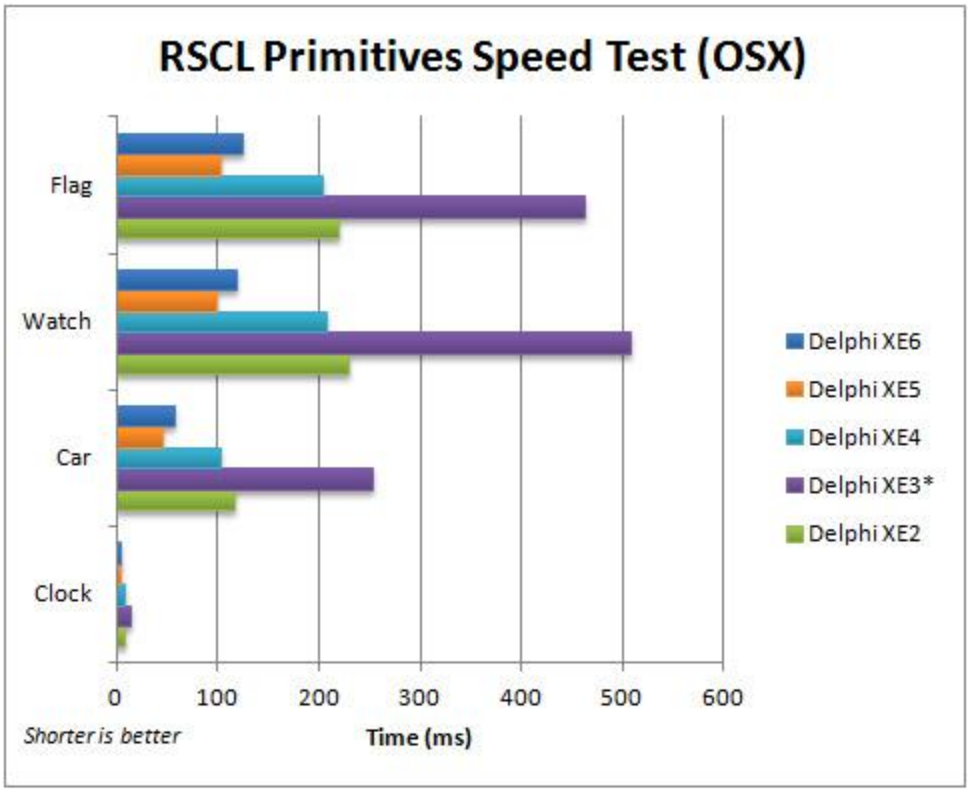


Figure 83 Comparison of execution speed for the FMX RSCL Primitives application (OSX) with Delphi XE2 to XE6

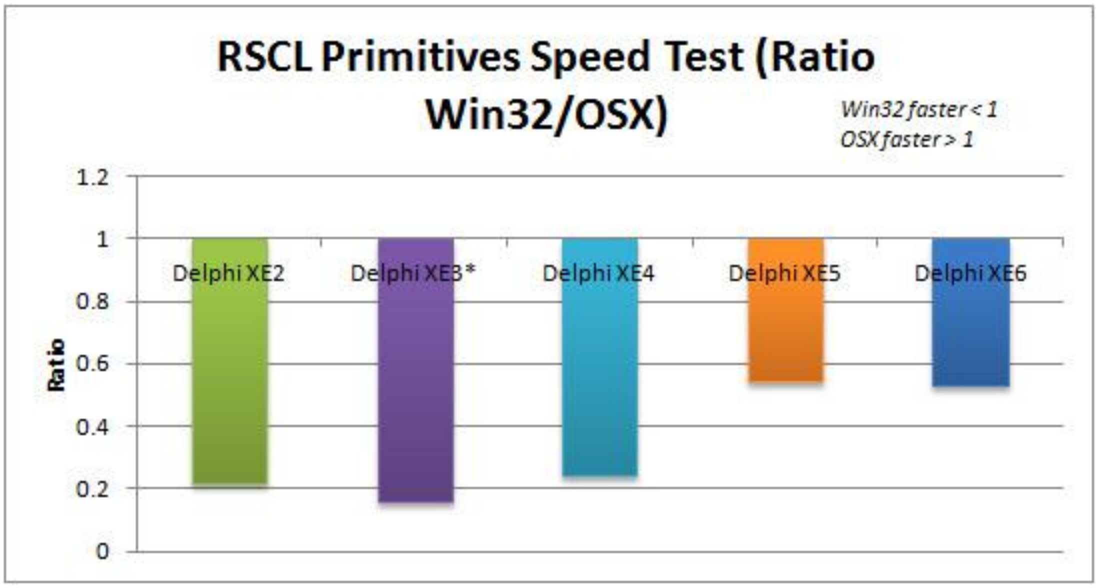


Figure 84 Ratio of Win32 to OSX execution speed for RSCL Primitives applications in Delphi XE2 to XE6 (FMX)

Delphi XE2, XE3, and XE4 suffer the most coming to the OSX platform. Delphi XE5 and XE6 make great showings in comparison. **Delphi XE5 is the fastest when using primitives by a large margin.** Delphi XE6 is in second place but about 18% slower than XE5.

**Note that Delphi XE3 had visual artifacts drawing the Car and the Watch.*

RSCL Primitives Application, FMX, iOS

Comparing Delphi XE4 through XE6 on mobile, Delphi XE5 maintains its performance advantage. On iOS, Delphi XE5 is fastest using FMX primitives, with a 6%-7% speed advantage. The Delphi XE6 numbers are very respectable. With SVGs with more paths (Watch and Flag), Delphi XE6 manages to be faster than XE4.

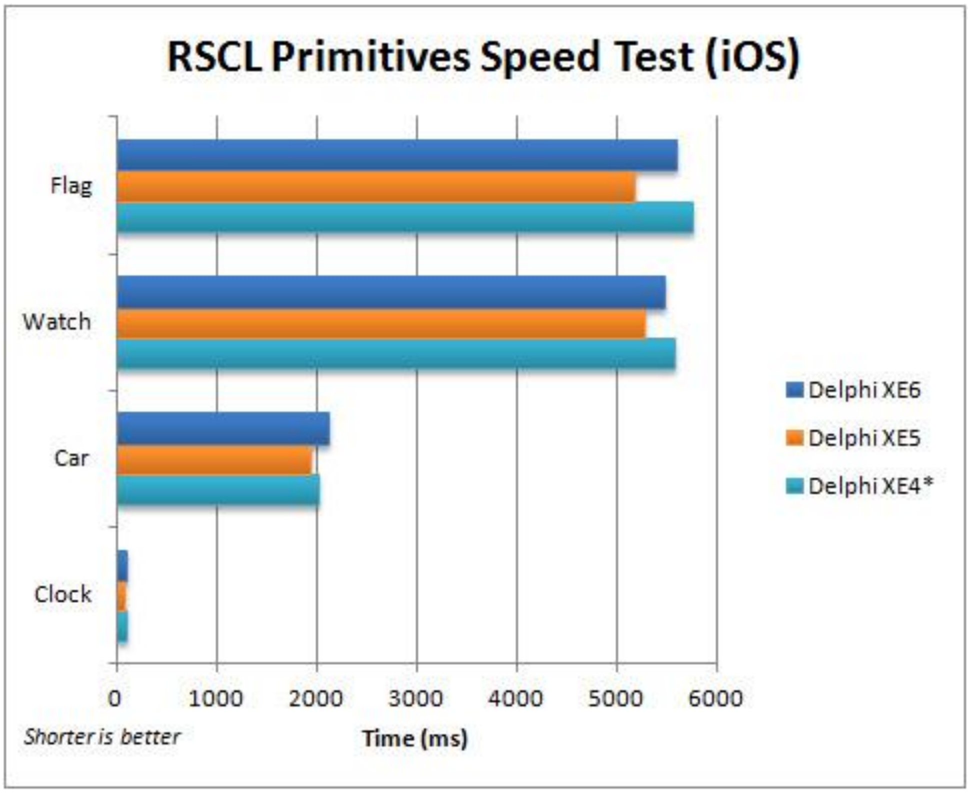


Figure 85 Comparison of execution speed for the FMX RSCL Primitives application (iOS) with Delphi XE4 to XE6

**Delphi XE4 was last on iOS. In addtion, there were minor visual artifacts (gradients) drawing the SVGs.*

RSCL Primitives Application, FMX, Android

Our final test is on the Android platform. **Delphi XE5 wins the performance crown when using FMX primitives on Android, though it is a narrow victory.** Delphi XE6 loses by over 6% when drawing the Clock. However, as the number of primitives go up, Delphi XE6 starts catching up and is less than 2% slower when drawing the most complex SVG (Watch).

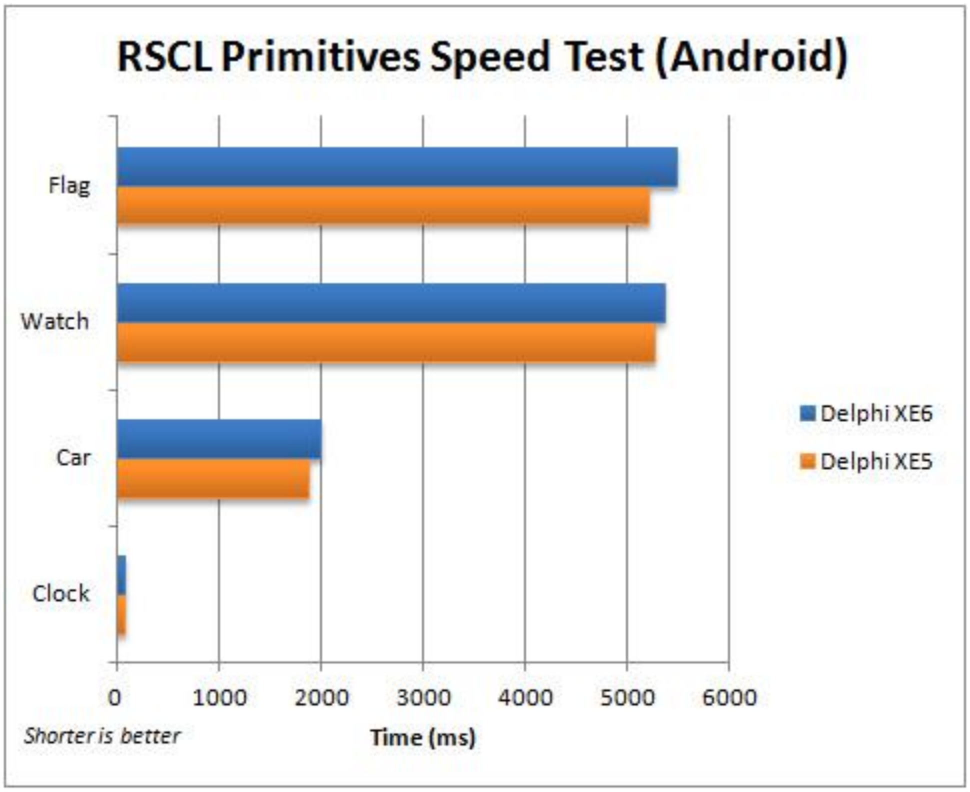


Figure 86 Comparison of execution speed for the FMX RSCL Primitives application (Android) between Delphi XE5 and XE6

Finally! We are finished with our performance tests! In the final section, I summarize the results and discuss how Delphi XE6 did.

The Last Chapter

With the release of Delphi XE6 and Embarcadero's emphasis on Quality, Performance, and Stability (QPS), I wanted to see for myself the level of improvement, especially in performance. Delphi XE6 is **definitely faster and more responsive** than the last few versions, especially in FMX, but I wanted to see if I could quantify the performance improvement. Before starting, I made some predictions about what I would see when comparing Delphi 2010-XE6. Let's see how I did:

- **EXE size will probably increase with every version of Delphi.** This turned out to be mostly true. Each version of Delphi has increased the EXE size over the one before... EXCEPT this ignores the one proverbial elephant in the room, Delphi XE3. Delphi XE3 massively increased EXE sizes, both in VCL and FMX. Thankfully, Delphi XE4 reduced XE3's excessive weight gain. However, EXE size has been growing back to XE3 levels with every version since then. It should be noted that VCL EXE sizes have been growing slower than FMX EXE sizes.
- **FMX executables will be larger than VCL executables.** This was definitely true. This was completely expected as FMX controls are non-native controls (i.e., they don't use OS level equivalents) so all the drawing and interaction code must be compiled into the executable.
- **FMX executables will be slower than VCL executables, though each new version of Delphi for a platform should improve.** This was mostly true. Except for where the VCL version is using a software renderer for heavy graphics drawing vs the FMX hardware renderer, the FMX versions are slower than their VCL counterparts. There was one outlier, when strings are added to a TMemo and BeginUpdate/EndUpdate have not been called, the Delphi XE6 FMX version was faster than the VCL feature. However, the discrepancy is easily explained as the VCL TMemo draws every string as it is added while the FMX TMemo does not. **The part that is not completely true is that each new version of Delphi would be improving the FMX speed. Speeds were all over the place, and earlier versions were sometimes significantly faster.** Delphi XE6 is definitely faster than XE5 though.
- **Win32 and Win64 compilation should be faster than other platforms.** This was true as well. The Win32 and Win64 compilers (written by Embarcadero) are fastest. However, the OSX version (also written by Embarcadero) is very close. The iOS and Android versions are much slower, even when not counting the deployment step.
- **Windows FMX executables will be faster than other platforms' FMX executables.** This was mostly true to true. Since the test machines were not comparable, it was difficult to really test this. However, when drawing the RSCL directly to the canvas, the OSX implementation was comparable to the Win32 implementation. With the exception of the Clock, the OSX versions cleaned the Win32 versions clocks, being anywhere from 1.2x-4.2x faster. It must be noted that the Windows box (Windows 7 64-bit Intel I7 930 @ 2.8 GHz CPU, ATI HD 5700 graphics card, and 6 GB RAM) and the Mac Mini (OSX 64-bit Intel I7 @ 2.3 GHz, integrated graphics, and 4GB RAM) are not exactly comparable. Admittedly, the ATI HD 5700 is almost 5 years old now, but the Mac Mini uses integrated graphics. Examining the outputs closely, there are slight differences in the output (see the dark lines in the Car). Generally, however, the output from the OSX versions are excellent. **This hypothesis would need to be more rigorously investigated.**

Comparing the competitors...

To make my final evaluation, I wanted to be as objective as possible and use concrete numbers. I decided to ignore EXE size (Delphi 2010 wins! :-)) and compilation speed (more mixed but generally earlier versions of Delphi). I concentrated solely on performance metrics. This means I ignored features (Delphi XE6 wins!), number of platforms (Delphi XE6 wins), and stability (???? wins!) For each test application, I averaged the scores (e.g., the Hello World Listbox performance score is an average of the TListBox scores). This does mean that the winner of an exponential test (e.g., 1000s of items in a TListBox) can win even if their performance with less items was not the best. I then normalized the averages (e.g., the winner is equal to 1 and every other version is between 0 and 1 depending on their average). Finally, I added each test score together to get a maximum score per platform. For example, the VCL 32-bit score includes the normalized performance scores from the Hello World project (ListBox and Memo), the IECS Advanced Console and the

IECS Basic Console (Max of 4 points). The OSX score includes the normalized performance scores from the Hello World project (ListBox and Memo), the IECS Advanced Console, the IECS Basic Console, RSCL Drawing project, and the RSCL Primitives project (Max of 6 points). In the graphs below, you can easily see how the normalized scores contributed to each Delphi version's score. ***Note that my scores are just for the tests in this document. Other tests would give vastly different results. This document is not meant to be a recommendation of one Delphi version over another.***

Now for the awards ceremony...

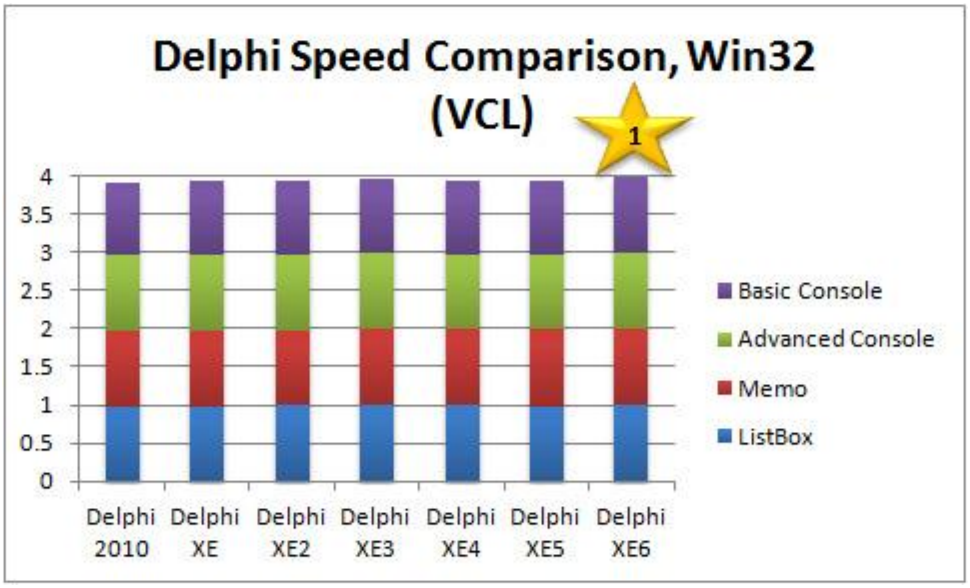


Figure 87 Overall Performance Score Comparison, Win32 VCL

Surprisingly, Delphi XE6 wins the gold award in my performance tests for Win32 VCL. The differences are **extremely** minor between all versions of Delphi so this coveted award could have gone to any of them :-) Delphi XE3 manages second place and a silver medal. Delphi XE scores the third place bronze award.

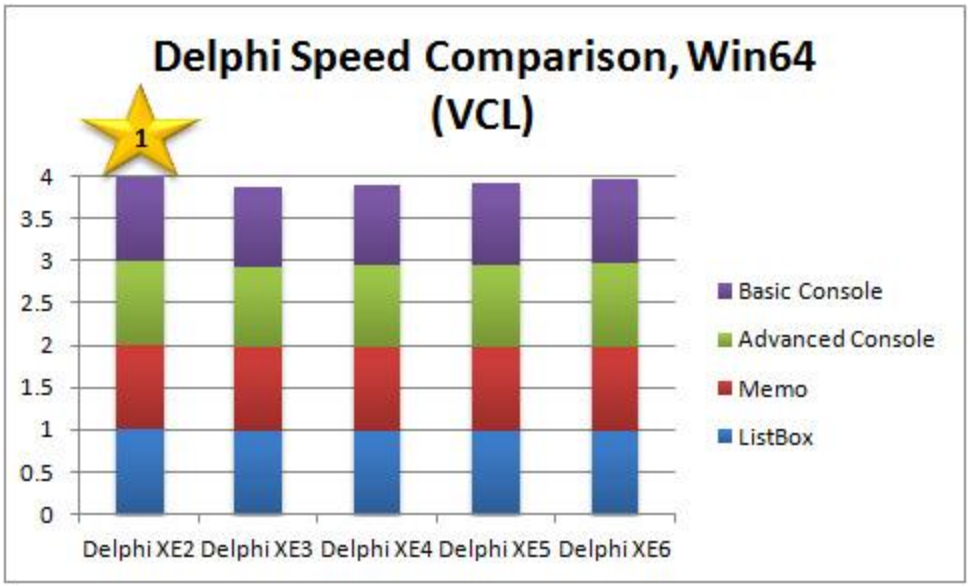


Figure 88 Overall Performance Score Comparison, Win64 VCL

For Win64 VCL (dropping Delphi 2010 and XE from the running), Delphi XE2 manages to barely eke out the gold award. Delphi XE6 wins silver and XE5 bronze.

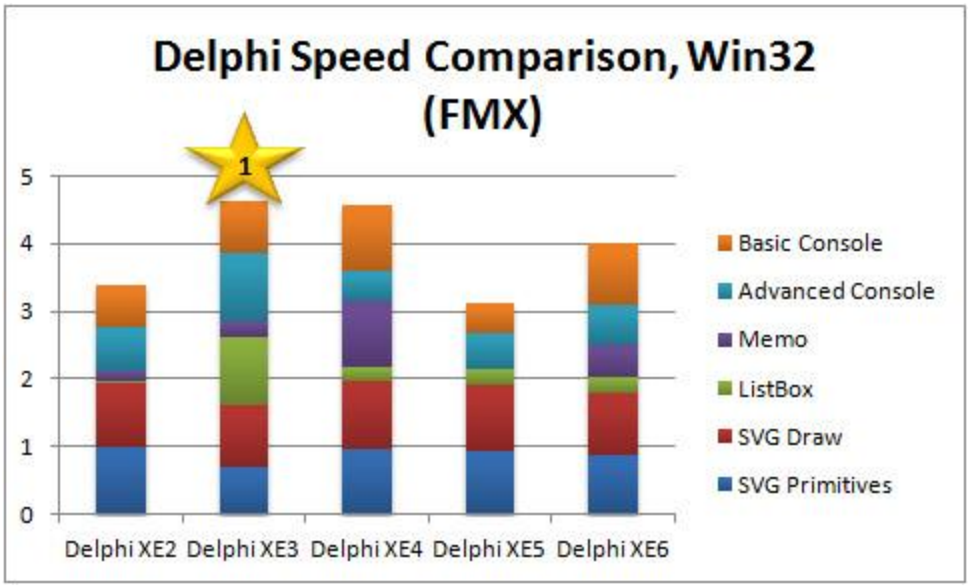


Figure 89 Overall Performance Score Comparison, Win32 FMX

Moving to FMX, the performance differences get much larger. **Delphi XE3, despite its huge EXE sizes and poor Memo performance, produces the fastest FMX executables on Windows in these tests to win a gold medal. Its ListBox performance is what really stands out.** Delphi XE4 makes a very strong second place showing for silver. Bronze goes to Delphi XE6.

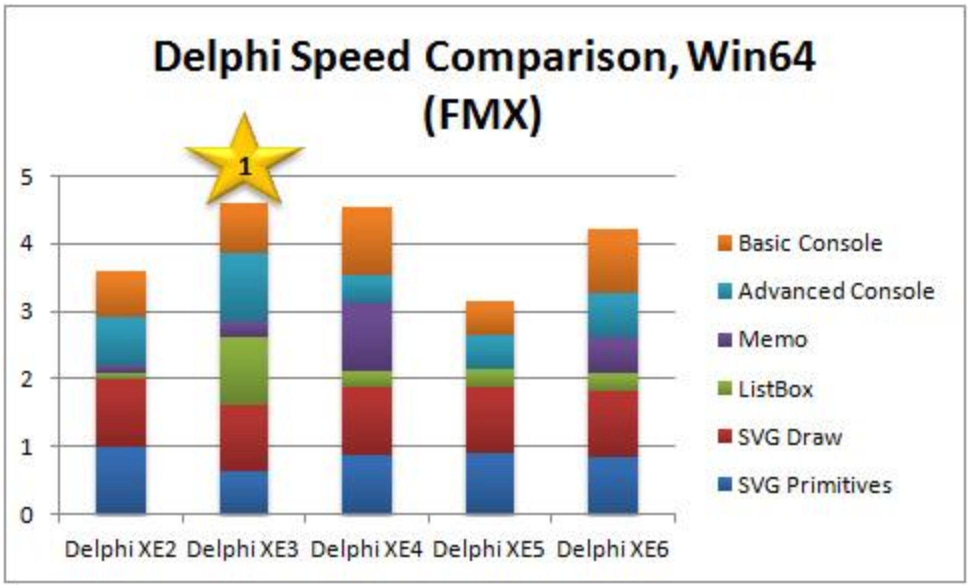


Figure 90 Overall Performance Score Comparison, Win64 FMX

Going to 64-bit, Delphi XE3 again wins the gold with Delphi XE4 getting the silver. Delphi XE6 comes again in third place, though its score improved noticeably over 32-bit.

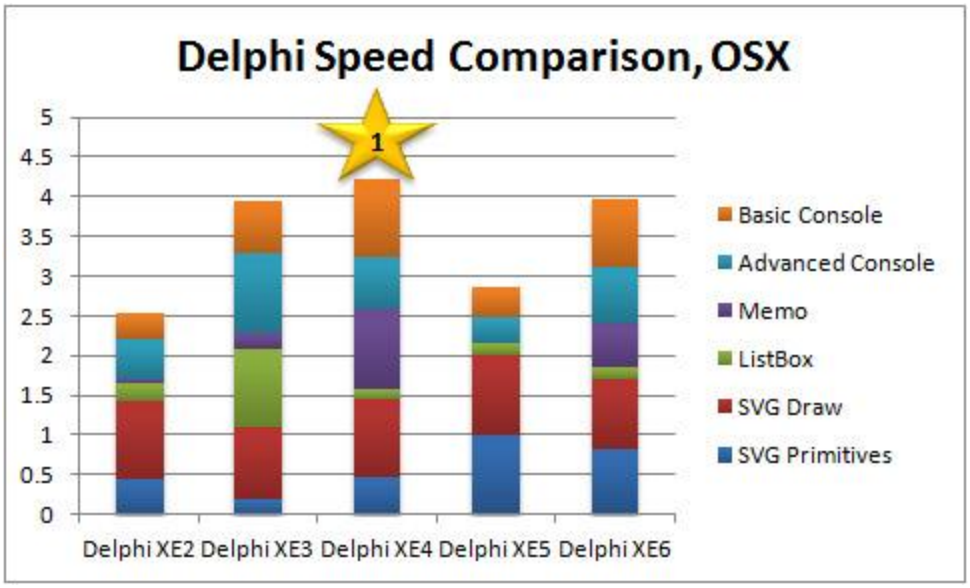


Figure 91 Overall Performance Score Comparison, OSX

Switching to the Mac, Delphi XE4 wins its first gold medal. Delphi XE6 sneaks past XE3 to win the silver.

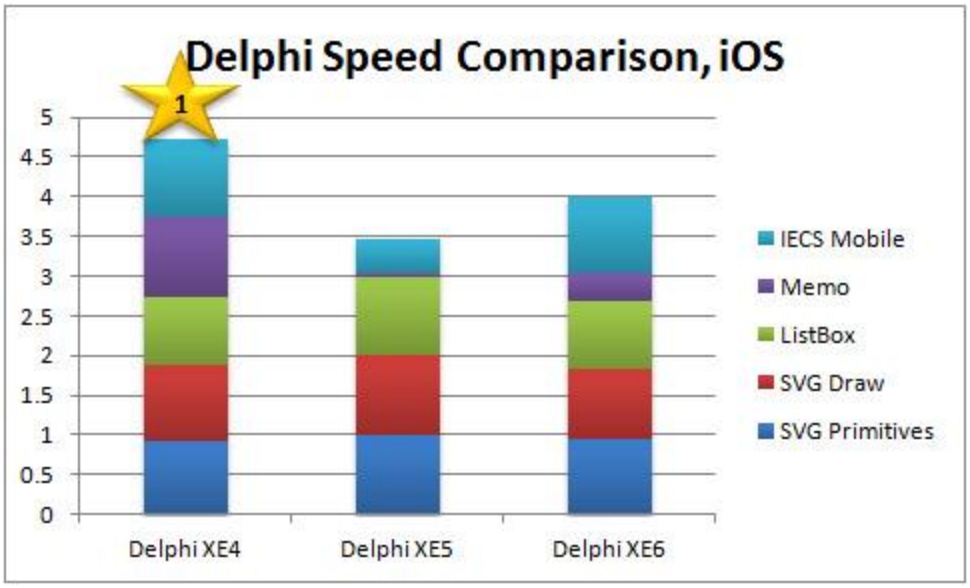


Figure 92 Overall Performance Score Comparison, iOS

Shifting to mobile, we are down to Delphi XE4, XE5, and XE6. **Delphi XE4 wins another gold medal for the iOS platform on its solid memo performance.** Delphi XE6 gets second place for the silver and XE5 trails for the bronze.

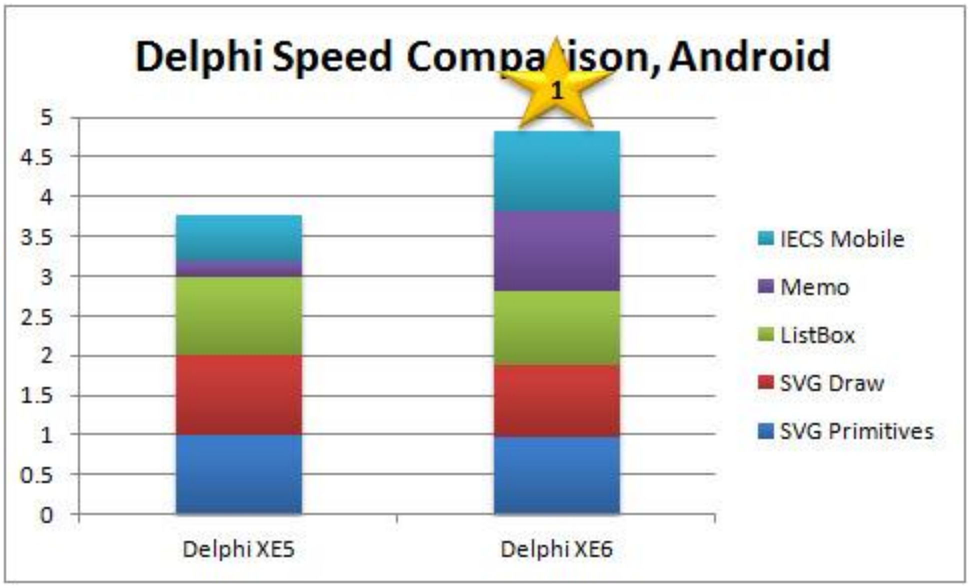
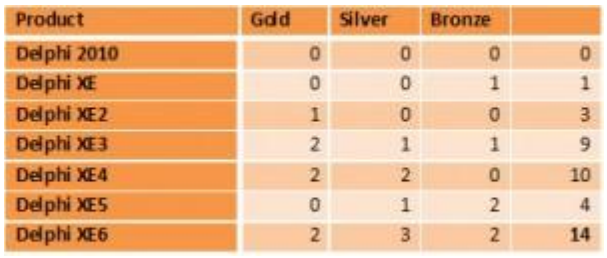


Figure 93 Overall Performance Score Comparison, Android

For our last platform, Android, Delphi XE6 wins handily, easily beating XE5.

Conclusion

So what does this mean? How did Delphi XE6 perform on its mantra of "Quality, Performance, and Stability?" As far as this series of **performance** tests are concerned, **Delphi XE6 did neither as well as I hoped nor as bad as I feared**. It only outright won a platform twice (and one of those was by a whisker on Win32 VCL). However, **Delphi XE6 was a solid performer and competitive for all platforms. Compared to its immediate successor (Delphi XE5), it is a huge advance in performance** and nearly brings it back up to the levels of Delphi XE3 and XE4, both of which performed surprisingly well in FMX.



Product	Gold	Silver	Bronze	
Delphi 2010	0	0	0	0
Delphi XE	0	0	1	1
Delphi XE2	1	0	0	3
Delphi XE3	2	1	1	9
Delphi XE4	2	2	0	10
Delphi XE5	0	1	2	4
Delphi XE6	2	3	2	14

Table 3 Overall Performance "Medal" Count (Gold=3, Silver=2, Bronze=1)

If you do a medal count of the different platforms and give 3 points for Gold, 2 points for Silver, and 1 point for Bronze, Delphi XE6 racks up the most points. Even if you ignore the Android platform, Delphi XE6 still did 1 point better than XE4. If you then *also* ignored the iOS platform, Delphi XE6 would still tie with XE3 at 9 points. **Considering that it adds both iOS and Android platforms, seems to have fewer issues compared to Delphi XE3 and XE4, and vastly superior performance compared to Delphi XE5, I would say that Embarcadero achieved their goal.**

Almost a month and over 12,000 words later, I have finally finished this series of tests on the performance comparison from Delphi 2010 to Delphi XE6. I hope you have enjoyed them. It was a ton of work, but I think it was worth it. Happy CodeSmithing!