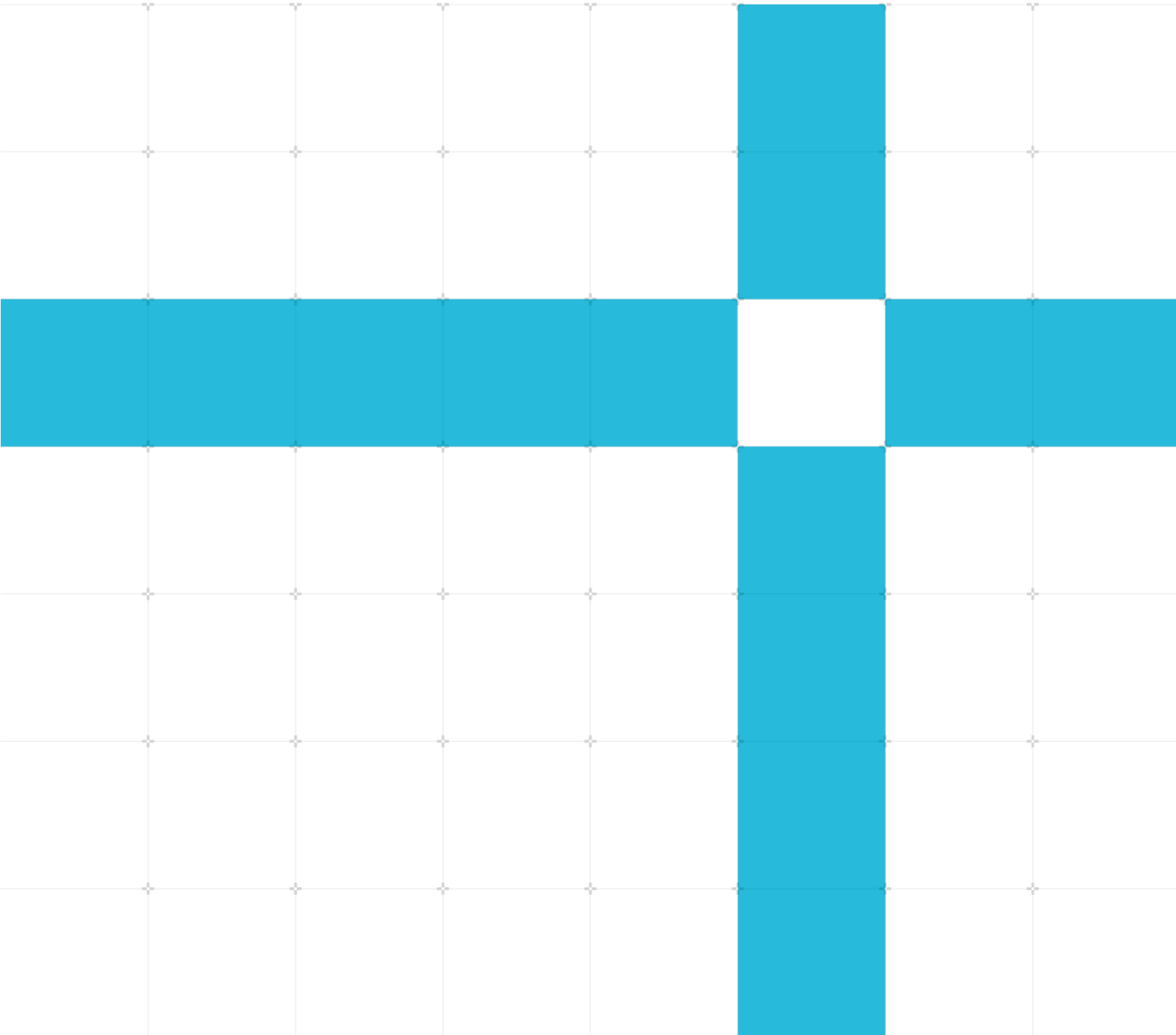
arm

Port applications to Windows on Arm

Non-Confidential



Port applications to Windows on Arm

Release information

Document history

Issue	Date	Confidentiality	Change
01	12 January 2021	Non-Confidential	First release
02	20 January 2021	Non-Confidential	Correction to testing instructions and editorial review

Non-Confidential Proprietary Notice

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.

Web Address

developer.arm.com

Progressive terminology commitment

We believe that this document contains no offensive terms. If you find offensive terms in this document, please email **terms@arm.com**.

Contents

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, and the Universal Windows Platform (UWP) porting for the StaffPad application.

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 the **General porting instructions** section.

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.

2.1 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

2.2 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.

2.3 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` of `#ifdef {PLATFORM}`.

2.4 Set compilation target and perform a build

Set the compilation target to Arm64 and perform a build.

2.5 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 Discovery**.
4. Pair your test device with your development machine.

2.6 Set up continuous integration

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

2.7 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.

3.1 Porting Tweeten

Tweeten was built on Electron 3 but building for Windows on Arm requires Electron 6.0.8 or newer. All of 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.

3.2 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.

4.1 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.

4.2 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 Related information

Here are some resources related to material in this guide:

- **Arm community** - Ask development questions, and find articles and blogs on specific topics from Arm experts
- **The StaffPad website**
- **The Electron framework documentation**
- **The Tweeten website**
- **The UWP framework documentation**
- **Other resources for Windows on Arm frameworks**

6 Next steps

In this guide, we saw the general steps required to port applications to Windows on Arm. We then saw two examples, each with its own challenges. You should now be able to use the general steps from this guide to understand the specific steps you need to port your own application.

To learn about other frameworks you can port to, see the Windows on Arm resources.