OHOS OpenHarmony OS for Next Gen Mobile

Jonathan Schwender
Senior OS Engineer

May 6th, 2024
What is OpenHarmony?

OpenHarmony

Advanced OS base for a connected, intelligent world

HarmonyOS

Ultimate experience with software-hardware-chip-cloud integration to support Huawei’s high-quality products.

Third-party commercial releases & products
Empower a range of industries.

Unified ecosystem for apps and services

HUawei

Open Source Community & Third-party

Image Source: Chen Haibo, STW 2023
OpenHarmony OS
350+ Software and Hardware Products Across Key Sectors

**Energy**
Mining and electric power terminals
- Automatic data reading and comparison
- Automatic data panel

**Aerospace**
Satellites
- Cloud
- Data is migrated to the cloud to generate gas reports and calibration work orders.

**Industry**
Drones and industrial terminals

**Finance**
Financial terminals

**Transportation**
Smart tunnels

**Healthcare**
Smart medicine cabinets

**Education**
Harmony classroom

**Government**
e-Government terminals
Harmony OS NEXT

• AOSP (Android) compatibility layer removed
• Apps need to use the new ArkUI framework based on ArkTS (TypeScript)
  > All apps need to be rewritten to use ArkUI
  > Huge effort to port the top 5000 apps to support (Open-) HarmonyOS
• **Custom Kernel** (with Linux / POSIX compatibility layer)
• Commercial Release: Q4 2024
• Target Audience
  > First: Chinese Mainland
Harmony OS NEXT apps

- Huawei phone users spend 99% of their time in 5000 apps.
- Huge Porting effort
  - 4000 out of the top 5000 apps already ported or being ported
  - Ongoing discussions with the developers of the remaining 1000
- In China Mini-apps are extremely popular
  - Mini web-based apps inside Wechat.
- Many Web developers are already very familiar with TypeScript
DevEco Studio IDE and SDK

- Dev Eco Studio is the official IDE for OpenHarmony
- Latest Release: 4.1
- Dev Eco Studio IDE and the SDK are available from the official release notes
  > English release notes are not available yet.
- Features include:
  > Debugging
  > Hot reloading
  > UI previewer
  > Emulator
  > Profiling / tracing
- Missing: Rust pluging
Dev Eco Studio

- Harmony OS NEXT SDKs are still in closed Developer Preview phase
Dev Eco Studio

- OpenHarmony 4.1 SDK is freely available
- Can be installed automatically in the IDE
- Additionally select Native, if you want to use C/C++/Rust code.
Dev Eco Studio

- Project Wizard to create an app, including all the boilerplate
- **Documentation of the package structure**
- The build-profile.json5 in the module level configuration contains a `targets` array, where the `runtimeOS` can be set to either HarmonyOS or OpenHarmony.
  > Affects signing of the bundle
App Signing

- OpenHarmony apps can run on all OpenHarmony devices
- For Security reasons, apps must be signed
- Required signature depends on the OpenHarmony distribution
- Hapsigner tool is used to sign an application bundle
- HarmonyOS: Signing keys can be automatically generated in Dev Eco Studio
- OpenHarmony: Requires manually generating the keys
OpenHarmony app (Stage Model)
Anatomy of an OpenHarmony App

Your App

AppScope

oh-package.json

build-profile.json

hvigorfile.ts

<module_name>

App Metadata

Package Manifest

Module declaration
Anatomy of an OpenHarmony app module
Anatomy of an OpenHarmony app module

- ets: Contains the Abilities and Pages of the module written in ArkTS
  - Commonly: 1x UI Ability with multiple Pages
- ArkTS is the primary language for OpenHarmony apps
- cpp: Optional - Native C/C++ code built with CMake
  - Types and Functions are declared via an `index.d.ts` file
  - ArkTS code can import those types / functions

```typescript
export const add: (a: number, b: number) => number;
```
ArkTS: Stricter TypeScript flavor

• Goals:
  > Easy to read
  > Performance and Efficiency
  > Prevent common errors

• Static types:
  > All types are known at compile-time
  > any/unknown is forbidden
  > Object layout cannot be changed at runtime

• projects that already follow the best TypeScript practices can keep 90% to 97% of their codebase intact.

• Further reading:
  > ArkTS introduction
  > ArkTS migration guide
ArkTS – ArkUI specific additions

- Additional built-in components
- ArkUI specific decorators
- Used within Pages.

```dart
@Entry
@Component
struct Hello {
  @State
  myText: string = 'World',

  build() {
    Column() {
      Text('Hello ${this.myText}')
        .fontSize(50)
        .divider()
        .button('Click me')
      .onClick() {
        this.myText = 'ArkUI'
      }
      .height(50)
      .width(100)
      .margin({ top: 20 })
    }
  }
}
```

Hello World
Example ArkTS App - Entry Page Ability

```java
export default class EntryAbility extends UIAbility {
    onCreate(want: Want, launchParam: AbilityConstant.LaunchParam) {}
    onDestroy() {}

    onWindowStageCreate(windowStage: window.WindowManagerStage) {
        windowStage.loadContent('pages/Index', (err, data) => {
            if (err.code) {
                hilog.error(0x0000, 'testTag', 'Failed to load the content.');
                return;
            }
        });
    }

    onWindowStageDestroy() {}
    onForeground() {}
    onBackground() {}
}
```
Example: ArkTS App – Main page

```rust
@Entry
@Component
struct Index {
    @State counter: number = 1;
    build() {
        Row() {
            Column() {
                Text('Test addition: ')
                    .fontSize(50)
                    .fontWeight( FontWeight.BOLD)
                Text('$(this.counter) + 1 = ${ add(this.counter, 1) } ')
                    .fontSize(50)
                    .textAlign(TextAlign.Center)
                    .fontWeight( FontWeight.BOLD)
                    .onClick() => {
                        this.counter += 1;
                    }
                }
                .width('100%')
            }
            .height('100%')
        }
    }
}
```

Source-code: https://github.com/jschwe/ohos-rust-demo
Example: ArkTS app with native C++ code

```javascript
import cpp_lib from 'libentry.so';

@Entry
@Component
struct Index {
  @State counter: number = 1;
  build() {
    Row() {
      Column() {
        Text('Test addition: ')
          .fontSize(50)
          .fontWeight(FontWeight.Bold)
        Text('${this.counter} + 1 = ${cpp_lib.add(this.counter, 1)} ')
          .fontSize(50)
          .textAlign(TextAlign.Center)
          .fontWeight(FontWeight.Bold)
          .onClick(() => {
            this.counter += 1;
          })
      }
      .width('100%')
    }
    .height('100%')
  }
}
```

Source-code: https://github.com/jschwe/ohos-rust-demo
Example app C/C++ Code

- Assumption: `add` takes a long time – We want to speed it up!
- Lots of Boilerplate:
  - module and function registration
  - Extracting the function arguments from the javascript containers
  - Corresponding ArkTS function definition

```javascript
export const add: (a: number, b: number) => number;
```
Example app: Rust code

- **napi-rs** is an existing “framework for building pre-compiled Node.js addons in Rust”
  - Community maintained fork with ohos support `napi-ohos` under development
- Boilerplate is significantly reduced
  - The ArkTS function declaration can be automatically generated by a build-script.

```rust
use napi derive ohos::napi;

#[napi]
pub fn add(left: u32, right: u32) -> u32 {
    left + right
}
```

```typescript
export const add: (a: number, b: number) => number;
```
How can we integrate the Rust library?

- A prebuilt dynamic library can be placed under `<module_name>/libs/<arch>/lib<name>.so`
- We could setup Dev Eco Studio to build the Rust project and copy the library and the `index.d.ts` files.
- We could write an hvigor plugin in TypeScript
- C/C++ code is built with CMake
- The [Corrosion](https://corrosion.dev/) CMake module can automatically import Cargo projects
  - Automatically sets the correct linker and Rust compiler target
  - The OpenHarmony SDK (4.1) ships CMake 3.16, which is missing a required feature
  - The feature could be backported to an older Corrosion version
  - Upstream CMake is missing one file `Platform/OHOS.cmake`
- Conclusion: For now the simplest solution is the first one.
Experiment: Compiling Ripgrep for OHOS

- Popular grep alternative written in Rust
- Add the std library for our target
- We need to specify the linker explicitly
What about bigger, native apps?

- Example: servo, a rendering engine written in Rust
- Main servo components ~240K lines of Rust code
- 700+ Rust and C/C++ dependencies
- Multiple build systems involved
  > cc-rs
  > cmake
  > autotools
- Simple UI (URL bar + Browser window)

### Servo Dependencies (Estimation)

<table>
<thead>
<tr>
<th>Language</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rust</td>
<td>3.9 million</td>
</tr>
<tr>
<td>C++</td>
<td>1.3 million</td>
</tr>
<tr>
<td>C</td>
<td>1.3 million</td>
</tr>
</tbody>
</table>

Counted with `scc` on results of `cargo vendor`, with winapi*, windows* and ndk crates removed.
Step 1: Compile libservo for OpenHarmony

• Goal: Estimate how much code needs to be adapted to OHOS APIs
• Create a dummy library that depends on libservo and fix all compilation and linking errors

1. Figure out environment variables needed for building C/C++ dependencies (next slide)
   > Set C/C++-Compilers, sysroot, pkg-config ...

2. Fix Rust dependencies failing to build for OpenHarmony
   > Often the issue was already fixed by other community members – Just need to update the dependency
   > But: Updating long dependency chains can be quite time-consuming!
   > Sometimes backporting an OHOS fix to an older version of a crate can be a quick band-aid solution.
   > Hardcode / stub everything else that still needs to be implemented (differently) for OpenHarmony

3. All dependencies compile ? -> Fix linking issues
   > Often simply select feature to build the library from source
   > Sometimes wrong dependencies get linked in.
     - Example: ‘the target OS is Linux -> Must have X11 or wayland)
Magic environment variables

- **OHOS_SDK_NATIVE**: Set by Dev Eco Studio to the native Directory of the SDK
- **OHOS_LLVM_BIN=${OHOS_SDK_NATIVE}/llvm/bin**
- **CARGO_TARGET_AARCH64_UNKNOWN_LINUX_OHOS_LINKER="${OHOS_LLVM_BIN}/aarch64-unknown-linux-ohos-clang"**
- **PATH=${PATH}:${OHOS_LLVM_BIN}**

**Bindgen**

| LIBCLANG_PATH=${OHOS_SDK_NATIVE}/llvm/lib |
| CLANG_PATH=${OHOS_LLVM_BIN}/aarch64-unknown-linux-ohos-clang |

Required to avoid bindgen #2682
## Magic environment variables Part 2

### pkg_config

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PKG_CONFIG_SYSROOT_DIR_aarch64_unknown_linux_ohos</code></td>
<td><code>${OHOS_SDK_NATIVE}/sysroot</code></td>
</tr>
<tr>
<td><code>PKG_CONFIG_PATH_aarch64_unknown_linux_ohos</code></td>
<td><code>=${OHOS_SDK_NATIVE}/sysroot/usr/lib/pkgconfig:=${OHOS_SDK_NATIVE}/sysroot/usr/share/pkgconfig</code></td>
</tr>
</tbody>
</table>

### cc-rs and cmake-rs

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CC_aarch64 Unknown Linux Ohos</code></td>
<td><code>${OHOS_LLVM_BIN}/aarch64-unknown-linux-ohos-clang</code></td>
</tr>
<tr>
<td><code>CXX_aarch64 Unknown Linux Ohos</code></td>
<td><code>${OHOS_LLVM_BIN}/aarch64-unknown-linux-ohos-clang++</code></td>
</tr>
<tr>
<td><code>AR=${OHOS_LLVM_BIN}/llvm-ar</code></td>
<td></td>
</tr>
<tr>
<td><code>CXXSTDLIB_aarch64_unknown_linux_ohos</code></td>
<td><code>c++</code></td>
</tr>
<tr>
<td><code>CMAKE_TOOLCHAIN_FILE_aarch64_unknown_linux_ohos</code></td>
<td><code>${OHOS_SDK_NATIVE}/build/cmake/ohos.toolchain.cmake</code></td>
</tr>
<tr>
<td><code>CMAKE_C_COMPILER_aarch64_unknown_linux_ohos</code></td>
<td><code>=${CC_aarch64_unknown_linux_ohos}</code></td>
</tr>
<tr>
<td><code>CMAKE_CXX_COMPILER_aarch64_unknown_linux_ohos</code></td>
<td><code>=${CXX_aarch64_unknown_linux_ohos}</code></td>
</tr>
<tr>
<td><code>PKG_CONFIG_SYSROOT_DIR_aarch64_unknown_linux_ohos</code></td>
<td><code>=${OHOS_SDK_NATIVE}/sysroot</code></td>
</tr>
<tr>
<td><code>PKG_CONFIG_PATH_aarch64_unknown_linux_ohos</code></td>
<td><code>=${OHOS_SDK_NATIVE}/sysroot/usr/lib/pkgconfig:=${OHOS_SDK_NATIVE}/sysroot/usr/share/pkgconfig</code></td>
</tr>
</tbody>
</table>
Step 2: Create a minimal ArkTS app for libservo

• In Step 1, we "fixed" some compilation issues by using `unimplemented!()` or `todo!()`.
• Now we implement the missing parts as we hit them.
• But we can still take shortcuts, like hardcoding some values to quickly get a demo.
ArkUI XComponent

- XComponent provides a window native code can render to
- Start with the simplest possible UI, which is just one Xcomponent
- Initialize servo from there
- Only minor changes required, in platform specific code during the graphics initialization phase
- After that servo loaded and rendered just fine
Demo: Servo browser app on OpenHarmony

- The UI currently only consists of the URL bar and the browser window
- The bare browsing experience works
- WebGL support is currently disabled
- Scrolling works, but fling support is not implemented yet
- Callbacks from Rust to ArkTS are not implemented yet (e.g. Updating the URL bar, if the user clicks a link)
- In the process of upstreaming changes
- [Demo ArkUI sources, Servo branch](https://servo.org)
Demo: Servo – Changes required

• ArkUI <-> Libservo Layer  
  > Easy – Thanks to the trait system  
• Adapt OS specific window initialization  
  > More challenging, Documentation could be improved  
  > Offscreen Buffer still on my todo list  
• Adapt the font-loading  
• Figure out all the Environment variables that need to be set for the build systems  
  > Also depends on the Host OS ...  
• Create Rust bindings for OpenHarmony APIs  
  > Hilog, Hitrace
Summary: Rust on OpenHarmony

• Since Rust 1.78: **ohos** is supported as a **Tier 2 Rust target**
  > Follow the instructions to install the OpenHarmony SDK
  > Install prebuilt std library via rustup:
  > rustup target add aarch64-unknown-linux-ohos
• The linker should be explicitly set (e.g., via `CARGO_TARGET_$TARGET_LINKER`)
• Cross-Compiling pure Rust code generally works fine
  > Some libc functions are purposely not available (1, 2)
• Cross-Compiling code with C/C++ dependencies is a bit more painful
  > Depending on the involved build-systems a bunch of environment variables need to be set
  > Some build-systems (autoconf) just fail if they don’t recognize `ohos` and need to be patched.
Future work

• Goal: Make Rust a first-class citizen for native OpenHarmony code
• Provide safe bindings for (more) native OpenHarmony APIs
• Setup a reusable Github CI action
• Explore if the changes required to use the napi-rs crates can be merged back upstream
• OpenHarmony provides the Function Flow Runtime Kit (FFRT) for coroutine based scheduling
  > Ideally we would only have one coroutine runtime
Thank you