Blurb on embedded vertran

# Design Goals

The master goal is to support live firmware download/activation with a generic tool while not locking ourselves to the first-released NVMEM layout. (The problem is exacerbated on the current SVI FF platform because we can’t read a good snapshot of NVMEM.)

A secondary but important goal is to eliminate non-value-added work by testers as well as developers not involved in the NVMEM change at the modest to moderate expense of the author of the change.

In that sense, embedded vertran serves the same purpose as offline vertran for offline (ISP) firmware download.

Derived requirements (similar to offline vertran):

1. If vertran fails for any reason, newly downloaded firmware cannot be activated
2. If vertran fails for any reason, the currently running firmware is not affected
3. If vertran succeeds, activating newly downloaded firmware must pick up where the old firmware left off, to the extent possible.

# Basic design

Because of requirements 1 and 2, raw NVMEM is partitioned to include two logical banks, each corresponding to the flash bank for the two firmware images.

During new firmware activation, old (still active) FRAM image is translated to the new (still inactive) image in the opposite bank. The starting image is taken from RAM image of the running firmware because it is faster and more current.

Like in offline vertran, translation is done iteratively from layout n to layout n+1 (or n-1, depending on direction).

Translation may take some time, so it runs as a process (in MNCB sense) and is responsible for process task’s contribution to the watchdog system.

Translation is performed as follows:

1. Choose which vertran to run
   1. If new FRAM layout is greater than the current layout, choose vertran from the new (inactive) image (because the current vertran doesn’t know how to translate forward)
   2. Otherwise, choose vertran of the current image
2. Run a process that will
   1. Create an FRAM image buffer in RAM to translate
   2. Run appropriate vertran to create a buffer with translated FRAM image
   3. Write out the translated FRAM image to the opposite FRAM bank
3. Complete activation and reset the device if and only if step 2 was successful

Steps 1 and 2c require that we find the function to run and the layout table to write to NVMEM; translation forward makes them both located in the inactive image. Thus, the image must have pointers to those two locatable from the running firmware.

# Design limitations

There are two known major limitations:

1. NVMEM layout change on a replaceable “secondary volume” (if any, such as DLT sensor board) is entirely out of scope. It is conceivable that the changes *in principle* must be limited to expansion by new data objects
2. Released SVI FF firmware does not have sufficient hooks in the image, so the existing devices can’t be upgraded directly, but can be upgraded using intermediate build.
3. Certain amount of “seeing into the future” is involved: NVMEM partition size and the data structure and location providing access to other image’s data.

# Free bonus

We can easily control

* whether a particular build is a new baseline and can’t be downgraded (just remove the “translate down” routine)
* whether a particular build is a dead end and will never be upgraded with features (just remove the “translate up” routine)

# Lessons learned

We proved to need some info about image(s) of various sorts. As a pattern, I came to appreciate a scheme where a top-level info contains

1. Signature
2. Version
3. Some content

The content may have pointers to specific info objects of the same nature (except that signature may be optional). The content should be backward-compatible, i.e. can only grow with Version.

The second point is how to find that top-level info. We experimented with this over the years, and the winning solution so far is to append a pointer to it to the interrupt vector table (credit: Ernie).

# Development suggestions

When (re)designing data objects, including adding and/or removing objects, it is beneficial to write image translation first, so that before the new stuff matures, the developers can switch back and forth transparently, especially developers not (yet) involved with the new layout.

# Debugging suggestions

Since the “other vertran” is executed with translating forward, a way to debug is

1. Write translations in the transitional code where vertran will live afterwards anyway
2. At a breakpoint at setting a pointer to vertran, pvertran, replace the content to point to this build’s vertran