Attributes design

Layered approach

- lower level read and write api's to access the device(eeprom/flash/file system)
- NVM class - an abstraction of non volatile memory, with following features
  - Implements "page" level abstraction, wherein a page is a block of memory which is writeable. This is required as some memory devices have the inherent restriction that it can be written in chunks.
  - Caching - We would need a form of caching to reduce device accesses making it time efficient
  - Update tracking - updates are tracked at page level to reduce writes and in turn increase the lifecycle
  - cache_flush method provided to commit changes to memory device, which can be scheduled to run in a low priority task to reduce write overhead and also optimize write cycles
- ATTR_TANK class - an abstraction of attribute tank which stores and retreives the Attributes
  - This includes a metadata, which is always stored at a fixed location - in our case PAGE_0. This is required to keep track of current pointers in memory, init sequence and ATTR_MAP table
  - Current pointers keep track of the next available page and offset in NVM. Addition of a new attribute will take constant time O(1). But there is a possibility of memory fragmentation if the size of attributes changes during runtime, which we will avoid conisdering in this solution
  - INIT_SEQ - is used to indicate if the NV memory is ever initialized or not. It can later be extended to implement versioning as well.
  - ATTR_MAP is a table which stores information to look up each attribute by its id. Ideally this would be a hash table, in our case attribute id itself would be a key, which would be an index to that particular attribute in the ATTR_MAP table. So the look up is constant time O(1).
  - ATTR_MAP stores - size, page and offset in the page, making it possible to store attribute of any size.
- Ideally the NVM and ATTR_TANK would be a singleton classes, but here for the ease of unit test I have not implemented as such

Memory corruption detection

- Each logical page of NVM will have a 1byte checksum for that page at the end. So the data_page_size is less than raw_page_size, accounting for checksum
- Whenever the NVM module reads a page from memory, it calculates the checksum on data and verifies it with checksum stored at the end to detect

memory corruption
- Here a simple 1byte checksum is used, instread of which a higher tolerant CRC and/or combination of hash and CRC can be used to make it more robust

Memory corruption correction

- Approach is to keep redundant copies of the pages, which can be used to correct the corruption
- In our case, the entire pysical memory can be logically divided into “primary” and “secondary” sections, which will be mirror of each other
- Whenever a write is done on specific page, its corresponding secondary page is also updated
- When we detect a corruption during a read, its secondary copy is accessed and updated in primary as well
- If secondary copy is also corrupt, then its the time for some “panic”

Compile and test

./run.sh

System requirements

C++11 gcc compiler