OCI Bundle Generator Requirements

Overarching Goals

- **Convert OCI images generated by the Firebolt SDK into a lightweight, minimal OCI bundle specific to a device that can be run by Dobby on the STB**
    - In traditional container solutions (Docker, Podman), this stage is performed by the container runtime itself. However, for performance reasons we cannot pull an entire image and unpack it/convert it on a STB. Therefore bundle generator moves this stage to a pre-processing step in a cloud environment
- Generated bundles must be able to run on a local machine/STB for use during application development
- Allow creating configuration files per DAC compatible device. This will allow BundleGenerator to generate bundles that will work on a device without actually running on the device.
- Allow creating operator configurations to apply business logic to a container image file across all platforms.
- Process an extended "application capabilities" section to add/remove/change specific sections of the generated bundle according to application requirements
- Prevent creating a bundle for an incompatible device. If the device does not support the required RDK release, then the image cannot be run on that device. However, aim to ensure broad compatibility of applications. The goal of running containerised apps is to allow applications to not be concerned about the platform they are running on
- Allow processing stages to be extended as necessary

Input

OCI Image

Description

OCI Image - generated locally or pulled from an OCI Registry

Format

- Directory storing the manifest, layer tarballs and signatures as individual files. Useful for quick debugging of a newly generated image locally (for developer using the SDK)
- OCI Image Layout Path - An image tag in a directory compliant with "Open Container Image Layout Specification" at path. See https://github.com/opencontainers/image-spec/blob/master/image-layout.md for more details
    - Allow optionally passing in tag to use a specific tagged image version. Default to latest

Platform Configuration

Description

Configuration file containing information specific to a single RDK hardware platform. The SDK would ship with configurations for common reference devices such as the Raspberry Pi, Emulator and x86.

The information in this configuration could include:

- Device name
- Architecture
- RDK Version (2020Q1, 2020Q2 etc)
    - Include available/installed RDK Services (Thunder NanoServices)
    - Could have multiple configurations for the same device but for different RDK releases if different versions are in use in the field
- GPU Device nodes (and similar - e.g. VPU) that need to be mapped inside the container if graphics is needed
- GPU Libraries
    - Set of tuples containing libraries that must be automatically mounted inside the container for graphics support
    - This allows for adding custom, platform specific graphics libraries into containers (for example `libnexus` on broadcom platforms)
    - These mounts will **override** any graphics libraries included with the application
- General Libraries
    - A list of all the libraries available on the device. This will be used when converting the image to bundle to reduce the size of the final output by bind mounting in libraries that are the same major version in the image and on the device
    - This section must be **automatically generated** for a platform
    - It should be possible for a developer to force the use of a library minor version (e.g. `libFoo.so.2.1.5`). In this case, unless that exact minor version is available on the STB, the version in the image will be used
        - The mechanism for this should be exposed in the STB publishing tools
- Maximum amount of RAM an application can use (maybe just the amount of RAM available on the platform?)
- Supported resolutions (1080p, 4k)
- Supported Networking options
- Custom environment variables that must be set, specific for the platform
    - XDG_DATA_DIRS
    - XDG_RUNTIME_DIR
- Custom files/paths that must be mounted inside the container
- Users/Groups
    - Containerised apps should run as a unique user and group. Define what users/groups a container should run in
    - Allow specifying a range: e.g. `AppUser[0-99]:AppGroup`
- Non-standard mappings
    - Across RDK, many paths and file names are standardised across builds. If the device uses any non-standard paths or devices, this set of tuples can override the defaults.

This configuration would likely be split into multiple files for readability/maintenance purposes.

Most of this configuration will be generated by hand per device, although this can be based off templates provided for reference devices. However, some large sections should be automatically generated, such as the library list

Format

- JSON file

Operator Configuration

Configuration file containing information that applies to **all** operator devices. This will allow enforcing consistency between devices and applications if necessary. This could include:

- Environment variables

TODO:: What else might we need here?

Output

OCI Bundle

Description

The main output of BundleGen should be an extended OCI bundle. This will contain the `config.json`, with an additional section containing the `rdkPlugins` section to provide additional functionality required by the application - normally specified in the application capabilities

This bundle would be the smallest possible compatible bundle, containing only libraries that are not present on the STB. Any libraries that can be provided by RDK on the STB should be bind-mounted inside the container. This reduces the size of the bundle, necessary to decrease download time and increase the amount of containers that can be stored on the limited flash on an STB.

This differs from the current approach in Dobby where the `/lib` and `/usr/lib` directory are mounted into the container in their entirety and the application is expected to use only the libraries on the STB - unable to provide its own.

This output format would be used to allow downloading the OCI Bundle* to a STB for execution directly by Dobby. Distribution could take place over any operator specific mechanism - FTP, HTTPS etc.

Format

- Directory
- Tarball of above directory

Minimal OCI Image (??)

Description

This is a potential alternative output format of a minimal, single-layer OCI image specific to this container. This small image could then be uploaded to an OCI registry, allowing an application to be downloaded and distributed using the OCI distribution specification. The image could then be unpacked and run by Dobby (TODO:: how/when would this occur? On box?)

Alternative: same image, but just add a tag for this single-layer version for a certain device?

TODO:: This needs to be fleshed out if we're

Processing Workflow

0. Obtain the OCI Image

Whilst out of the main scope of BundleGen, we must first obtain an OCI image for processing.

One potential candidate tool for retrieving images is **skopeo**: https://github.com/containers/skopeo

> `skopeo` is a command line utility that performs various operations on container images and image repositories.
>
> `skopeo` does not require the user to be running as root to do most of its operations.
>
> `skopeo` does not require a daemon to be running to perform its operations.
>
> `skopeo` can work with OCI images as well as the original Docker v2 images.
>
> Skopeo works with API V2 container image registries such as docker.io and quay.io registries, private registries, local directories and local OCI-layout directories. Skopeo can perform operations which consist of:
>
> - Copying an image from and to various storage mechanisms. For example you can copy images from one registry to another, without requiring privilege.
> - Inspecting a remote image showing its properties including its layers, without requiring you to pull the image to the host.
> - Deleting an image from an image repository.
> - When required by the repository, skopeo can pass the appropriate credentials and certificates for authentication.

1. Unpack the OCI image

The first stage of processing should be to unpack to the OCI image into an OCI bundle (rootfs + `config.json`) that we can manipulate to form our final bundle to be delivered to the STB.

Solution A

Proposed tool: **umoci**: https://umo.ci/quick-start/workflow/

Umoci is an official opencontainers (formally OpenSUSE) tool that can manipulate downloaded OCI images and convert them to OCI bundles. Written in Go.

```
% skopeo copy docker://opensuse/amd64:42.2 oci:opensuse:42.2 # Obtain the image
from the docker hub using Skopeo
% sudo umoci unpack --image opensuse:42.2 bundle
% ls -l bundle
total 720
-rw-r--r-- 1 root root   3247 Jul  3 17:58 config.json
drwxr-xr-x 1 root root    128 Jan  1  1970 rootfs
-rw-r--r-- 1 root root 725320 Jul  3 17:58
sha256_8eac95fae2d9d0144607ffde0248b2eb46556318dcce7a9e4cc92edcd2100b67.mtree
-rw-r--r-- 1 root root    270 Jul  3 17:58 umoci.json
```

Here the opensuse image with tag 42.2 has been unpacked to produce a rootfs directory and config.json. As per the documentation:

> ## SYNOPSIS
>
> **umoci unpack --image**=*image*[:*tag*] [**--rootless**] [**--uid-map**=*value*] [**--uid-map**=*value*] [**--keep-dirlinks**] *bundle*
>
> ## DESCRIPTION
>
> Extracts all of the layers (deterministically) to an OCI runtime bundle at the path *bundle*, as well as generating an OCI runtime configuration that corresponds to the image's configuration. In addition, an **mtree**(8) specification is generated at the time of unpacking to allow filesystem deltas to be generated by **umoci-repack**(1) and thus allowing for the creation of layered OCI images.

Note that umoci can also be used to repack the modified rootfs back into the original image as a delta layer, or into a completely new image, potentially useful for a minimal OCI image output format. Changes to the config.json are not reflected in a repacked image and must be modified with the `umoci config` command

Potential Pitfalls

umoci doesn't currently support multi-arch images, and doesn't expose a way through the CLI to select which arch to unpack. See:

https://github.com/opencontainers/umoci/issues/10

> Right now, `umoci` supports recursively looking through indexes to find a manifest -- all the code is in place for that.
>
> The main issue is that (in the CLI front-end) there's no way to say (in the case of ambiguity, like a multi-arch image) what manifest you'd prefer. I was thinking of just making it use the host, but given the fun CLI spaghetti that `skopeo` has turned into (in this regard) it'd be nicer if the CLI UX was more reasonable. Docker just uses the current host, but given the need for cross-compiling this is a bit of a pain.
>
> In addition, `umoci` doesn't know how to create a new multi-arch image. It can modify one that already exists, but it doesn't know how to "upgrade" a single-arch image to a multi-arch one. This would be quite trivial to add, but without the first part (a CLI for extraction that makes sense) the CLI for the creation similarly wouldn't make sense.

https://github.com/opencontainers/umoci/issues/313

> `umoci` has very ... limited ... support for architecture specification because the entire design of architecture handling in OCI was being redesigned when I first started working on `umoci`.

Potentially in the roadmap for 0.5 release

Solution B

Proposed tool: **OCI-Image-Tool** https://github.com/opencontainers/image-tools

Another official opencontainers tool for working with images. Again written in Go.

This does currently have support for dealing with multi-arch images, unlike umoci

```
$ skopeo copy docker://busybox oci:busybox-oci:latest
$ mkdir busybox-bundle
$ oci-image-tool unpack --ref name=latest busybox-oci busybox-bundle
$ tree busybox-bundle
busybox-bundle
├── bin
│   ├── [
│   ├── [[
│   ├── acpid
│   ├── addgroup
│   ├── add-shell
│   ├── adduser
│   ├── adjtimex
│   ├── ar
│   ├── arp
│   ├── arping
│   ├── ash
[...]
```

> ## NAME
>
> oci-image-tool unpack - Unpack an image or image source layout
>
> ## SYNOPSIS
>
> **oci-image-tool unpack** [src] [dest] [OPTIONS]
>
> ## DESCRIPTION
>
> `oci-image-tool unpack` validates an application/vnd.oci.image.manifest.v1+json and unpacks its layered filesystem to `dest`.
>
> ## OPTIONS
>
> **--help** Print usage statement
>
> **--ref**=[] Specify the search criteria for the validated reference, format is A=B. Reference should point to a manifest or index. e.g. --ref name=v1.0 --ref platform.os=latest Only support `name`, `platform.os` and `digest` three cases.
>
> **--type**="" Type of the file to unpack. If unset, oci-image-tool will try to auto-detect the type. One of "imageLayout,image,imageZip"
>
> **--platform**="" Specify the os and architecture of the manifest, format is OS:Architecture. e.g. --platform linux:amd64 Only applicable if reftype is index.

However, this tool is now unmaintained and umoci is recommended as the replacement...:

> This project is no longer actively maintained. However, umoci is a much more full-featured tool for manipulating OCI images, and is now an OCI project as a reference implementation of the OCI image-spec. I would strongly suggest people move to using umoci.

https://github.com/opencontainers/image-tools/issues/222

2. Validate App is compatible

This should be a first-pass, fail-fast test to ensure that the app can actually be packaged and run on the target device. Check that the specified app RDK version is suitable with the RDK versioned defined in the platform configuration. Check that the platform can supply the capabilities required by the app - such as RAM, storage space and resolutions.

If the app is incompatible, return an error.

3. Library Matching

Run an `ldd` on the binary (using a suitable version of ldd for the arch), to work out it's dependencies. Then, based on the information in the platform configuration, determine if a suitable library is available on the STB.

If the STB provides a compatible library - deemed to be one of the same MAJOR version (assuming semver), then edit the `config.json` in the bundle to add the new bind mount into the container. Delete the library from the bundle rootfs.

If the STB does not provide a suitable library, then the library should remain in the rootfs.

This step should then remove any libraries that are not needed by the application.

Here be dragons: If the app uses `dlopen` to manually load dynamic libraries, we somehow need to make sure those aren't accidentally removed at this stage

Failure here should go back to the safe option of just using the bundled libraries for the app - since that means the app will work, but final size will be larger.

3a. Add GPU Lib Mounts

Based on the libraries specified in the "GPU Libraries" section of the platform config, add bind mounts for the necessary graphics libraries into the container.

Failure here should be fatal.

4. Add Extra Bind Mounts

If any other bind mounts are needed, for example for OCDM or similar, add them now.

5. Add Devices

Any devices set in the platform configuration should now be added to the OCI config in the bind mounts section and the `devices` section. Note as per the specification, crun will automatically provide the container with:

- `/dev/null`
- `/dev/zero`

- `/dev/full`
- `/dev/random`
- `/dev/urandom`
- `/dev/tty`
- `/dev/console`
- `/dev/ptmx`. A bind-mount or symlink of the container's `/dev/pts/ptmx`.

When starting from a Dobby spec, Dobby also only allows devices in the device whitelist:

```
// ----------------------------------------------------------------------------
/**
 *  @brief Builds the whitelist of allowed device numbers
 *
 *  At the moment only the following devices are added:
 *      hidraw, 0 - 100
 *      input, 64 - 95
 *
 *  The list is exclusive of the opengl / graphics device nodes and the set of
 *  'standard' device nodes.
 *
 */
```

7. Add Environment Vars

Any environment variables in the platform/operator config should be added to the runtime config now

7. Add Plugins

Based on the capabilities required by the application and provided by the platform, generate and write the `rdkPlugins` section of the runtime config if necessary

8. Write Anything Else

Any other sections of the runtime config that need to be modified should be done now.