



Port applications to Windows on Arm

Version 4.0

Non-Confidential

Copyright © 2022 Arm Limited (or its affiliates).
All rights reserved.

Issue 01

102341_0400_01_en



Port applications to Windows on Arm

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.

Release information

Document history

Issue	Date	Confidentiality	Change
0400-01	10 February 2022	Non-Confidential	Initial release

Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly

or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm’s trademark usage guidelines at <https://www.arm.com/company/policies/trademarks>.

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on <https://support.developer.arm.com>

To provide feedback on the document, fill the following survey: <https://developer.arm.com/documentation-feedback-survey>.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Overview.....	6
2. General porting instructions.....	7
3. Tweeten application for the Electron framework.....	10
4. StaffPad application for the UWP framework.....	11
5. Chromium Embedded Framework.....	12
6. WPF to UWP and .NET 4.8.....	20
7. WinForms and .NET 5.....	27
8. WPF and .NET 6.....	35
9. Related information.....	43

1. Overview

This guide shows how to port applications to Windows running on Arm-based devices. The guide first reviews general guidelines, and then shows examples for different frameworks: Electron porting for the Tweeten application, the Universal Windows Platform (UWP) porting for the StaffPad application, and building a native WebView application with the Chromium Embedded Framework.

To follow this guide, you need:

- A framework and development environment compatible with Windows on Arm.
- Any Windows on Arm device for testing.

We look at both requirements in greater detail in [General porting instructions](#).

By the end of this guide, you should know the general steps to follow if you want to port your application to Windows on Arm.

2. General porting instructions

This section reviews the general instructions for porting applications to Windows on Arm. It covers software requirements, changes in the application, and testing and deploying.

Authoring Windows on Arm GUI applications

This section describes other options you can use to author Windows on Arm GUI applications.

If you are familiar with Microsoft frameworks and APIs for building GUI applications, you can use a Microsoft framework of your choice. Native Windows on Arm applications can be built using one of the following frameworks:

- Win32 API
- Universal Windows Platform (UWP)
- Windows Forms, supported in .NET 5.0
- Windows Presentation Foundation (WPF), supported in .NET 5.0.8
- Xamarin.Forms

You can reuse existing knowledge and code with one of these frameworks. However, except for Xamarin.Forms, none of these platforms are cross-platform.

Another application building option is the [Electron](#) framework. Electron is a framework for building desktop applications with JavaScript (JS), HTML, and CSS. Electron embeds [Chromium](#) and [Node.js](#) into its binary, which allows you to maintain one JS codebase and create cross-platform apps that interacts directly with the operating system (OS). In a typical Electron app, you author the user interface (UI) and frontend logic using web technologies such as HTML and CSS with JS or TypeScript. You can call native code from JS using [node-addon-api](#).

Electron applications have a main process and one or more renderer processes. Any process can link shared libraries containing native code. To invoke native functions, you must marshal data to and over the Application Binary Interface (ABI) boundary, which can incur some overhead.

The benefit of Electron is that web UIs are portable. Also, the Electron framework provides a portable API to interface with various OS facilities, such as the taskbar and clipboard across multiple platforms including iOS, MacOS, Android, Windows, and Windows on Arm.

Electron is appropriate when most of the business logic does not need native performance. It also offers the ability to use native extensions for parts of your application where performance is critical. When packaging an Electron app, you must ensure the native code is compiled for the current target architecture.

The drawback of Electron is that a minimal application can be 30MB. Applications are pinned to the version of Chromium embedded in the Electron build which can be beneficial, because it ensures your application will not break when Chromium changes. Using a specific version is also a risk due to occasional security vulnerabilities in Chromium. When zero-day vulnerabilities occur, you must

wait until the issue is patched in Chromium and a new Electron build incorporates the Chromium update before you can update your application to keep it secure.

Another app creation option is the CEF, which is supported with Windows on Arm. Like Electron, CEF builds on the Chromium project to provide cross-platform GUI functions.

Electron generally functions as the host application, however you can embed CEF into an existing native application. Instead of using a JavaScript API, C and C++ APIs in CEF manage its runtime.

CEF is a good choice when a significant portion of the application needs to be, or is already, native code. CEF is smaller than Electron, using 4MB instead of 30MB. Like Electron, CEF introduces risk due to the potential of zero-day vulnerabilities in Chromium. When these appear, you must wait for a build of CEF incorporating the fixes and then create a new build of your application.

Select framework version and development environment

Frameworks that support applications for Windows on Arm include:

- Universal Windows Platform (UWP)
- Electron
- Chromium Embedded Framework (CEF)
- Chromium

You can write code for Windows on Arm on:

- Visual Studio 2017 with the 15.9 update and Arm64 toolset
- Visual Studio 2019, which is ready for Windows on Arm by default
- Visual Studio Code

Check third-party dependencies

Check that all your third-party dependencies have a version that is compatible with Windows on Arm. If they do not, check whether they are open source and easily convertible. If they are not, you will need to either find alternative libraries or rewrite your code to avoid these dependencies.

Here are some things to keep in mind while you are checking your third-party dependencies:

- Check frameworks and SDKs. Some, like React Native and OpenJDK, are available for Windows on Arm.
- Check libraries. Some, like Boost and Scintilla, are available for Windows on Arm.
- If you are using C/C++ libraries that have Intel Intrinsics you need to do one of the following:
 - Switch to a library without Intrinsics
 - Switch to a library that also supports Arm Intrinsics
 - Write a new library
- Platform-independent JavaScript or TypeScript modules like MobX and dragula work out of the box.

- Check the tools that you use for building and testing. Python, Yarn, and PuTTY support Windows on Arm.

Check internal dependencies

Before you can build your code for Windows on Arm, you need ensure that platform-specific code is handled correctly. Most bugs occur in the `#else` or `#ifdef {PLATFORM}`.

Set compilation target and perform a build

Set the compilation target to Arm64 and perform a build.

Test your build

You should always check your compiled application on a physical device. You can easily fix many layout and JavaScript bugs when you see the application running on a device.

If you need to debug in Visual Studio 2017 or 2019, you need to use remote debugging. To set it up, you need to configure the test device to accept debugging:

1. Go to Windows Update > For Developer.
2. Activate Developer Mode.
3. Activate Device Discover.
4. Pair your test device with your development machine.

Set up continuous integration

Add a Windows on Arm target to your continuous integration system.

Deploy your application

You can deploy the updated application in two ways:

- Upload to the Windows Store. The target must be a minimum Windows 10 version 1809 (10.0 build 17763). Older versions generate the error Package is invalid.
- Create an installer with NSIS 3.04 or newer, or other Windows on Arm compatible installation programs.

3. Tweeten application for the Electron framework

Building for Windows on Arm on Electron requires version 6.0.8 or newer. Tweeten is an example of an application that ran on an older version of Electron, and was upgraded to a new version of Electron and then ported to Windows on Arm.

Porting Tweeten

Tweeten was built on Electron 3 but building for Windows on Arm requires Electron 6.0.8 or newer. All the Tweeten code was upgraded to Electron 7.0. The Electron upgrade was the largest piece of work involved in porting Tweeten.

Tweeten was created on Visual Studio Code, so there was no need to change development environment.

After the Electron framework was upgraded, Tweeten had no third party or internal dependency issues. This is the case for many Electron apps, especially those that are almost entirely JavaScript.

Tweeten was tested on a Microsoft Surface Pro X. It was not necessary to set up the Surface Pro X for debugging. This is because Tweeten, like most Electron apps, is mainly JavaScript, and JavaScript can be adjusted while it runs.

Tweeten was uploaded to the Windows Store, because at the time NSIS had no Electron integration. Tweeten was initially uploaded with the wrong Windows 10 target and generated an error. When the target was changed, the upload worked.



NSIS 3.04 or newer integrates with Electron and electron-builder 22.0.0 or newer.

Comparing Tweeten performance on different platforms

Running Tweeten natively rather than on a Win32 emulator improves performance in the following ways:

- CPU usage on start-up: 20% on native, 30% on the emulator.
- CPU usage while running: 4% on native, 7% on the emulator.
- Overall CPU performance improvement of 33-42%, with accompanying battery savings.
- From a user perspective, apart from start-up time and window resizing, there is not much difference between the two modes. However, when running multiple applications in parallel to Tweeten, performance improvements and battery-life savings add up.

4. StaffPad application for the UWP framework

StaffPad is an example of an application that had third party and internal dependencies that required handling.

Porting StaffPad

StaffPad was built on Visual Studio 2019, which is set up for Windows on Arm by default.

StaffPad had a Fast Fourier Transform (FFT) library that relied on Intel Intrinsics for its audio engine. Although third-party libraries without Intel Intrinsics exist, the StaffPad engineers chose to write an architecture-independent internal library. Writing the library formed the bulk of the porting work.

StaffPad also had a few internal dependencies, but updating them took only two hours.

StaffPad's tested on a Surface Pro X, and exposed small, obvious errors that were easily fixed.

The team used Azure DevOps for continuous integration. Adding the Windows on Arm target was straightforward.

StaffPad was uploaded to the Windows Store, because it already integrates with UWP and StaffPad was already correctly set up for the store.

Comparing StaffPad performance on different platforms

Running StaffPad natively rather than on a win32 emulator improves performance. StaffPad is a processor-heavy application that offers rich graphics, audio, touch, machine learning, and networking features. On an emulator StaffPad was slow to start up, its memory usage was high, and the audio engine failed. Running natively, StaffPad was fast, responsive and fully functional, and could deliver the full user experience.

Overall, StaffPad running natively on Windows on Arm was faster and more responsive than other setups the team tested on.

5. Chromium Embedded Framework

In this section, learn about the options to author cross-platform Graphical User Interfaces (GUIs) in C++ and why [Chromium Embedded Framework](#) (CEF) can be a suitable choice for your application. Then, we build a small demonstration application using CEF targeting Windows on Arm. Finally, we show the workflows you need to deploy and debug the application. fPad is an example of an application that had third party and internal dependencies that required handling.

Setting up CEF on Windows on Arm

Because Visual Studio is not yet supported on Windows on Arm, we build and compile our sample project on a host x86_64 machine. We use Visual Studio remote tools to deploy and test our application on an Arm-powered Surface Pro X running Windows 10. For this guide, we use C++17.

Before you begin, install the Visual Studio ARM64 build tools as follows:

1. Open the Visual Studio Installer and click Modify for VS2019.
2. Select Desktop development with C++.
3. Select the MSVC v142 – VS 2019 C++ ARM64 build tools checkbox.
4. Click Modify.

To debug the application on the target Arm device, use [ARM Debugging Tools for Windows](#) or remotely debug your app from the host x86 machine using [Visual Studio remote debugging tools](#).

Then, integrate CEF as part of a [CMake](#) project using the following steps:

1. Create an empty directory for a CMake.
2. Download and extract the CEF Minimal Distribution binary release.
3. Rename the extracted folder to cef_arm64 and move it to the root directory of your project.
4. Download the Windows 64-bit minimal distribution binary to test the app on the host x86_64 machine.
5. Extract the binary to the root directory and rename the binary contents to cef_win64.
6. Paste the following code in an empty CMakeLists.txt file to set up the root CMake project file:

```
cmake_minimum_required(VERSION 3.19)
project(woa_cef LANGUAGES C CXX)

set(CMAKE_EXPORT_COMPILE_COMMANDS ON)
if(${CMAKE_GENERATOR_PLATFORM} MATCHES "arm64")
    set(CEF_ROOT ${CMAKE_CURRENT_SOURCE_DIR}/cef_arm64 CACHE INTERNAL "")
else()
    set(CEF_ROOT ${CMAKE_CURRENT_SOURCE_DIR}/cef_win64 CACHE INTERNAL "")
endif()

add_subdirectory(${CEF_ROOT})

add_executable(
    woa_cef
    # NOTE: This property is needed to use WinMain as the entry point
    WIN32
    app.cpp
)
```

```

    main.cpp
)

target_compile_features(
    woa_cef
    PRIVATE
    cxx_std_17
)

target_link_directories(
    woa_cef
    PRIVATE
    ${CEF_ROOT}/Release
)

target_link_libraries(
    woa_cef
    PRIVATE
    libcef
    cef_sandbox
    libcef_dll_wrapper
)

target_include_directories(
    woa_cef
    PRIVATE
    ${CEF_ROOT}
)

target_compile_definitions(
    woa_cef
    PRIVATE
    _ITERATOR_DEBUG_LEVEL=0
)

set_property(
    TARGET
    woa_cef
    PROPERTY
    MSVC_RUNTIME_LIBRARY "MultiThreaded$<$<CONFIG:Debug>:Debug>"
)

function(file_copy FILES BASE_DIR)
message(STATUS "FILES: ${FILES}")
foreach(FILE ${FILES})
    get_filename_component(FILE_DIR ${FILE} DIRECTORY)
    file(RELATIVE_PATH REL ${BASE_DIR} ${FILE_DIR})
    get_filename_component(FILE_NAME ${FILE} NAME)
    add_custom_target(
        copy_${REL}_${FILE_NAME}
        COMMAND ${CMAKE_COMMAND} -E echo ${CMAKE_CURRENT_BINARY_DIR}/${<IF:
$<CONFIG:Debug>,Debug,Release>/${REL}
        COMMAND ${CMAKE_COMMAND} -E copy ${FILE} ${CMAKE_CURRENT_BINARY_DIR}/${
$<IF:$<CONFIG:Debug>,Debug,Release>/${REL}/${FILE_NAME}
    )
    add_dependencies(woa_cef copy_${REL}_${FILE_NAME})
endforeach()
endfunction()

file(
    GLOB_RECURSE
    RESOURCE_FILES
    "${CEF_ROOT}/Resources/*"
)
message(STATUS "CEF resources: ${RESOURCE_FILES}")
file_copy("${RESOURCE_FILES}" ${CEF_ROOT}/Resources)

file(
    GLOB_RECURSE
    DLLS
    "${CEF_ROOT}/Release/*.dll"

```

```

)
message(STATUS "CEF dlls: ${DLLS}")
file_copy("${DLLS}" ${CEF_ROOT}/Release)

file(
  GLOB_RECURSE
  BINS
  "${CEF_ROOT}/Release/*.bin"
)
message(STATUS "CEF bins: ${BINS}")
file_copy("${BINS}" ${CEF_ROOT}/Release)

```

The project file creates a new executable called `woa_cef`, and the C++ source files `main.cpp` and `app.cpp`.

This CMake project file does the following:

1. To include and link CEF properly, set the `CEF_ROOT` variable to the appropriate directory. We can use this variable to switch between target architectures such as ARM64, x86, and x86_64. Then, we add the corresponding Release folder as a link directory.
2. Link the required libraries, `libcef`, `cef_sandbox`, and `libcef_dll_wrapper`. We compile the DLL wrapper as part of the project, with the first two libraries being precompiled libraries in the release.
3. All the CEF headers expect that the root release path is in the header include path, so we add that as an include directory. The last compilation setting changes specifically for Microsoft Visual C++ (MSVC) are `_ITERATOR_DEBUG_LEVEL` and `MSVC_RUNTIME_LIBRARY`. These settings must match the CEF-linked settings.
4. Use a small `file_copy` function to ensure runtime artifacts correctly copy to our binary directory as part of the build. We copy several DLLs that are needed for WebGL and other functions, locale data, V8 runtime snapshot binaries, and packed resources. This code allows us to create an executable that spawns the Arm Developer homepage in a CEF window and terminates the program on close.
5. Write our application interface. In `app.hpp`, insert the following code:

```

#pragma once
#include <include/cef_app.h>

class App : public CefApp, public CefBrowserProcessHandler
{
public:
    App() = default;

    CefRefPtr<CefBrowserProcessHandler> GetBrowserProcessHandler() override
    {
        return this;
    }

    void OnContextInitialized() override;

private:
    IMPLEMENT_REFCOUNTING(App);
};

```

The `App` class is a `CefApp`, which is the top-level C++ wrapper around the CEF C API. A typical CEF application has a browser process and one or more renderer processes, and `CefApp` can handle callbacks for both processes. This reflects Chromium's underlying [process-per-site-instance](#)

[model](#). This process handles general tasks like window creation and network access, and a separate, sandboxed renderer process for each site you visit. The renderer process is responsible for rendering HTML, CSS, and running JavaScript on the site.

The App class also inherits from [CefBrowserProcessHandler](#). CefBrowserProcessHandler includes methods you can override if you need to do work at specific points during the creation of the browser process. If you need to do any complex work before or after the browser process initializes, you can also create a separate browser process handler class that inherits from CefBrowserProcessHandler.

In this guide, we handle browser process events inherited from CefBrowserProcessHandler. In the body of the class declaration, we override the GetBrowserProcessHandler method to advertise this class as a browser process handler. We also override the OnContextInitialized browser event. In the private section of our App class decoration, use the IMPLEMENT_REFCOUNTING convenience macro to add boilerplate CEF needs on all classes that use ref-counting to manage lifetimes. The following snippet shows the code in the app.cpp file:

```
#include "app.hpp"
// A CefClient is an event handler that can optionally handle events corresponding
// to the various CEF handlers
// CefDisplayHandler handles events related to the browser display state
// CefLifeSpanHandler handles events related to the browser window lifetime
// CefLoadHandler handles events related to the browser loading status
class Handler : public CefClient, public CefLifeSpanHandler
{
public:
    Handler() = default;

    // Overrides on CefClient indicate that this class implements the following
    // handlers
    CefRefPtr<CefLifeSpanHandler> GetLifeSpanHandler() override { return this; }

    void OnBeforeClose(CefRefPtr<CefBrowser> browser) override
    {
        CefQuitMessageLoop();
    }

private:
    IMPLEMENT_REFCOUNTING(Handler);
};

static CefRefPtr<Handler> handler{new Handler};

void App::OnContextInitialized()
{
    CefBrowserSettings browser_settings;
    CefWindowInfo window_info;
    window_info.SetAsPopup(nullptr, "CEF Demo");

    CefBrowserHost::CreateBrowser(window_info, handler, "https://
developer.arm.com/", browser_settings, nullptr, nullptr);
}
```

While the App class handles events during the creation of the browser process, we must define an internal Handler class to handle browser events fired by the browser process once it is running.

We handle only the single onBeforeClose event to shut down the CEF event loop in this minimal application. Otherwise, after the user closes the window, our application still runs in the background.

We create a single window in this guide, so we create a single static handler. You can use a different handler per window in a more complicated scenario and coordinate handlers to determine when the app should shut down.

We spawn the window in the App class `onContextInitialized` implementation. This event occurs on the browser process UI thread immediately after the CEF context initializes.

Leaving the default `CefBrowserSettings`, we change the Windows classification to a Win32 [pop-up window style](#) with the title CEF Demo. Then, we use `CreateBrowser` to spawn a window with our specified settings to load the Google homepage.

The final step in this section is the `main.cpp` file, which initializes and starts the CEF context and event loop. The contents of the `main.cpp` source file is shown in the following code extract:

```
#include "app.hpp"
#include <Windows.h>

int APIENTRY WinMain(HINSTANCE instance, HINSTANCE previous_instance, LPTSTR args,
int command_show)
{
    CefMainArgs main_args(instance);

    int exit_code = CefExecuteProcess(main_args, nullptr, nullptr);
    if (exit_code >= 0)
    {
        return exit_code;
    }

    CefSettings settings;

    CefRefPtr<App> app(new App);

    CefInitialize(main_args, settings, app.get(), nullptr);

    CefRunMessageLoop();

    CefShutdown();

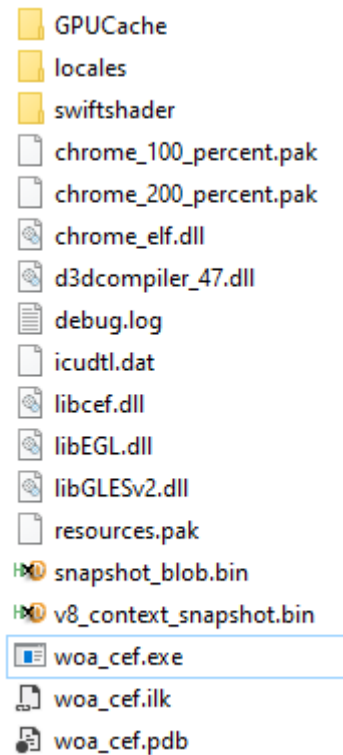
    return 0;
}
```

We use the [typical WinMain entry point](#) for GUI applications not expected to spawn a console. `CefExecuteProcess` runs CEF and `cefInitialize` spawns the browser process. By registering our App class as the CefApp in the browser, we receive the `onContextInitialized` event to generate the browser window. Finally, we run the message loop and tear everything down when the message loop finishes.

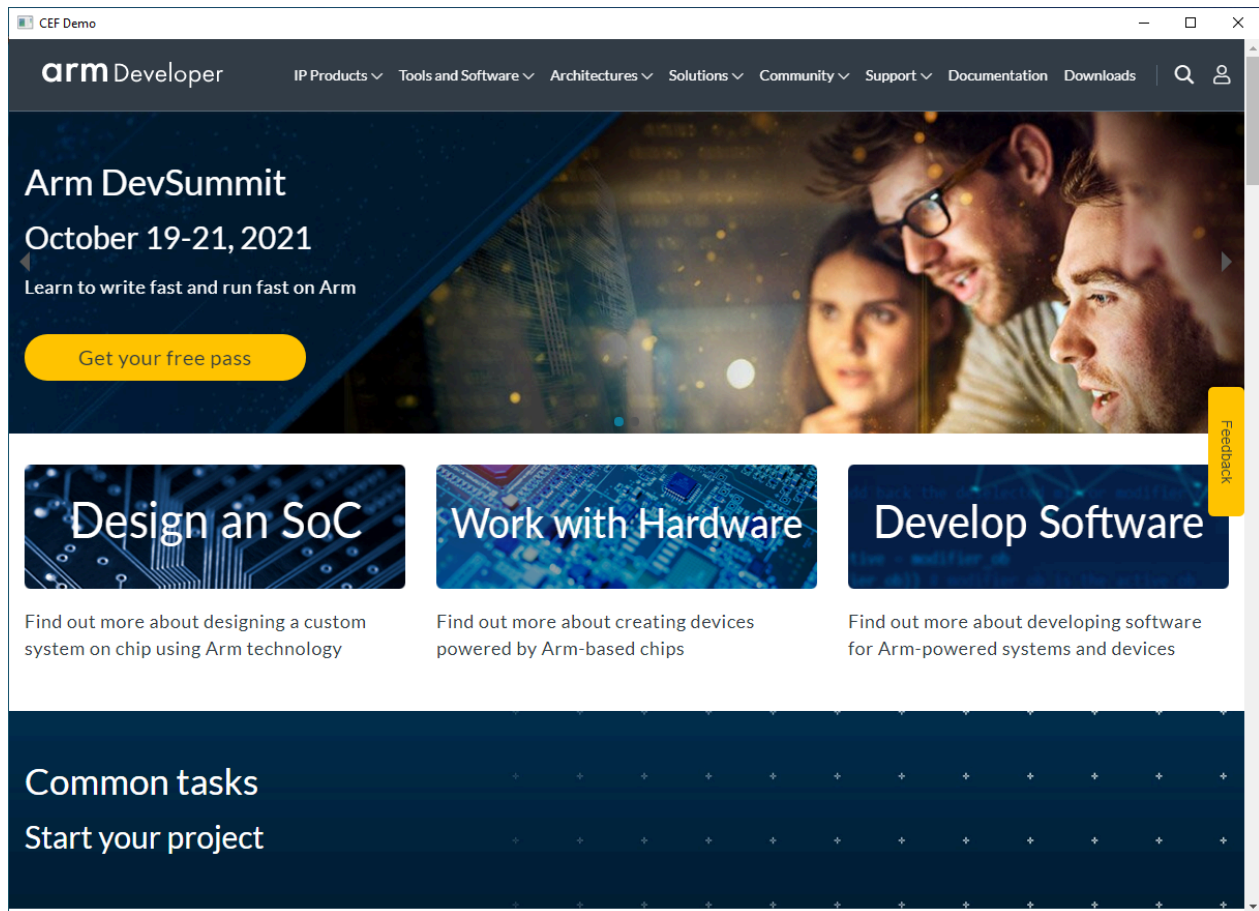
We verify our compiled demo works using the ARM64 MSVC CMake generator by running the following code:

```
<a>cmake -G "Visual Studio 16" -A arm64 -B build</a>
```

After the project is generated, open the `build_cef.sln` in Visual Studio 2019 and select Build > Build Solutions. After compiling, the `build/ARM64` directory folder will look like the following screenshot:

Figure 5-1: Build directory tree

Since this build is cross compiled for arm64 from an x86 machine, copy it to a Windows on Arm device to test it. Run the `woa_cef.exe` executable to display a window with the Arm Developer site. The following image shows the Arm Developer site running on a Windows on Arm device:

Figure 5-2: An example site running on Windows on Arm

To create an x64 build to try on your development machine, use the following command:

```
<a>cmake -G "Visual Studio 16" -A x64 -B build</a>
```

Open and build the solution file, and in the `build/x64` directory, run the `woa_cef.exe` file.



Note

Browser navigation will work, however the browser UI, such as menu bars and URL fields, will be missing. Click X to close this window and terminate the application.

In this section we left most of the default settings, however you can customize most aspects of the runtime and browser process in CEF. In addition to changing static settings, you can intercept other browser events when the page loads such as navigation or download. You can also intercept window events such as window move, focus, or minimize, and so on.

To learn more about the general architecture and use of CEF, read the [General Usage wiki](#). The source code for a demo using command-line argument parsing and multi-window scenarios can be found in the [main CEF repository](#).

6. WPF to UWP and .NET 4.8

This section shows you how to port a Windows .NET 4.8 Windows Presentation Foundation (WPF) application to a Windows on Arm (WoA) application. This guide uses a sample WPF application written using the C# programming language, and ports it to run on a WoA device. WoA can run almost every .NET application, whether they are compiled for Arm or not. This is because WoA uses an x86-emulation layer to execute programs that are not compiled for Arm. However, even though the emulation layer is optimized as far as possible, natively compiled applications usually run much faster than emulated applications. Benchmark tests run by Arm show that natively compiled applications can be up to 11 times faster. To learn more, see [How x86 emulation works on ARM](#) in the Microsoft Windows Developer documentation.

In the .NET 4.8 framework, it is not possible to compile a WPF application directly for ARM64. Therefore, we will migrate the WPF application to UWP, reusing as much of the WPF code as possible. By creating a new UWP project using .NET Framework 4.8 we can compile a native ARM64 application, which will run on an Arm device without emulation.

Later frameworks, for example .NET 6, include WPF for Arm, so it is possible to port these applications directly to Windows on Arm. For more information, see [WPF and .NET 6](#).

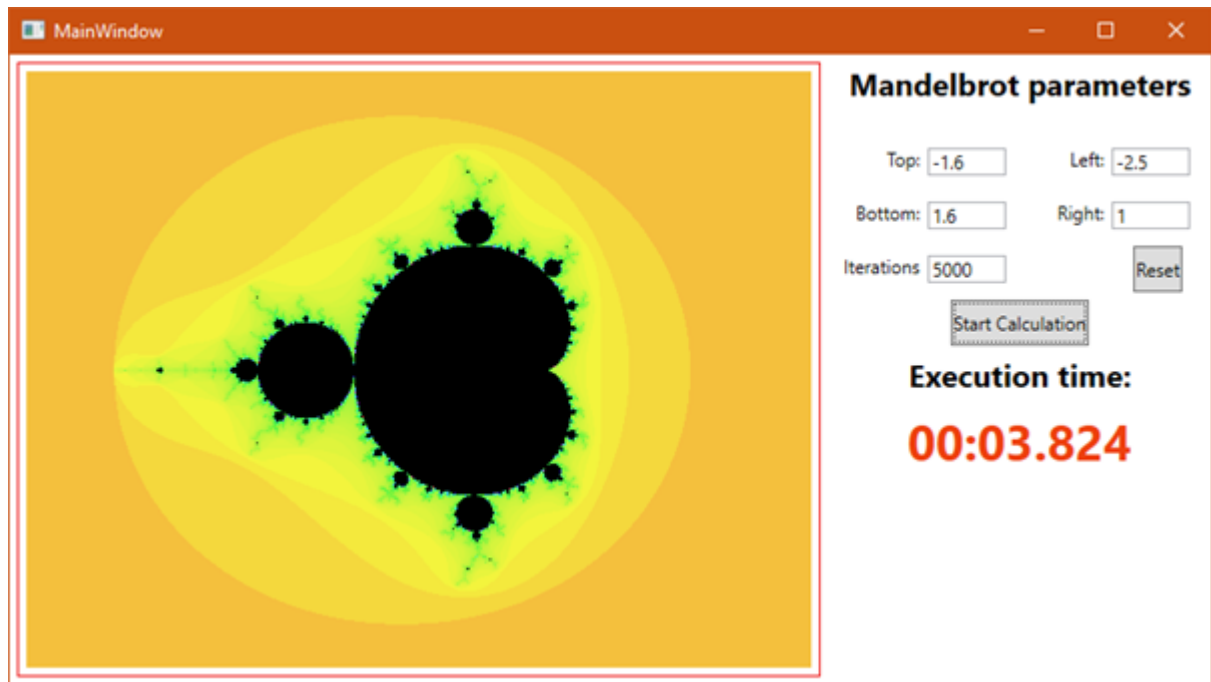
Before you begin

Before you begin this tutorial, you need the following tools and knowledge:

- Some knowledge of the C# programming language, WPF, UWP, and Extensible Application Markup Language (XAML) is required.
- [Visual Studio 2019](#), running on a host x86_64 machine. You should update Visual Studio to the latest version (version 16.11 or higher). The easiest way is to use the Visual Studio Installer.
- A Windows on Arm device, to run the ported application.

The sample application

This tutorial uses a sample WPF application that performs many floating-point calculations, so that we can measure the difference in performance. The sample application calculates and displays the [Mandelbrot set](#) using user-defined parameters, as shown in the following image:

Figure 6-1: The Madlebrot set

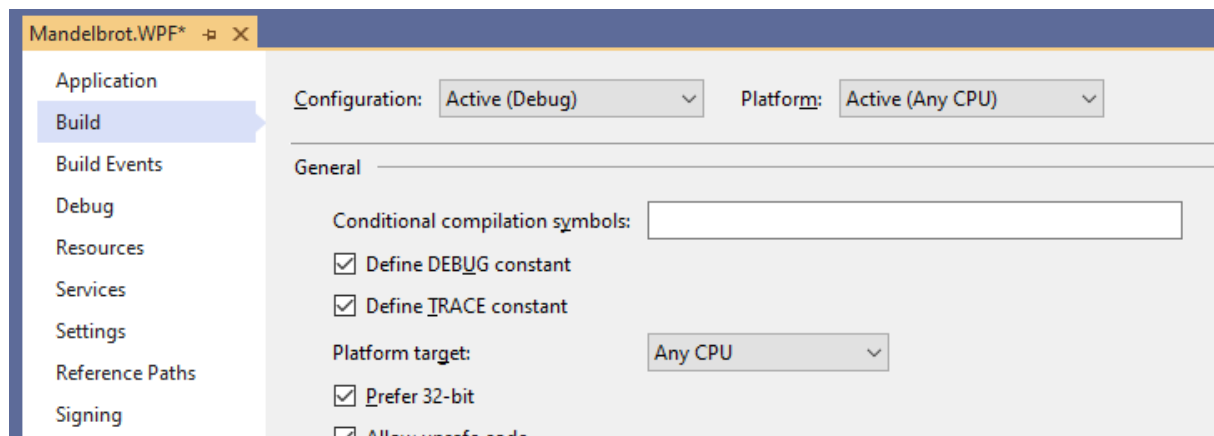
This tutorial does not discuss the implementation of the application or the mechanics of the Mandelbrot set itself, but you can examine the source code at [GVerelst/Mandelbrot](#). The important point for the purposes of this guide is that many floating-point calculations need to be performed to determine the color of each individual point in the final image.

Building the x86 WPF application

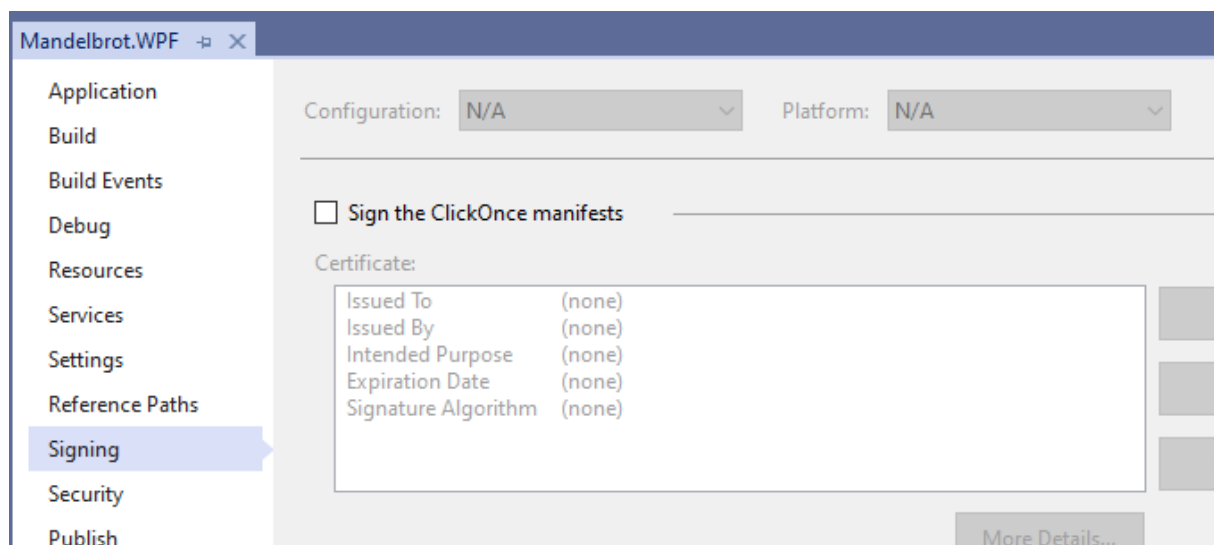
First, we will investigate the performance of the sample application running in emulation mode. To do this, we compile the application for x86 on the host machine then run the executable on the WoA device. Because the application was not specifically compiled for ARM64, it runs in x86 emulation mode.

To build the application, do the following:

1. At [GVerelst/Mandelbrot](#), click Code > Download ZIP and extract the downloaded ZIP file to a folder.
2. Double-click `Mandelbrot.sln` to open the source code in Visual Studio.
3. Set the build platform for the `Mandelbrot.WPF` project to Any CPU. Right-click the `Mandelbrot.WPF` project in Solution Explorer, click Properties, and set the Platform option on the Build tab to Any CPU, as shown in the following image:

Figure 6-2: Build menu

4. Also in the Properties window, clear the Sign the ClickOnce manifests option on the Signing tab, as shown in the following image:

Figure 6-3: Signing menu

5. In Solution Explorer, right-click the `Mandelbrot.WPF` project and select Build. The Output windows shows console output from the build process. Visual Studio compiles an x86 version of the application, `Mandelbrot.WPF\bin\Debug\ Mandelbrot.WPF.exe`.
6. Copy the entire `Mandelbrot.WPF\bin\Debug` folder to your Windows on Arm device and run the `Mandelbrot.WPF.exe` application. The x86 executable runs under emulation. This impacts performance, giving a slower execution time than expected with a native executable.

Porting the application to ARM64 UWP

It is not possible to compile a WPF application directly for ARM64 with the current .NET 5 Framework. Instead, we can use UWP to create an application that will compile for ARM64. We will create a new UWP project in our solution using .NET Framework 4.8. We can then publish this UWP application as a native ARM64 application, which runs on the target without emulation.

The UWP application reuses as much of the WPF code as possible.



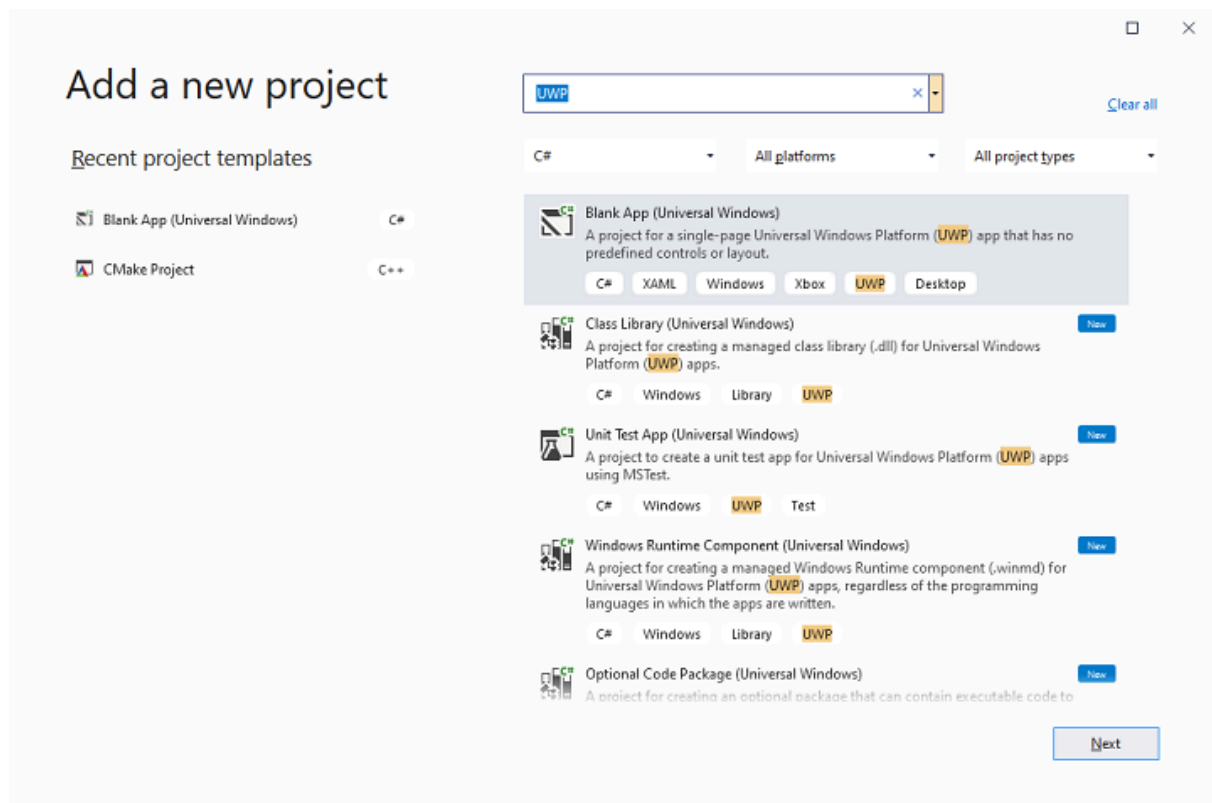
Note

The ZIP file you downloaded from at GVerelst/Mandelbrot already contains a ported UWP implementation, `Mandelbrot.UWP`. To follow this tutorial, we create a new application with the name `MyMandelbrot.UWP`. When you have finished this tutorial, `Mandelbrot.UWP` and `MyMandelbrot.UWP` should be identical, apart from the name differences.

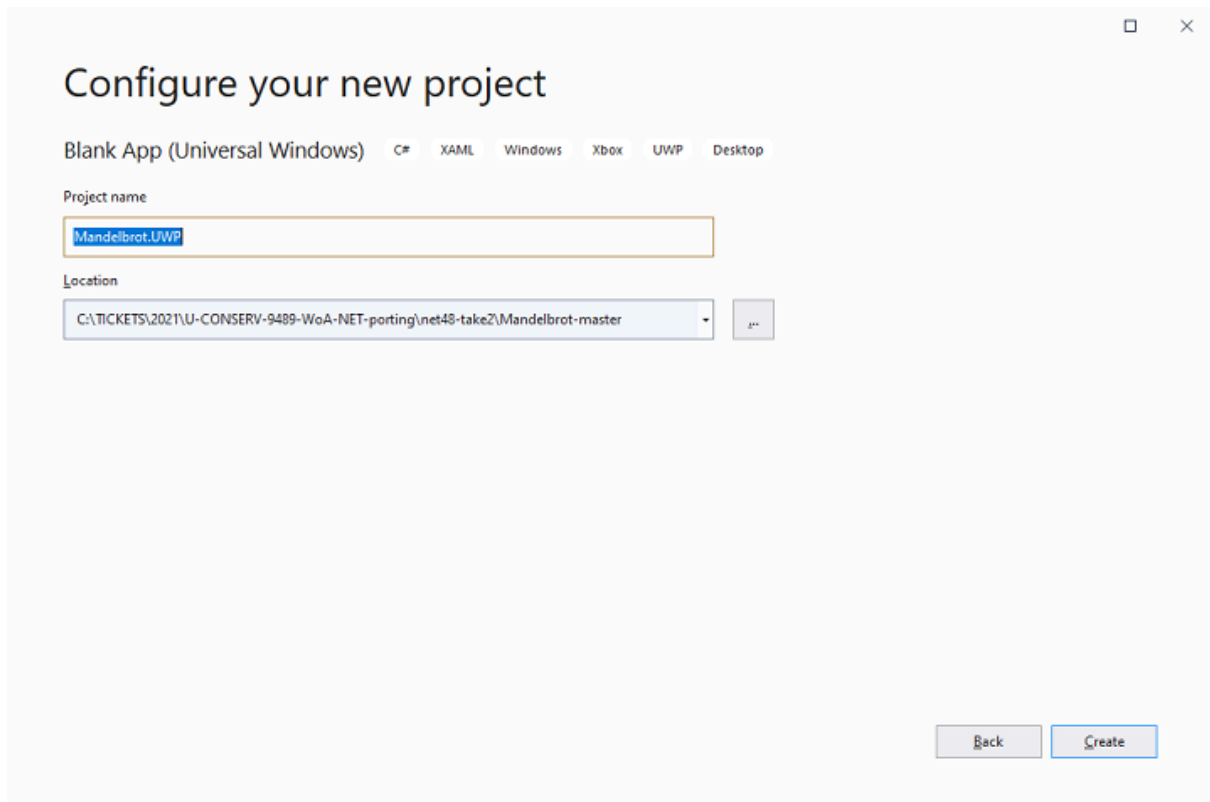
To port the WPF application to a new UWP application, do the following:

1. In Solution Explorer, right-click the top-level Mandelbrot solution object, then click Add > New object. The Add a new project dialog appears.
2. In the search box at the top of the window, type UWP to filter the list of available project types.
3. Select Blank App (Universal Windows). If there are multiple project types with this name, make sure to select the C# variant. The following screenshot shows a Blank App template:

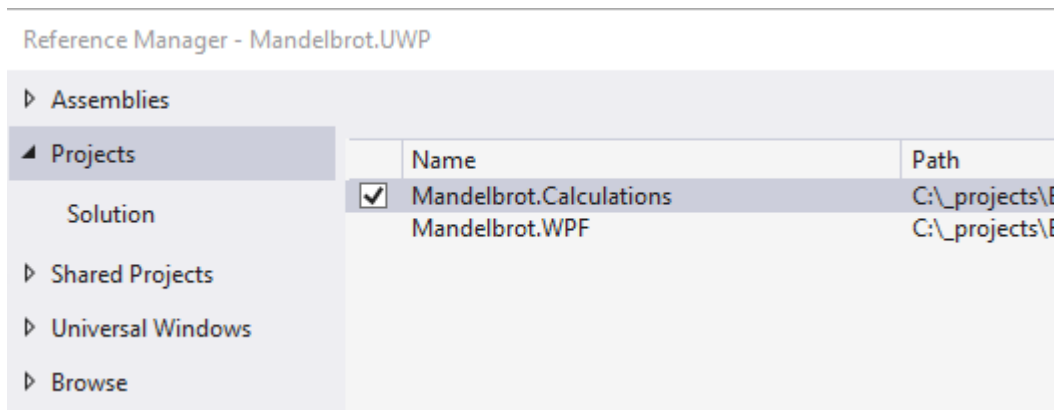
Figure 6-4: Blank App template



4. Click Next, give the project the name `MyMandelbrot.UWP`, keep the default location, and click Create. The following screenshot shows the new project configuration window:

Figure 6-5: New project configuration

5. In `MyMandelbrot.UWP`, reference the `Mandelbrot.Calculations` project by right-clicking References, then select Add reference... Select the `Mandelbrot.Calculations` checkbox to include the `Mandelbrot.Calculations` project, and click OK. The following screenshot shows the `Mandelbrot.Calculations` option:

Figure 6-6: Mandelbrot.Calculations option

6. Create a new folder called `viewModels` under the `MyMandelbrot.UWP` project. Right-click the project, then select Add > New Folder. Repeat to create another folder called `Extensions`.

7. Copy the file `MandelbrotParameters.cs` from `Mandelbrot.WPF/ViewModels` to `MyMandelbrot.UWP/ViewModels`.

8. Edit the file to change the following:

- a. Change the namespace from `Mandelbrot.WPF.ViewModels` to `MyMandelbrot.UWP.ViewModels` using the following code:

```
namespace MyMandelbrot.UWP.ViewModels
```

- b. The WPF application runs inside a WPF window, but the UWP application runs inside a page, so change the datatype as follows:

```
class MandelbrotParameters : INotifyPropertyChanged
{
    private readonly MainPage _window;
    public MandelbrotParameters(MainPage window)
    {
        window = window;
        Reset();
    }
    // ...
}
```

9. Copy the file `RelayCommand.cs` from `Mandelbrot.WPF/ViewModels` to `MyMandelbrot.UWP/viewModels`, editing the file to change the namespace from `Mandelbrot.WPF` to `MyMandelbrot.UWP` using the following code: `namespace MyMandelbrot.UWP`.

10. Edit the `MyMandelbrot.UWP/MainPage.xaml` file and make the following changes:

- a. Copy the top-level `<Grid>` element from `Mandelbrot.WPF/MainWindow.xaml` to `MyMandelbrot.UWP/MainPage.xaml`, replacing all `<Label>` elements with `<TextBlock>` elements as shown in the snippet:

```
// WPF Version
<Label Content="Top:" Grid.Row="1" Grid.Column="0" VerticalAlignment="Top"
Margin="0,0,0,5" HorizontalAlignment="Right" />
// UWP version
<TextBlock Grid.Row="1" Grid.Column="0" VerticalAlignment="Top"
Margin="0,0,0,5" HorizontalAlignment="Right">Top: </TextBlock>
```

- b. In UWP, data binding is by default `OneWay`. That means that if you set the value of a property bound to a control in code it will be reflected in the page, but if you change the value of the control in the GUI its value will not be sent back to the bound property. Change every `{Binding <dir>}` element in `MyMandelbrot.UWP/MainPage.xaml` to `{Binding <dir>, Mode=TwoWay}`. For example:

```
<TextBox HorizontalAlignment="Left" VerticalAlignment="Center" Width="51"
Grid.Row="1" Grid.Column="1" Margin="0,0,0,5" Text="{Binding Top,
Mode=TwoWay}" />
```

11. Edit `MyMandelbrot.UWP/MainPage.xaml.cs` and change the include files as follows:

```
using Mandelbrot.Calculations;
using MyMandelbrot.UWP.Extensions;
using MyMandelbrot.UWP.ViewModels;
using System.Collections.Generic;
using System.Drawing;
using Windows.UI.Xaml.Controls;
```

```
using Windows.UI.Xaml.Media.Imaging;
```

12. In the `MainPage` class, set `DataContext` to a new `MandelbrotParameters` instance. This change allows `viewModel` to call the `DrawImage` method.

```
public MainPage()
{
    this.InitializeComponent();
    DataContext = new MandelbrotParameters(this);
}
```

13. Copy the `DrawImage` and `GetImageViewport` methods from `Mandelbrot.WPF/MainWindow.xaml.cs` to `MyMandelbrot.UWP/MainPage.xaml.cs`. Here we see an example of the fact that the UWP APIs are not always the same as the WPF APIs. The `WriteableBitmap` constructor doesn't take six parameters, but only two: the width and the height. Remove the other parameters from the call to `WriteableBitmap` as follows:

```
WriteableBitmap wbmap = new WriteableBitmap(width, height);
```

14. Copy the file `WriteableBitmapExtensions.cs` from `Mandelbrot.WPF/Extensions` to `MyMandelbrot.UWP/Extensions`.
15. Edit the file to change the following:
- Change the namespace from `Mandelbrot.WPF.Extensions` to `MyMandelbrot.UWP.Extensions` using the code namespace `MyMandelbrot.UWP.Extensions`.
 - Change the include files as follows:

```
using System.Collections.Generic;
using System.IO;
using System.Runtime.InteropServices.WindowsRuntime;
using Windows.UI.Xaml.Media.Imaging;
```

- Replace the `setPixels` method as follows because the bitmap APIs in UWP are slightly different from their WPF counterparts:

```
public static void SetPixels(
    this WriteableBitmap wbm,
    IEnumerable<FractalPoint> pts)
{
    byte[] imageArray = new byte[(int)(wbm.PixelWidth * wbm.PixelHeight * 4)];
    int i = 0;
    foreach (var p in pts)
    {
        imageArray[i] = p.Color.B;
        imageArray[i + 1] = p.Color.G;
        imageArray[i + 2] = p.Color.R;
        imageArray[i + 3] = p.Color.A;

        i += 4;
    }
    using (Stream stream = wbm.PixelBuffer.AsStream())
    {
        //write to bitmap
        stream.Write(imageArray, 0, imageArray.Length);
    }
}
```

7. WinForms and .NET 5

This section shows you how to create a simple Windows on Arm (WoA) native Windows Forms application.

WoA can run x86 Windows applications whether they are compiled for Arm or not. This is because WoA uses an x86-emulation layer to execute programs that are not compiled for Arm. Emulation translates the x86 processor instructions to Arm instructions.

Even though the emulation layer is optimized as far as possible, natively compiled applications usually run much faster than emulated applications. To take advantage of your WoA device's power and maximize battery life, consider creating native applications.

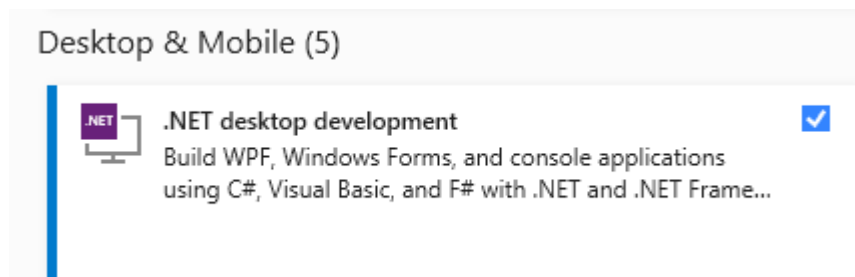
There are several different approaches you can take to create a native application for WoA devices. For example, you can use [WPF and .NET 6](#) or UWP with earlier .NET releases. The approach used in this section is to look at an existing Windows Forms (WinForms) application developed on an x86-64 machine and port it to a native WoA WinForms application.

Before you begin

Before you begin this tutorial, you need the following tools and knowledge:

- Some knowledge of the C# programming language, .NET, and WinForms is required.
- [Visual Studio 2019](#), running on a host x86_64 machine. You should update Visual Studio to the latest version (version 16.11 or higher). The easiest way is to use the Visual Studio Installer. Install the .NET desktop development component using the Visual Studio installer, as shown in the following screenshot:

Figure 7-1: .NET desktop development option



- Windows on Arm device, to run the ported application.

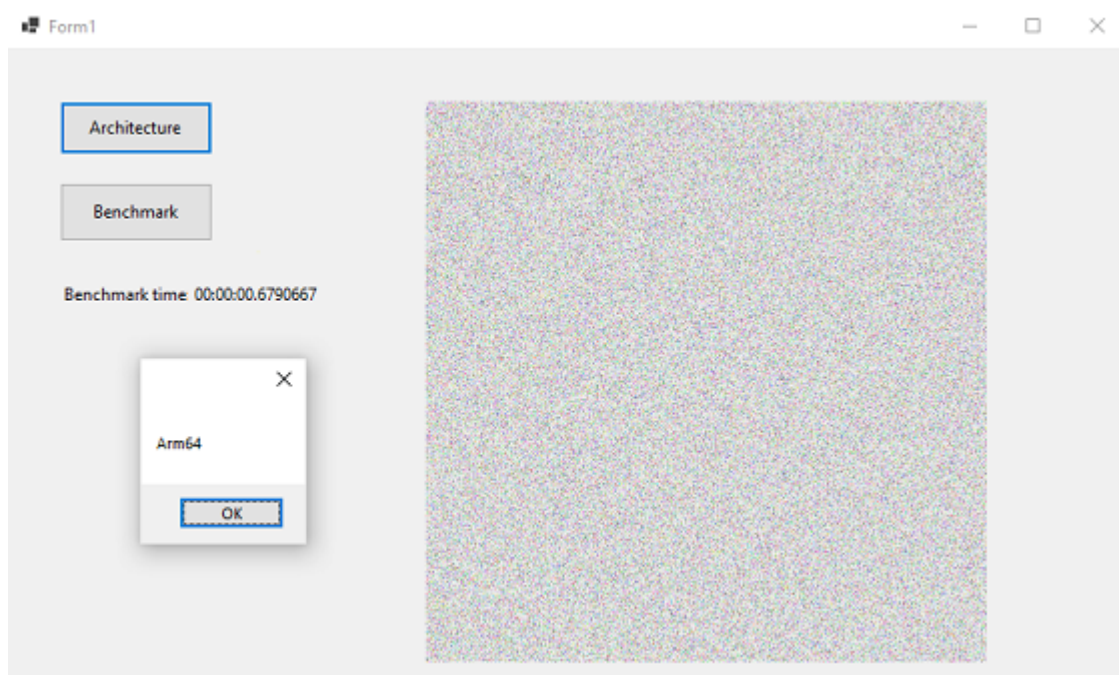
The sample application

This guide uses a simple project to demonstrate the process of porting a WinForms application from x86 to ARM64. The application has two buttons:

- When you click Architecture, the application displays the target device's architecture information, either Arm64 (when running on the WoA device) or x86 (when running on the x86 host computer).

- When you click Benchmark, the application generates a 400 by 400 bitmap, fills it with random pixels, and displays both the resulting bitmap and the time taken, as shown in the following screenshot:

Figure 7-2: Benchmark example

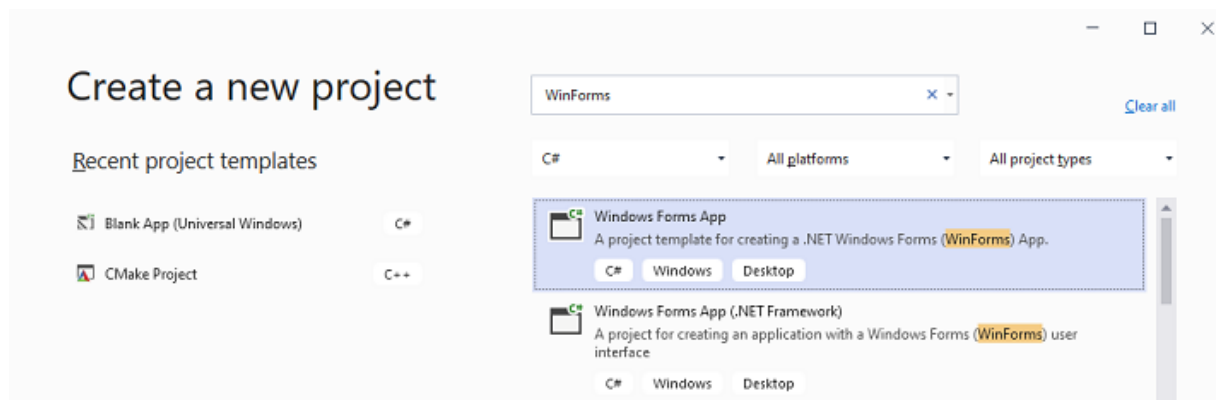


Creating the WinForms application

To create the sample Windows Forms project, do the following:

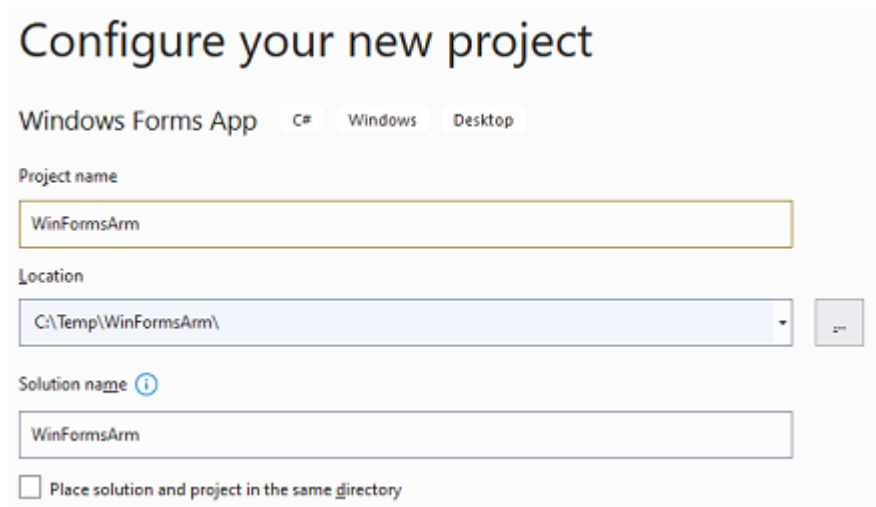
- Open Visual Studio and click Create a new project.
- Click Windows Forms App in the template list, then click Next. You can type `winForms` in the search box to help find the template if needed. Do not click Windows Forms App (.NET Framework) because that is for .NET Framework 4.x, rather than .NET 5. The following screenshot shows the Create a new project window:

Figure 7-3: Benchmark example



3. Choose a Project name and Location for the application, and click Next. The new project configuration is shown in the screenshot:

Figure 7-4: Configure your new project window



Configure your new project

Windows Forms App C# Windows Desktop

Project name

WinFormsArm

Location

C:\Temp\WinFormsArm\

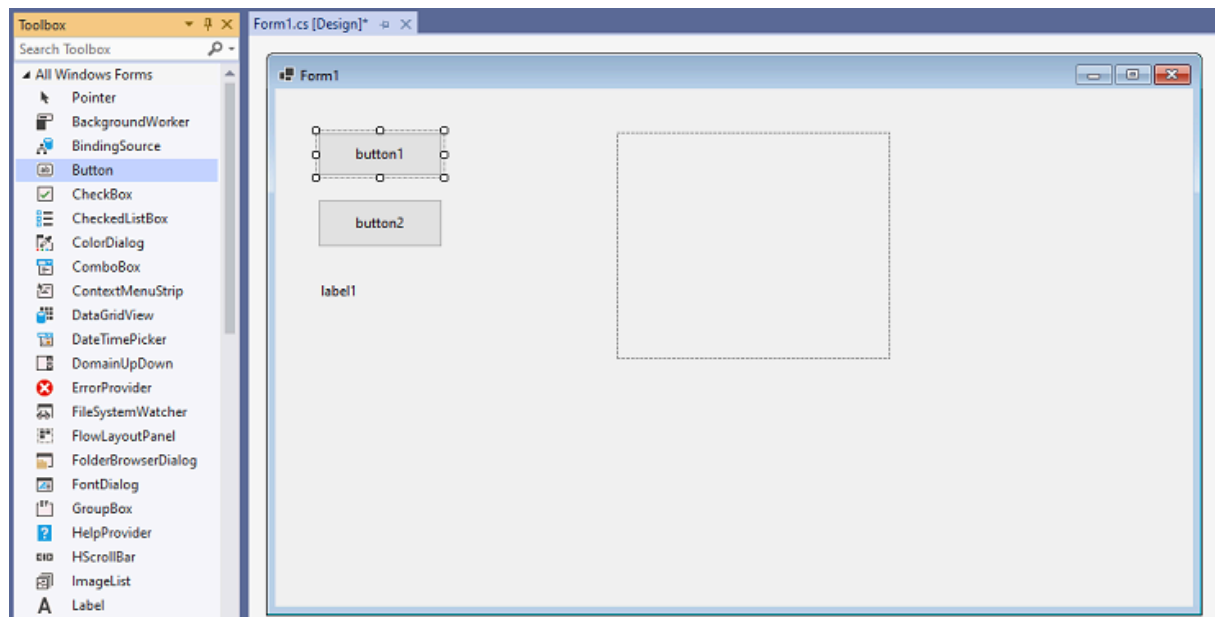
Solution name

WinFormsArm

☐ Place solution and project in the same directory

4. Select .NET 5 in the Target Framework, then click Create. Both the .NET Core 3.1 and .NET 5 frameworks allow us to compile to AArch64, however performance is better with .NET 5.
5. Drag the following controls from the Toolbox pane to your form: two Button controls, one Label control, one PictureBox control. The form now looks like the following screenshot:

Figure 7-5: Form example

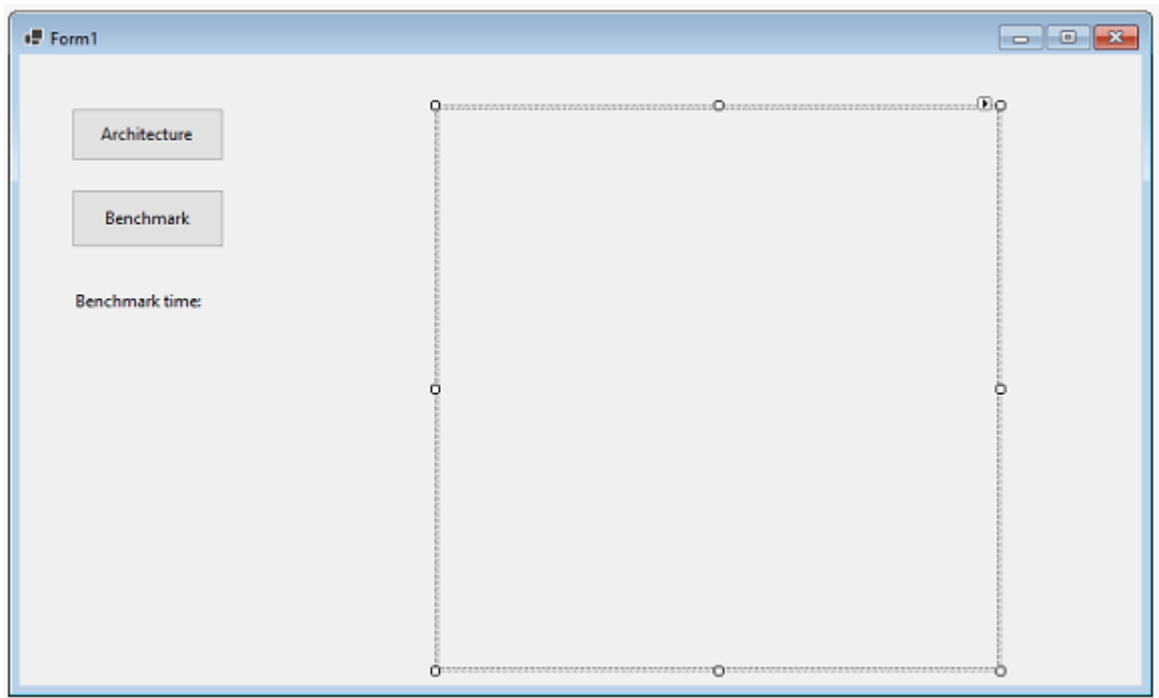


6. Use the Properties pane to set the following object properties:

Object	Property	Value
button1	(Name)	archBtn
button1	Text	Architecture
button2	(Name)	benchmarkBtn
button2	Text	benchmarkBtn
label1	(Name)	benchmarkBtn
label1	Text	Benchmark time:
pictureBox1	(Name)	benchmarkPbox
pictureBox1	Size > Width	400
pictureBox1	Size > Height	400

- When porting an application for Windows on Arm, consider the Scale and layout Windows setting. Your target, like a Surface Pro, might have a high resolution on a small screen with a value of 200%. Or, the desktop development machine has resolution set between 100% and 150%. The Scale and layout setting affects text and form size but controls like PictureBox do not scale. Ensure your application looks correct on all target platforms, including Windows on Arm devices. The form now looks like the following screenshot:

Figure 7-6: Form results



- Double-click the archBtn button to add a Click event handler with the following code:

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show (
```

```
System.Runtime.InteropServices.  
    RuntimeInformation.ProcessArchitecture.ToString());  
}
```

9. Double-click the benchmarkBtn button to add a Click event handler with the following code:

```
private void benchmarkBtn_Click(object sender, EventArgs e)
{
    Stopwatch sw = new Stopwatch();
    sw.Start();

    Random rng = new Random();
    Bitmap bmp = new Bitmap(400, 400);

    for (int i = 0; i < 10; i++)
    {
        for (int x = 0; x < 400; x++)
        {
            for (int y = 0; y < 400; y++)
            {
                bmp.SetPixel(x, y, Color.FromArgb(rng.Next()));
            }
        }
    }
    benchmarkPbox.Image = bmp;
    sw.Stop();
    benchmarkLabel.Text = "Benchmark time: " + sw.Elapsed.ToString();
}
```

10. Add the `System.Diagnostics` include to the top of the `WinFormsArm\Form1.cs` code file, to allow us to use the `Stopwatch()` timer function:

```
using System.Diagnostics;
```

The benchmark uses `SetPixel` to fill one pixel at a time. This is an inefficient way to generate a random image. However, it does not matter for a benchmark of this kind because results are only significant when compared to each other. The application repeats the process of generating a random image ten times to average the results for each iteration. When the picture finishes generating, the application displays the elapsed time on the label.

Building the x86 and ARM64 WinForms applications

To test the performance of both the emulated x86 application and the native ARM64 application, build the application twice: once for each platform.

To build the application, do the following:

1. Select Build > Configuration Manager to display the Configuration Manager dialog.
2. For Active solution configuration, select Release then click Close.
3. In Solution Explorer, right-click the WinFormsArm project and select Properties to display the Properties pane.
4. For Platform target, select x86 then press Ctrl+S to save.
5. In Solution Explorer, right-click the WinFormsArm project and select Build to compile the executable.

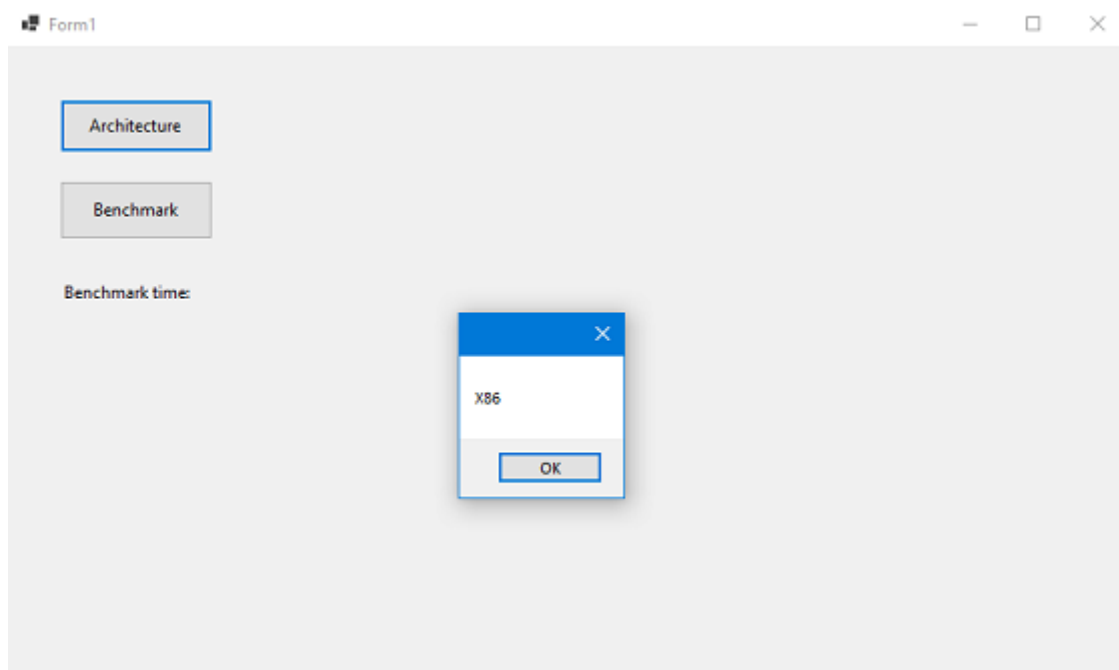
6. Copy the contents of the build directory `Release\net5.0-windows` to a new folder, named so that is identifiable as the x86 build, for example `C:\Temp\x86WinFormsArm`.
7. In Solution Explorer, right-click the WinFormsArm project and select Properties to display the Properties pane.
8. For Platform target, select ARM64 then press Ctrl+S to save.
9. In Solution Explorer, right-click the WinFormsArm project and select Build to compile the executable.
10. Copy the contents of the build directory `Release\net5.0-windows` to a new folder, named so that is identifiable as the ARM64 build, for example `C:\Temp\ARM64WinFormsArm`.

Running the x86 application

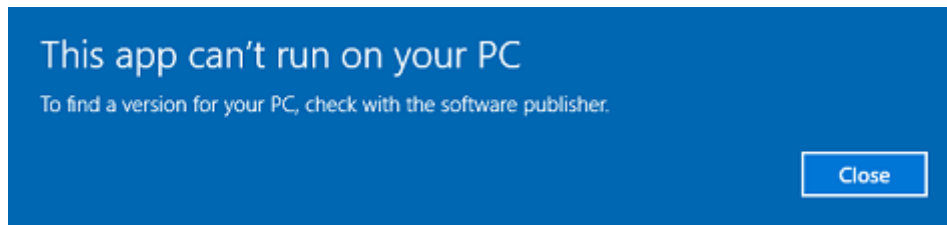
We can now test the executables we have built by running both on the host x86 computer.

The x86WinFormsArm application runs correctly, reporting the architecture x86, when you click Architecture, as shown:

Figure 7-7: Form application



The ARM64 application reports the following error because you cannot run ARM64 binaries on an x86 device:

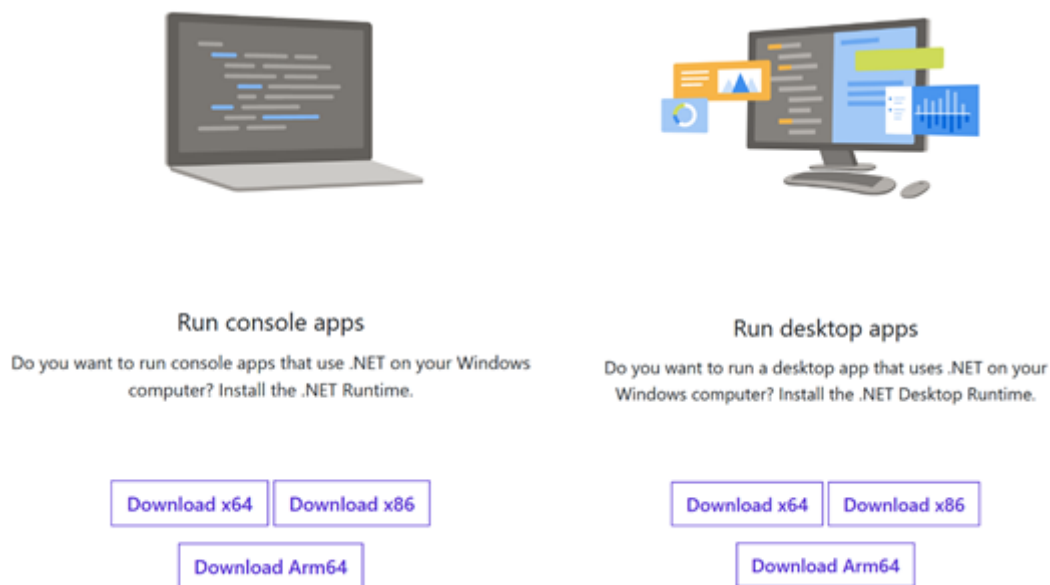
Figure 7-8: Error message

Running the ARM64 application

Now we can run both the x86 and ARM64 executables on the ARM64 device to compare performance.

Copy the build folders `ARM64WinFormsArm` and `x86WinFormsArm` you created to the target device, for example using a USB stick.

When you run the applications, if you do not have the .NET runtime installed yet, the device prompts you to install and redirects you to the download page. On the download page, select Download Arm64 in Run console apps, and run the installer. Then do the same for Run desktop apps. The following image shows the installation prompts:

Figure 7-9: Installation prompts

The `ARM64WinFormsArm` application runs natively. When you click Architecture, the application reports the architecture `Arm64`, and reports the benchmark execution time when you click Benchmark.

The `x86WinFormsArm` application runs in emulation mode and reports a slower benchmark time. This shows that the native-compiled application is more efficient.

8. WPF and .NET 6

This section shows how to port a Windows .NET 6 Windows Presentation Foundation (WPF) application to a Windows on Arm (WoA) application. This guide uses a sample WPF application written using the C# programming language, and ports it to run on a WoA device.

WoA can run almost every .NET application, whether they are compiled for Arm or not. This is because WoA uses an x86-emulation layer to execute programs that are not compiled for Arm. However, even though the emulation layer is optimized as far as possible, natively compiled applications usually run much faster than emulated applications. Benchmark tests run by Arm show that natively compiled applications can be up to 11 times faster. To learn more, see [How x86 emulation works on ARM](#) in the Microsoft Windows Developer documentation.

.NET 6 includes WPF for Arm, so this tutorial shows how to port a Windows .NET 6 WPF application to Windows on Arm (WoA). This requires configuring the development environment and making some changes to the application so that it can run natively on WoA without the emulator.

For earlier frameworks, for example .NET 5, it is possible to port applications by creating a UWP application that runs on WoA in native mode.

Before you begin

Before you begin this tutorial, you need the following tools and knowledge:

- Some knowledge of the C# programming language, WPF, and Extensible Application Markup Language (XAML) is required.
- [Visual Studio 2022](#), running on a host x86_64 machine.



.NET 6 is only supported with Visual Studio 2022. If you want to use .NET 6, and you are using an earlier version of Visual Studio, you will need to upgrade to Visual Studio 2022.

Update Visual Studio to the latest version using the Visual Studio Installer. * Knowledge about how to create projects, classes, and WPF forms in Visual Studio. * A Windows on Arm device, to run the ported application.

The sample application

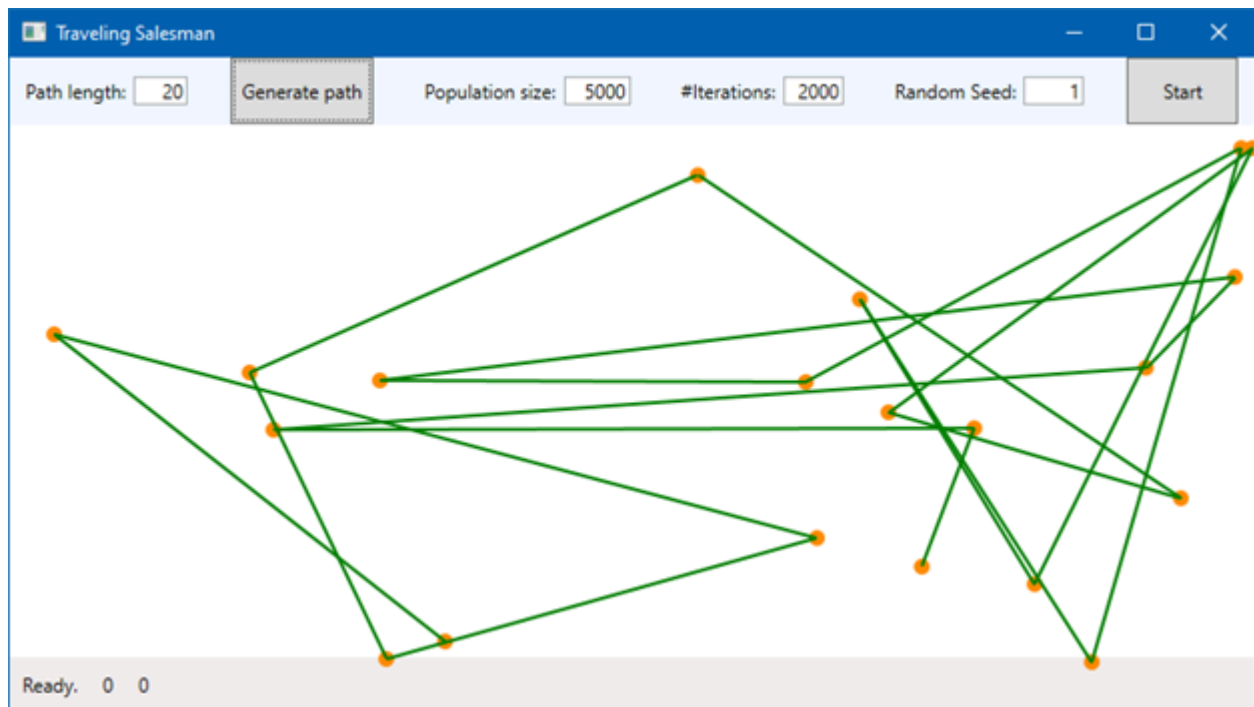
This tutorial uses a sample application that solves a well-known algorithmic problem: the [Travelling Salesman Problem \(TSP\)](#).

The purpose of the sample application is let us perform speed benchmarks between the emulated version and the native .NET 6 WPF on AArch64. This sample application has therefore been chosen because it is reasonably CPU intensive, and makes use of graphics to test the capabilities of WPF in .NET 6.

Solving the TSP using brute is impractical even for relatively small numbers of points. The brute force approach calculates all possible paths between all points. That is, to calculate all the possible paths between N points requires calculating $N!$ paths. This is feasible for problems with a very small number of points, but quickly becomes impractically slow. For example, finding the optimal path between 20 points would require $20!$ (2,432,902,008,176,640,000) calculations. Even on a very fast computer, this takes a very long time.

We need another solution. The sample application implements a genetic algorithm (GA) to solve the TSP. This tutorial does not describe the implementation of the application, but you can examine the source code at [GVerelst/TravelingSalesman](#).

Figure 8-1: Example of the Travelling Salesman application



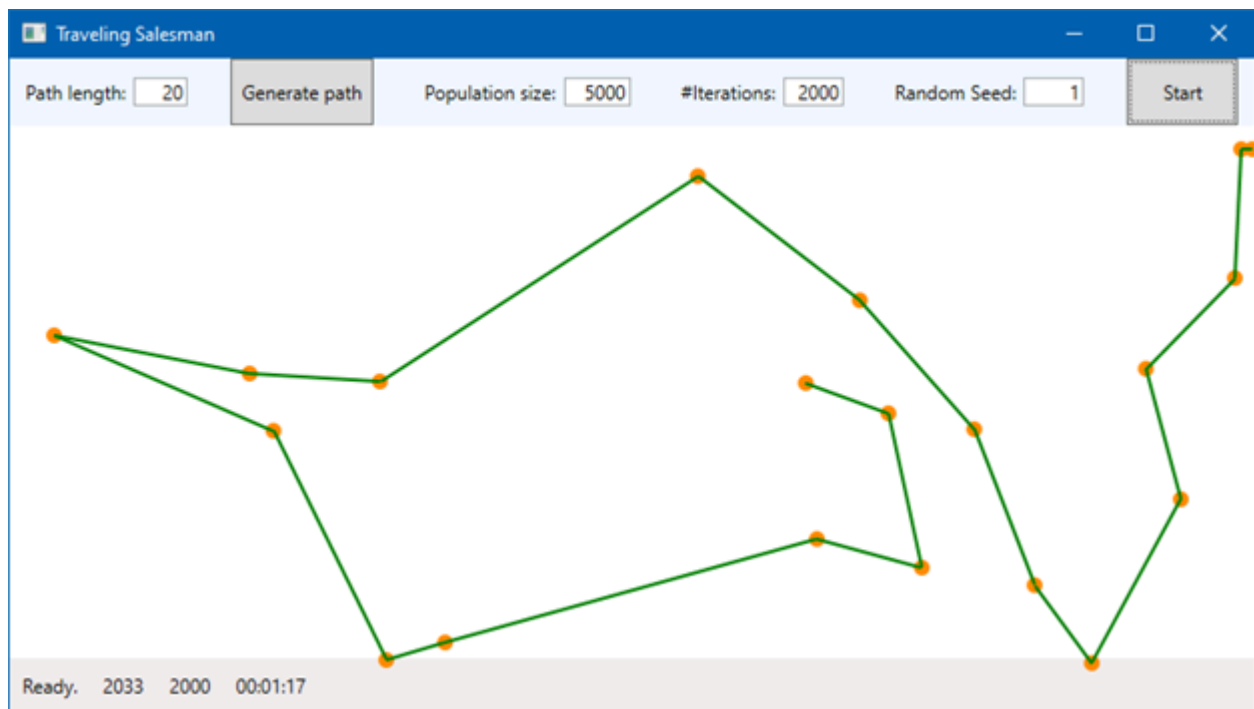
The sample application has the following functionality:

- Generate Path generates a new random path, using the following settings:
 - Path length specifies the number of points that are generated. The default is 20.
 - Population size determines the length of the population for the genetic algorithm. The default is 5,000.
 - Random seed is used for the pseudo random generator. Starting from the same number will always generate the same sequence of numbers. Using the same random seed for different runs helps us compare run times. The default is 1.
 - Iterations specifies how many times the algorithm runs. The default is 2000.
- For each iteration of the algorithm:
 - The distance of 5,000 paths is calculated.

- The 5,000 paths are ordered by their length.
- 5,000 new paths are generated using crossover and mutation algorithms.
- The best path is displayed on the canvas.

Here is the result of the algorithm after 1,000,000 calculations:

Figure 8-2: Sample application



The TSP sample application source code is configured to build and compile using .NET 5. The resulting executable will not run on an Arm device, because the .NET 5 framework for Arm does not include WPF. We must port the application to .NET 6.

Port the application on the host machine

WPF is implemented in the .NET 6 framework for Arm and later.

Because Visual Studio is not yet supported on Windows on Arm, we build and compile our sample project on a host x86_64 machine, then transfer the executable to a WoA device to run it.

To port the sample application to Windows on Arm, do the following on the x86_64 host machine:

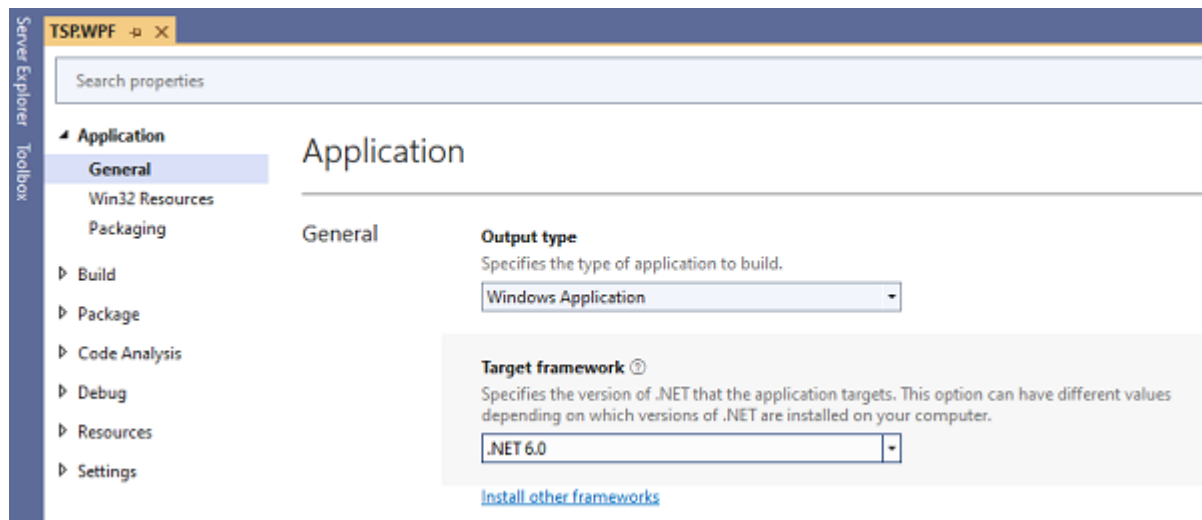
1. Install the [.NET 6 SDK](#).
2. Download the sample application source code from [GVerelst/TravelingSalesman](#) by clicking Code > Download ZIP and extracting the downloaded ZIP file to a folder.

3. In the source code folder, edit the file `TSP.WPF/TSP.WPF.csproj` and add the `<RuntimeIdentifiers>` element as follows:

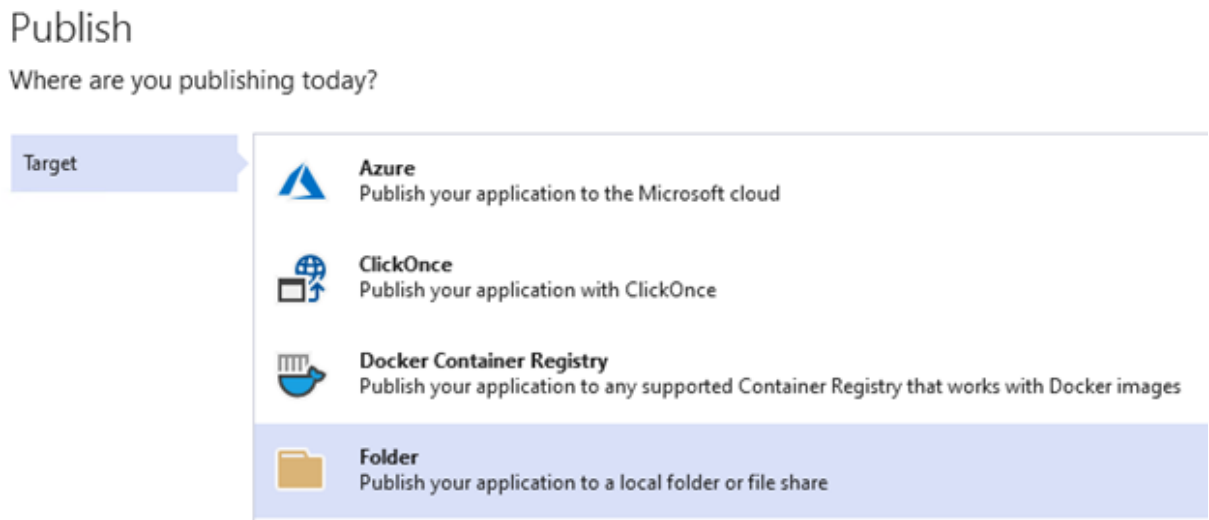
```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net6.0-windows</TargetFramework>
    <RuntimeIdentifiers>win-x64;win-arm64</RuntimeIdentifiers>
    <UseWPF>true</UseWPF>
  </PropertyGroup>
  <ItemGroup>
    <ProjectReference Include="..\TSP.GA\TSP.GA.csproj" />
  </ItemGroup>
</Project>
```

4. Double-click `TSP.sln` to open the source code in Visual Studio.
5. For both the `TSP.WPF` and the `TSP.GA` projects, and optionally for the Tests project, set the target framework to .NET 6.0. Right-click the project in Solution Explorer, click Properties, and set the Target framework option on the Application > General tab, as shown in the following image:

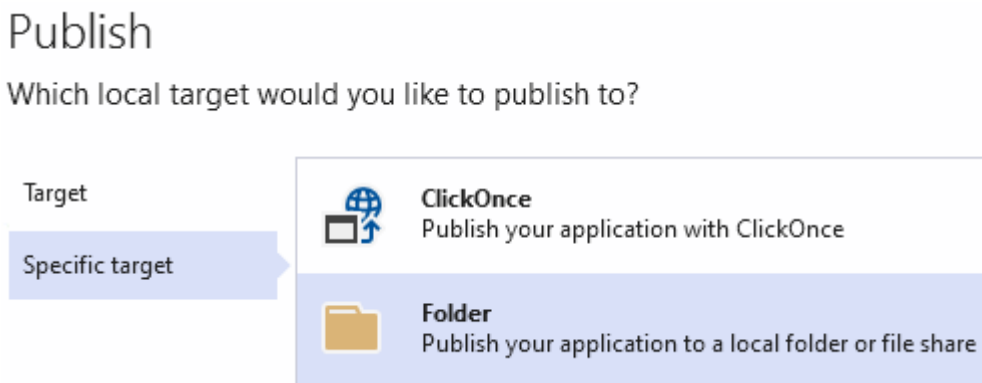
Figure 8-3: Application page



6. In Solution Explorer, right-click the `TSP.WPF` project and select Publish. The Publish dialog appears.
7. Under Target, click Folder then click Next, as shown in the following image:

Figure 8-4: Publish page

- Under Specific target, click Folder then click Next, as shown in the following image:

Figure 8-5: Publish settings for the specific target

- Select a location to store the generated binaries.
- Click Finish to create the initial publish profile.
- Click Show all settings to display the Profile settings dialog, set the values as shown in the following image, and click Save:

Figure 8-6: Profile settings

Profile settings

Profile name FolderProfile

Configuration Release | Any CPU

Target framework net6.0-windows

Deployment mode Framework-dependent

Target runtime win-arm64

Target location bin\Release\net6.0-windows\publish\ ...









File publish options

Save Cancel

12. Click Publish. Visual Studio compiles a version of the project that is natively executable on Aarch64 devices, called TSP.WPF.exe:

Figure 8-7: TSP.WPF.exe file location

app > TravelingSalesman-master > TSP.WPF > bin > Release > net6.0-windows > win-arm64 >

<input type="checkbox"/> Name	Date modified	Type	Size
 ref	18/10/2021 16:59	File folder	
 TSP.GA.dll	18/10/2021 16:58	Application exten...	8 KB
 TSP.GA.pdb	18/10/2021 16:58	Program Debug D...	12 KB
 TSP.WPF.deps.json	18/10/2021 16:59	JSON Source File	1 KB
 TSP.WPF.dll	18/10/2021 16:59	Application exten...	16 KB
<input checked="" type="checkbox"/>  TSP.WPF.exe	18/10/2021 16:59	Application	140 KB
 TSP.WPF.pdb	18/10/2021 16:59	Program Debug D...	16 KB
 TSP.WPF.runtimeconfig.json	18/10/2021 16:59	JSON Source File	1 KB

Run the application on the WoA device









Now that we have built the sample application for Windows on Arm, we are ready to run the executable.

To run the application, do the following on your WoA device:

1. Copy the entire `win-arm64` folder from your `x86_64` host machine to your Windows on Arm device.
2. Double-click `TSP.WPF.exe` to run the application. The .NET Desktop Runtime is required to run the application. If you have not already installed the .NET 6 Desktop Runtime, you are prompted to install it as shown in the following image:

Figure 8-8: Runtime error

app > TravelingSalesman-master > TSP.WPF > bin > Release > net6.0-windows > win-arm64 >

<input type="checkbox"/> Name	Date modified	Type	Size
 ref	18/10/2021 16:59	File folder	
 TSP.GA.dll	18/10/2021 16:58	Application exten...	8 KB
 TSP.GA.pdb	18/10/2021 16:58	Program Debug D...	12 KB
 TSP.WPF.deps.json	18/10/2021 16:59	JSON Source File	1 KB
 TSP.WPF.dll	18/10/2021 16:59	Application exten...	16 KB
<input checked="" type="checkbox"/>  TSP.WPF.exe	18/10/2021 16:59	Application	140 KB
 TSP.WPF.pdb	18/10/2021 16:59	Program Debug D...	16 KB
 TSP.WPF.runtimeconfig.json	18/10/2021 16:59	JSON Source File	1 KB

3. Click Yes to install the .NET Desktop Runtime. The application starts.

Conclusion

Comparing the execution times for both the original sample application running on x86 and the ported application running on Windows on Arm shows similar results. Also the ported application runs 1.5 times faster than the emulated application for Windows on Arm.

The porting process to take an existing x86/x64 WPF application and run it natively on an AArch64-based Windows machine was relatively straightforward. We needed to make several configuration changes to build the project for WPF on .NET 6, but no code changes were needed.

You will need to use caution if your WPF application calls native C or C++ libraries with PInvoke or uses NuGet packages that wrap C and C++ libraries. In these situations, you will need AArch64 builds of these libraries to run your application natively.

For more information about .NET 6, see [Announcing .NET 6 – The Fastest .NET Yet](#).

9. Related information

Here are some resources related to material in this guide:

- [Arm community](#)
- [Electron framework](#)
- [StaffPad](#)
- [Tweeten](#)
- [UWP framework](#)
- [Windows on Arm frameworks](#)