Extend CBFS to Storage Media

Visibility: Public *See go/data-security-policy for definitions if you want to change this.*
Authors: kramasub@
Last major revision: 2024-06-10 [*Click the date chip to change*]

Objective

Extend the Coreboot Filesystem (CBFS) to Storage Media such that non boot-critical RW Firmware components can be moved to the extended region. This will help to reduce the space pressure on SPI Flash and hence the BOM Cost.

Requirements

- Move RW firmware components to extended CBFS in a configurable way.
- No impact to boot-to-kernel time during normal/verified boot mode.
- Extend CBFS Verification to extended CBFS (eCBFS).
- Updateable through existing firmware update mechanism
- Enable the feature per mainboard/program basis.

Background

Application Processor (AP) SPI Flash has been growing on a regular basis to support an increase in the number and size of firmware components. ChromeOS platforms with Intel Big-Core SoCs are running out of 32 MiB SPI flash, whereas ChromeOS platforms with AMD and Intel Small-Core SoCs are running out of 16 MiB SPI flash.

The storage device is initialized and accessible during the later stages of boot flow - ramstage & payload. Some firmware components in the AP SPI flash, eg. Developer mode FW UI assets, EC RW binary, alternate FW/payload etc. are not needed until the boot flow jumps to the later stages of the bootflow. Also on Intel platforms, there are plans to move Converged Security Management Engine (CSME) firmware sync to later stages of the bootflow. The ability to move all these firmware components from SPI flash to the storage media can help with reducing the pressure on SPI flash and hence the BOM Cost.

Design ideas

Create one GUID partition table (GPT) entry for each FMAP section to be extended in the storage media. Some of these partitions accommodate the extended CBFS (eCBFS) that can hold arbitrary-length files. Based on the current requirement, 3 partitions will be added to the

storage media. An example AP firmware layout with extended FMAP sections and eCBFS is shown in Fig 1.

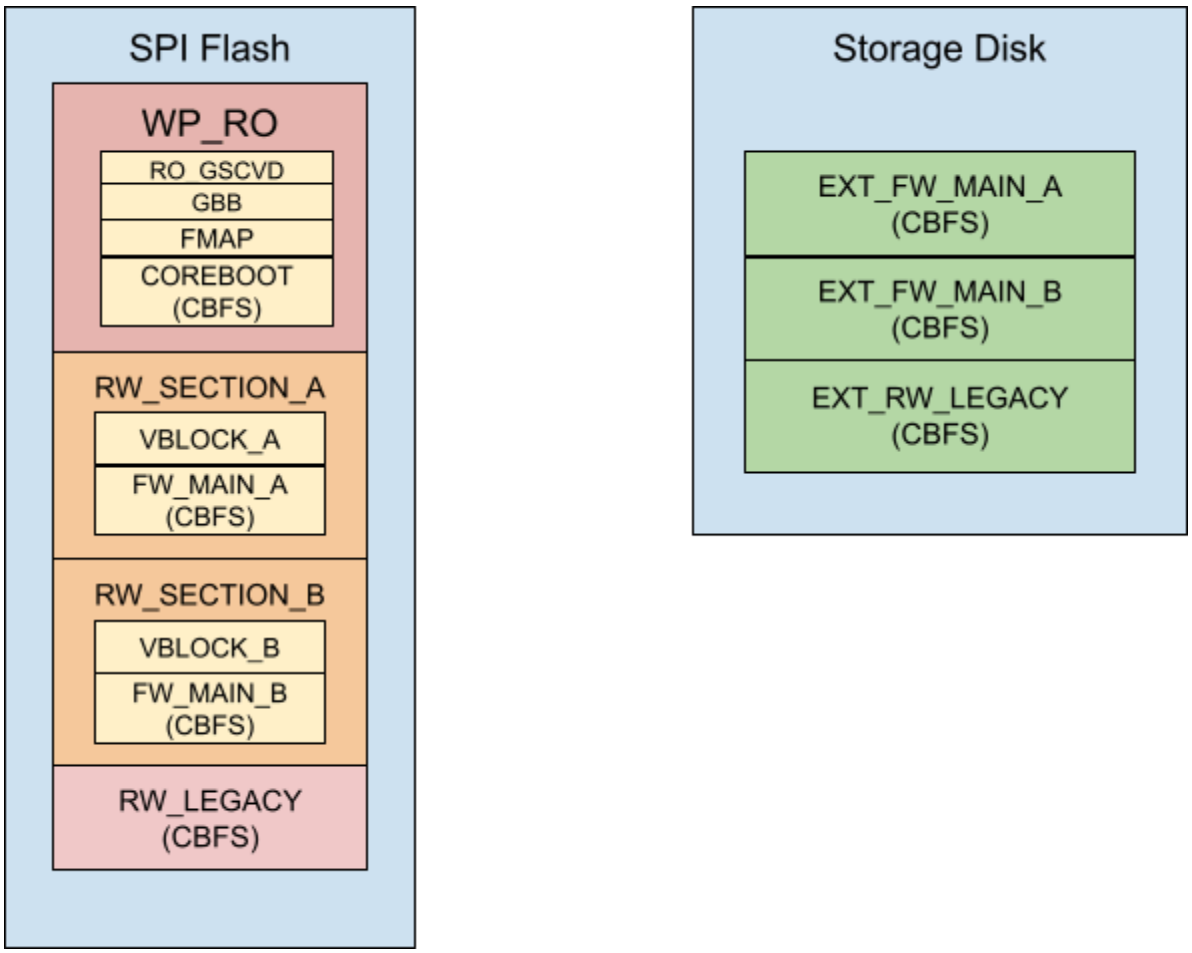

Fig 1 - AP Firmware Layout

During the bootup, RO firmware through verified boot chooses the appropriate RW Firmware slot, either RW_A or RW_B, to boot. The chosen RW firmware in turn verifies the associated eCBFS for that slot. For the example AP firmware layout shown above, RW_A slot verifies EXT_RW_A partition and RW_B slot verifies EXT_RW_B partition. This verification is also done on demand i.e. only when the eCBFS for a FW slot is available and only when an asset/file from the eCBFS is required. On successful verification, the RW slot continues booting to the kernel. On failure, boot status for the current RW slot is marked as failed and the alternate RW slot is chosen and the device reboots. If the alternate RW + EXT_RW slot too fails to boot, verified boot will fall to recovery mode. Fig 2 shows the verified boot flow with extended CBFS

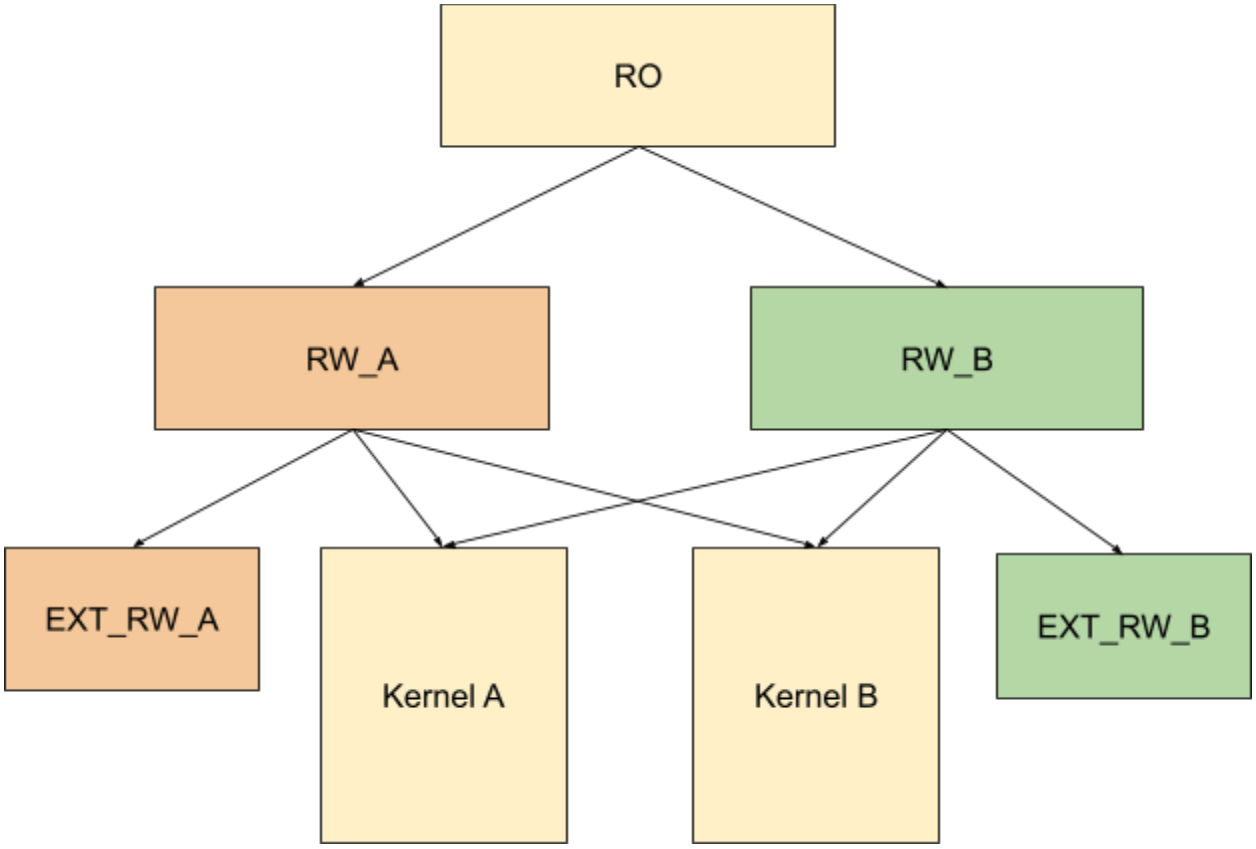

Fig 2 - Verified Boot Flow with extended FMAP and CBFS

Verification of extended CBFS

Extended CBFS verification works very similar to the existing CBFS verification. The metadata for each CBFS file along with the extended CBFS header are collected together to compute a hash. The choice of the hash algorithm will be made after some consultation with the security team/experts. The computed hash along with the extended CBFS descriptor is added as a file in the parent CBFS. This way the extended CBFS descriptor is verified as part of parent CBFS verification. This verified eCBFS descriptor is in turn used by the relevant boot stages (eg. payload) to verify the extended CBFS. Fig 3 shows the entire verification sequence discussed in this option.

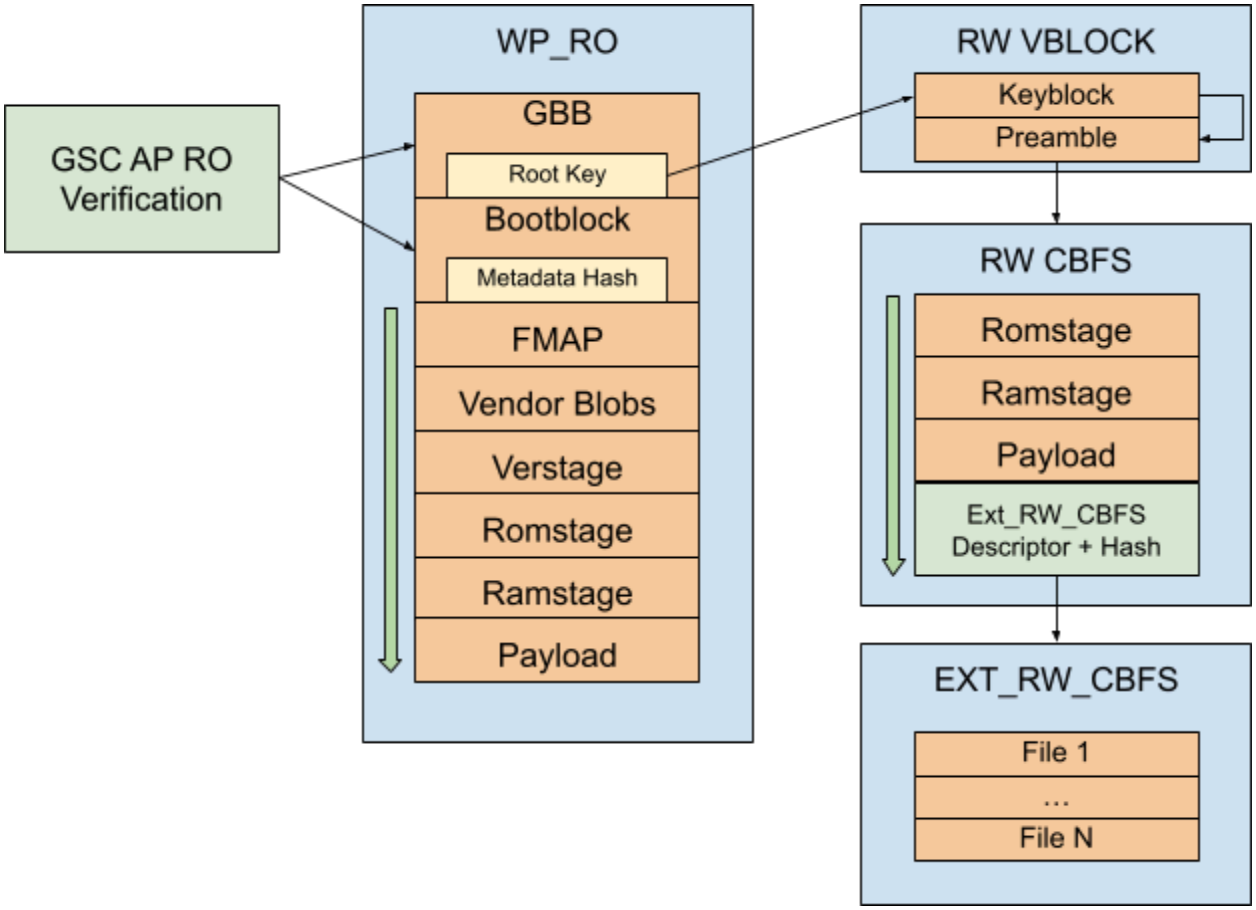

Fig 3 - Verification of eCBFS through hash inside parent CBFS

coreboot/Payload APIs and their internals to access eCBFS

The extended CBFS is accessed through the same set of coreboot & payload APIs as CBFS. Any time a file in eCBFS is accessed, it is first looked up in the parent CBFS metadata cache (MCACHE). If it is not found in the MCACHE and MCACHE is full, then it is searched in the parent CBFS itself. If still the file metadata is not found in the parent CBFS, then the extended CBFS device is accessed and the file is searched in that eCBFS device. A new internal accessor to get the eCBFS device is added - eg.

```
C/C++
const struct cbfs_boot_device *cbfs_get_ext_boot_device(void);
```

Again an optional eCBFS metadata cache can be built and maintained inside the eCBFS device. This is not strictly required, since the eCBFS is accessed only in certain boot scenarios. Please refer to the Appendix Reference Code Block Section for more details.

Building extended BIOS image

In coreboot, a new flashmap rule to extend the CBFS regions is defined. All the extended regions are grouped together under an extended flashmap purely for organizational purposes to aid during the firmware update:

- All the regions to be extended are gathered as part of a single extended BIOS image.
- Depending on the available regions in the extended BIOS image, associated GPT partitions in the disk can be updated.

Fmaptool is updated to process the new rule. Cbfstool is updated to add the extended CBFS descriptor into the concerned base/parent CBFS and also any files to the extended CBFS.

Alternate Ideas

Extend the FMAP and use one GPT partition

Create a partition in the storage media to extend the FMAP and CBFS. The storage partition is split into clearly delimited sections. Some of these sections accommodate the extended CBFS (eCBFS) that can hold arbitrary-length files. The layout of this partition is described using extended FMAP (eFMAP) and is stored in a separate FMAP section named EFMAP in the SPI Flash. The eFMAP, when present, has a header and area definition - similar to the FMAP. The eFMAP header defines the name, size and number of areas/sections in the extended storage partition. The area definition describes the offset, size and attributes of the individual eFMAP sections. An example AP firmware layout with eFMAP and eCBFS is shown in Fig 4

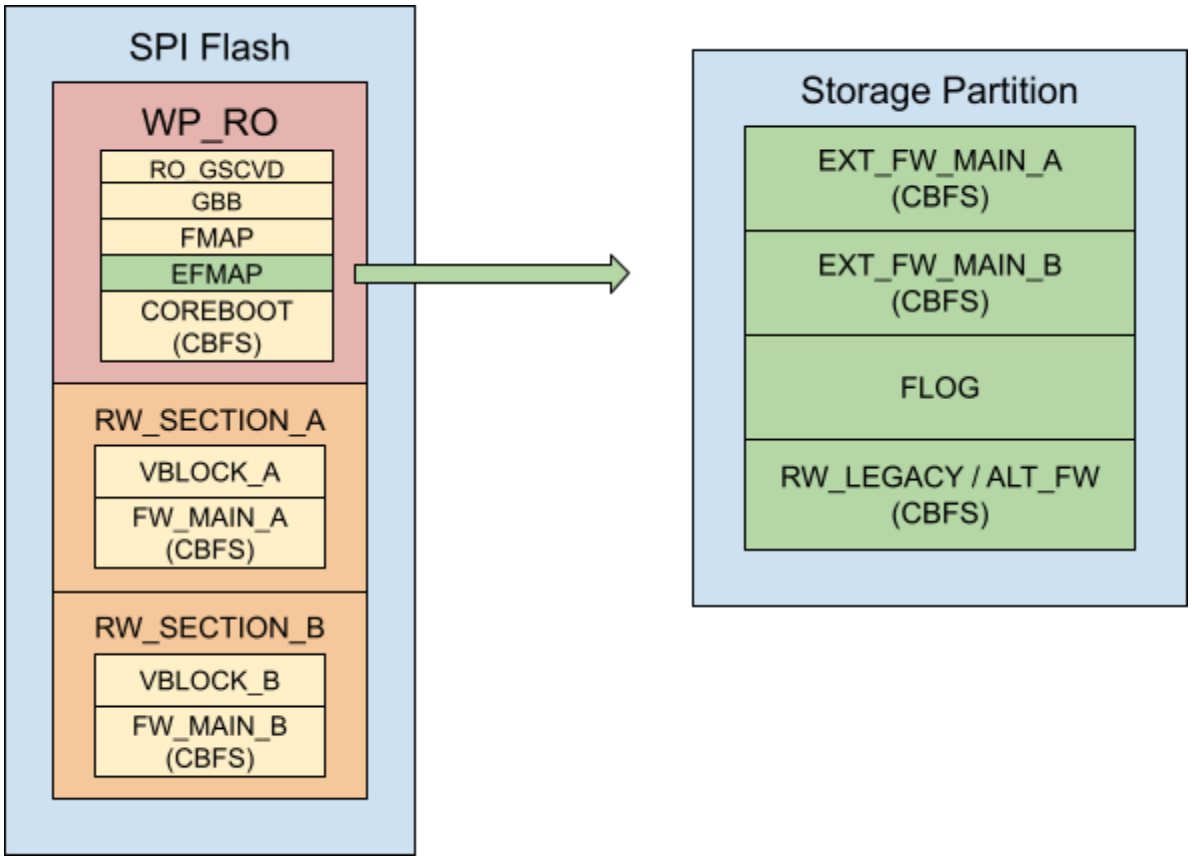

Fig 4 - AP Firmware Layout with Extended FMAP and CBFS

With extended FMAP in the RO section, this tightly couples the firmware in SPI flash with the disk storage - specifically the eFMAP sections, their offset and sizes. Also currently there is a need to move only a few FMAP sections (3 - 5) to the disk storage. One GPT partition per FMAP section provides a natural way of extending that without any additional overhead.

Isolated/Decoupled Verification of eCBFS

This option introduces a custom EXT_VBLOCK* partition while leveraging the existing CBFS verification to verify the extended CBFS. The metadata for each CBFS file in an eCBFS partition are collected together to compute a hash. The choice of the hash algorithm will be made after some consultation with the security team/experts. The computed hash is signed using a private key/token through the signing infrastructure and the signed hash is maintained in the associated EXT_VBLOCK* partition. The public part of the signing key/token is added as a file in the parent CBFS. This way the signing token is verified as part of parent CBFS verification. During the boot process, when a file from eCBFS is required, the public key file from the SPI flash is used to verify the hash signature in EXT_VBLOCK* partition. On successful verification, the hash is in turn used with the CBFS verification to verify the

extended CBFS. Fig 5 shows the entire verification sequence of parent CBFS and extended CBFS.

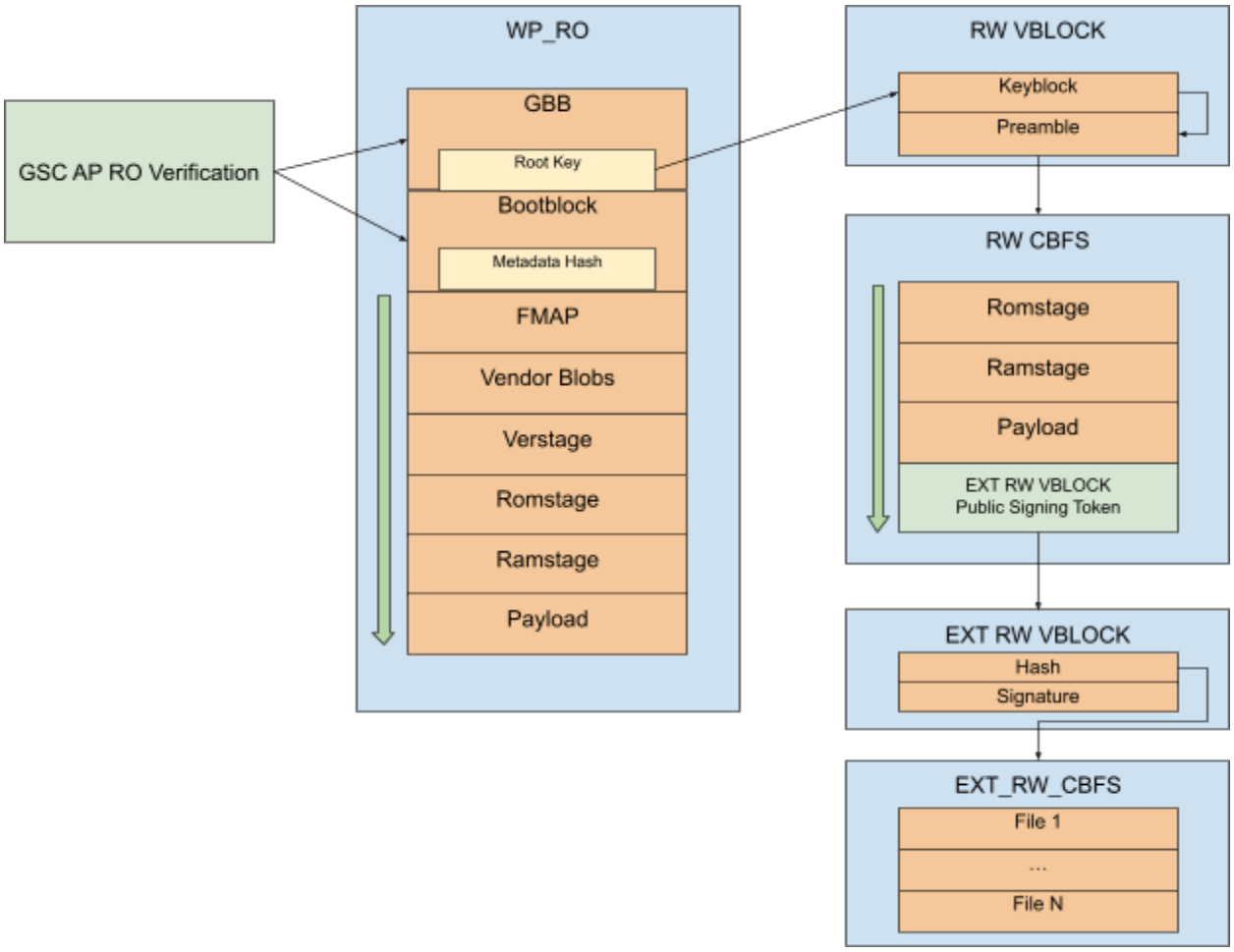

Fig 5 - Verification of eCBFS through custom VBLOCK + CBFS Verification

This option decouples the eCBFS from the parent CBFS and allows for any combination of CBFS (SPI flash) + eCBFS (disk). While this is beneficial from the factory and developer workflow standpoint, having a deterministic combination of CBFS + eCBFS is preferred from the testing and stability standpoint. Another major downside with this option is that it introduces additional signing overhead of the extended BIOS image.

Appendix

Potential Use Cases

When the system enters recovery mode, firmware debug information (e.g. firmware logs) cannot be extracted from the system and is not persistent across the recovery sequence. Since the later stages of bootflow have access to the storage media, this debug information

can be backed up to the storage media and retrieved later into the feedback report.

Reference Coreboot/Payload Code Blocks

Modified CBFS File Lookup Function

C/C++

```
enum cb_err _cbfs_ext_boot_lookup(const char *name,
                  union cbfs_mdata *mdata, struct region_device
*rdev)
{
     struct cbfs_boot_device *ecbd = cbfs_get_ext_boot_device();
     if (!ecbd)
          return CB_CBFS_NOT_FOUND;

     err = cbfs_lookup(&ecbd->rdev, name, mdata, &data_offset,
ecbd->ext_cbfs_hash);
     if (err)
          return err;

     if (rdev_chain(rdev, &ecbd->rdev, data_offset,
                    be32toh(mdata->h.len)))
          return CB_ERR;
     return CB_SUCCESS;
}

enum cb_err _cbfs_boot_lookup(const char *name, bool force_ro,
                  union cbfs_mdata *mdata, struct region_device
*rdev)
{

     /* Lookup the file name in CBFS RW Metadata Cache(MCACHE) */
     /* If RW MCACHE is full, lookup the file name in RW CBFS */

     /* File not found in MCACHE and RW_CBFS. Lookup in eCBFS */
     if (!force_ro && err == CB_CBFS_NOT_FOUND) {
          err = _cbfs_ext_boot_lookup(name, mdata, rdev);
          if (err == CB_SUCCESS)
               return err;
```

```
	}

	/* File not found in RW_CBFS and EXT_RW_CBFS. Fallback to
RO. */
	if (CONFIG(VBOOT_ENABLE_CBFS_FALLBACK) && !force_ro && err
== CB_CBFS_NOT_FOUND) {
		printk(BIOS_INFO, "CBFS: Fall back to RO region for
%s\n", name);
		return _cbfs_boot_lookup(name, true, mdata, rdev);
	}

	if (err) {
		...
		return err;
	}
	...
	return CB_SUCCESS;
}
```

eCBFS device getter

C/C++

```
Coreboot:
=========
const struct cbfs_boot_device *cbfs_get_ext_boot_device(void){
	/* For now return null. Once required storage drivers are in
place and extended CBFS is online then return extended device. */
	return NULL;
}

Libpayload:
===========
```

```
struct cbfs_dev {
      size_t offset;
      size_t size;
      /* Add a read function to struct cbfs_dev so that
cbfs_dev_read can
         be abstracted. */
      ssize_t (*read)(void *buf, size_t offset, size_t size);
}

const struct cbfs_boot_device *cbfs_get_ext_boot_device(void){
      static struct cbfs_boot_device ext_rw;

      if (!lib_sysinfo.ext_cbfs_size)
            return NULL;

      /* eCBFS offset, size, metadata hash read from RW_CBFS and
passed
         through coreboot table */
      if (!ext_rw.dev.size) {
            ext_rw.dev.offset = lib_sysinfo.ext_cbfs_offset;
            ext_rw.dev.size = lib_sysinfo.ext_cbfs_size;
            ext_rw.dev.read = ext_boot_device_read;
      }
      return &ext_rw;
}
```

Proposed flow of building extended BIOS image

1) Add new rules in FMAP to define extended CBFS regions and process it using fmaptool with a new -E option. The FMAP sections with eCBFS will point to the parent CBFS i.e. parent FMAP sections with CBFS and fmaptool will output an artifact(fmap.ext_desc) capturing the eCBFS => parent CBFS map.

```
None
fmaptool -h fmap_config.h -R fmap.desc -E fmap.ext -X fmap.ext_desc
chromeos.fmd fmap.fmap
```

2) Create the extended binary - with either a single eCBFS or an extended BIOS image with multiple eCBFSes. A new -E option to indicate the extended coreboot ROM will be added to the create command in cbfstool. If the binary contains multiple eCBFSes based on an extended FMAP, then the eCBFS => parent CBFS map file is passed as an additional input. In addition to creating the extended binary, this step will also add the eCBFS descriptor file in the parent CBFS. This eCBFS descriptor file will include the eCBFS name, size, metadata hash anchor, hash algorithm etc. The -A option will be added to the create command to specify the hash algorithm.

```
None
cbfstool coreboot.pre create -M fmap.fmap -X ext_coreboot.rom -E fmap.ext -D
fmap.ext_desc
```

3) Add individual files to the extended binary. A new -B option to indicate the parent binary file will be added to the add* command in cbfstool. If the binary contains multiple eCBFSes based on an extended FMAP, then the eCBFS => parent CBFS map file is passed as an additional input. In addition to adding the file to the extended binary, this step will also update the metadata hash in the parent CBFS.

```
None
cbfstool ext_coreboot.rom add -r EXT_FW_MAIN_A -f FILE -n NAME -t TYPE -B
base_coreboot.rom -D fmap.ext_desc -X fmap.ext
```

Suggested changes? Discuss on the g/design-group mailing list.