Device-Specific Files and Processes

Introduction

With the growth of devices on which EVE runs, we have seen more "flavours" of EVE. Not just architecture - amd64 and arm64 - but also board-specific, e. g. imx.

These variants are referred to as "platforms". Different variants of EVE are built by calling:

```
make eve PLATFORM=<platform>
```

With some of these variants, we need not only specific kernels, but also specific files to be available on the system. These cannot be downloaded runtime because:

- The data source with the packages might not be available to end devices at boot time;
- Even if the data source is available, it might be bandwidth constrained;
- The device might need the files very soon after startup.

Thus, the only option is to make these available built-into the platform-specific-variant of EVE.

This document describes a standard method for making these device-specific files available on an EVE build, as well as executing necessary startup scripts, at both a device level and an EVE-specific services (pillar) level.

File Locations

All device-specific files should be placed on the root filesystem in:

```
/opt/vendor/<vendorname>/
```

Note that, by definition, no 2 platform-specific versions of EVE are the same and built for the same platform. Thus, it is highly likely that in all cases there will be either zero (no device-specific flavour) or one (device-specific flavour) subdirectory under `/opt/vendor/` . For example, an Nvidia Jetson EVE will have `/opt/vendor/nvidia` , while an imx EVE will have `/opt/vendor/imx` . In theory, we could simply place everything under `/opt/vendor` and avoid another layer of device-name-specific subdirectory. We choose not to do this for two reasons.

1. We cannot guarantee that in the future we never will have a device with files from two distinct vendors; the `/opt/vendor/<vendor_name>` structure provides future flexibility.
2. Analysis and debugging are much easier if it explicitly states what the device the vendor-specific files are for.

The directory `/opt/vendor/` can be mounted to any system container that needs it as a mount in `rootfs.yml` . It always will be mounted into pillar as `/opt/vendor:/opt/vendor`.

Populating Vendor Directory

As `/opt/vendor` is in the base OS, which is read-only, it must be populated build-time. We use the `init` section of `rootfs.yml` to populate it.

Any device that requires a Board Support Package (BSP) should create a container via a directory in `lf-edge/eve/pkg/bsp-<vendor>` , e.g. `bsp-imx` (which already exists) or `bsp-nvidia` . The contents of the final stage in the Dockerfile **must** be `FROM scratch` and **must** save files solely to `/opt/vendor/<vendorname>` so that the files will be properly placed in the rootfs.

The actual update of `rootfs.yml` on a per-vendor basis, so that the files can be placed in the base operating system, is covered in the later section "Updating rootfs.yml".

Long-Running Services

If the device requires device-specific long-running services, for example a fan or other device controller, these are considered system-level services, and should be added to the `services` section of `rootfs.yml` .

The source to such services should create a container via a directory in `lf-edge/eve/pkg/services-<vendor>` , e.g. `services-nvidia` . As this runs in `services` , and therefore in its own container, it can be structured however it wants internally.

It should avoid duplicating any files already in `/opt/vendor/<vendor>` , instead mounting those in, if at all possible.

The actual update of `rootfs.yml` on a per-vendor basis, so that the the `services` section can be modified, is covered in the later section "Updating rootfs.yml".

Startup Services

If the device requires startup services, it depends upon the nature of the startup service:

- system-wide
- pillar-specific

System-Wide

System-wide startup services, e.g. initializing a device, should be performed in an `onboot` container in `rootfs.yml` .

The source to such startup services should create a container via a directory in `lf-edge/eve/pkg/onboot-<vendor>` , e.g. `onboot-nvidia` . As this runs in `onboot` , and therefore in its own container, it can be structured however it wants internally.

It should avoid duplicating any files already in `/opt/vendor/<vendor>` , instead mounting those in, if at all possible.

The actual update of `rootfs.yml` on a per-vendor basis, so that the files can be placed in the `onboot` section, is covered in the later section "Updating rootfs.yml".

Pillar-Specific

Pillar-specific startup services, e.g. modifications to user containerd `config.toml` or communications, should be performed by pillar itself.

On startup, pillar will execute any files found in `/opt/vendor/*/init.d/` . Obviously, if no `/opt/vendor/<vendorname>` directories exist, or those that do have no `init.d/` subdirectory, or those have no executable files, then nothing will get executed.

The `init.d/` startup programs should be created as part of `bsp-<vendor>` . Since those files are in `/opt/vendor/<vendorname>` , and are mounted into pillar, they will be available to pillar on startup.

Updating rootfs.yml

The above requires both permanent and platform-dependent dynamic changes to `rootfs.yml` .

Permanent

The only permanent change is to *always* have `/opt/vendor:/opt/vendor` mounted into pillar. This actually does not require changing `rootfs.yml` , but instead extending permissions in the pillar image. We modify pkg/pillar/build.yml:

```
org: lfedge
image: eve-pillar
config:
  binds:
    - /lib/modules:/lib/modules
    - /dev:/dev
    - /etc/resolv.conf:/etc/resolv.conf
    - /run:/run
    - /config:/config
    - /:/hostfs
    - /persist:/persist:rshared,rbind
    - /usr/bin/containerd:/usr/bin/containerd
    - /opt/vendor:/opt/vendor     # <---- NEW
  net: host
  capabilities:
    - all
  pid: host
  rootfsPropagation: shared
  devices:
    - path: all
      type: a
```

Dynamic

Dynamic changes to `rootfs.yml` are ones that sometimes are added and sometimes are not. Before we describe the mechanism, we need to describe how yml generation currently works, and changes to extend it.

Extending and Standardizing yml generation

`rootfs.yml` is composed from a template `rootfs.yml.in` which is modified by `.yq` files, and then filled in by `parse-pkgs.sh` with the names of the dynamic images.

Currently, there is no standard way of running any of the `.yq` files, although several exist for some variants. The only ones that get executed are for different hypervisors. The rootfs build process in BUILD.md under #generating-yml describes that the final `rootfs.yml` is built as follows:

1. The Makefile includes kernel-version.mk. This sets the variable `KERNEL_TAG` inside the make process to a specific docker image tag, based on the `ZARCH` and, if set, `PLATFORM`
2. The Makefile sees a dependency on `images/rootfs-$(HV).yml`
3. The Makefile runs `tools/compose-image-yml.sh images/rootfs.yml.in images/rootfs-$(HV).yml.in "$(ROOTFS_VERSION)-$(HV)-$(ZARCH)" $(HV)`, i.e. the utility compose-image-yml.sh, passing it:
    - the base template `images/rootfs.yml.in`, i.e. input file
    - the template for the specific HV file `images/rootfs-$(HV).yml.in`, i.e. output file
    - the version string, which is the `ROOTFS_VERSION`, hypervisor, and architecture
    - the hypervisor
4. `compose-image-yml.sh` does the following:
    a. Look for a modifier file `images/rootfs-$(HV).yml.in.yq`; this is identical to the HV-specific template (2nd argument), but with `.yq` appended to the filename.
    b. If it finds a modifier file, apply it to the base template, and save the result to HV-specific template.
    c. Search through the output file for the string `EVE_HV` and, if found, replace it with the hypervisor.
    d. If the version argument, which was generated from the git commit, contains the phrase `dirty`, i.e. uncommitted, then change the `PILLAR_TAG` in the output file to `PILLAR_DEV_TAG`, which will be used in a later stage.
5. The Makefile runs `./tools/parse-pkgs.sh images/rootfs-$(HV).yml.in > images/rootfs-$(HV).yml`, i.e. the utility parse-pkgs.sh, passing it as an input the HV-specific template generated in the previous step `rootfs-$(HV).yml.in`, and saving the output to the final `rootfs-$(HV).yml` file. In addition, the variable `KERNEL_TAG` is passed as an environment variable.
6. `parse-pkgs.sh` does the following:
    a. Gets the package tag for each directory in pkg/ via `linuxkit pkg show-tag ${dir}`, and save it to variable which looks like `<PKGNAME>_TAG`, e.g. `PILLAR_TAG` or `WWAN_TAG`.
    b. Go through the input file - the HV-specific template - and replace the tags with the appropriate values. This includes the value of `KERNEL_TAG` as passed by the Makefile on calling `parse-pkgs.sh`.

Notably, `compose-image-yml.sh` look for a file named `images/rootfs-$(HV).yml.in.yq` and applies it to the base template `images/rootfs.yml.in` to generate `rootfs-$(HV).yml.in`. This, in turn, is used as input to `parse-pkgs.sh`.

This process has the following issue:

- It is limited to hypervisors; theoretically, calling it with something else as if it were a hypervisor, e.g. `make rootfs HV=something` would work, but then would get stuck at other stages, where `HV` really should be hypervisor.
- It is limited to just one modifier. For example, if we want multiple different variants on multiple hypervisors, there is no way to do that.

We propose modifying `compose-image.yml.sh` as follows:

1. Use flagged arguments, i.e. the current mode would be: `tools/compose-image-yml.sh -b images/rootfs.yml.in -m images/rootfs-$(HV).yml.in -v "$(ROOTFS_VERSION)-$(HV)-$(ZARCH)" -h $(HV)`
2. Replace the usage of a single modifer with multiple, e.g. `tools/compose-image-yml.sh -b images/rootfs.yml.in -v "$(ROOTFS_VERSION)-$(HV)-$(ZARCH)" -h $(HV) images/rootfs-$(HV).yml.in.yq images/modifier1.yq images/modifier2.yq`
3. Update the Makefile to call `compose-image-yml.sh` with multiple modifiers. `compose-image-yml.sh` will largely be "dumb", modifying with as many yq modifiers as passed to it, if they can be found. The Makefile will call pass modifiers for `HV` and `PLATFORM`

Modifier for Specific Device Files

With the above generic extension mechanism in place, we can update `rootfs.yml` for specific device files by:

1. Add `rootfs-$(PLATFORM).yml.in.yq`.
2. When building, call `make eve PLATFORM=<device>`

Sample modifiers

To ease in usage, the following is a simple `.yq` file for a platform named `foo`. It adds files to `init`, an `onboot` and long-running `services`.

```
.services += {"name": "services-foo","image": "SERVICES_FOO_TAG", "cgroupsPath": "/eve/services/services-foo"} |
.onboot += {"name": "onboot-foo","image": "ONBOOT_FOO_TAG"} |
.init += "BSP_FOO_TAG"
```

Note the usage of | to connect independent lines.