

New Horizons Demo EMS Logger Filesystem

Description of firmware approach with NAND flash

2025-09-11

Rod Hinman

1 Overview

The New Horizons Demo EMS Logger uses a NAND flash IC to store the logged data and other files. The firmware implements a File Allocation Table (FAT) filesystem to be compliant with WHO PQS logger specifications (WHO/PQS/E006/DL01), and this filesystem is available alternately to the logging subsystem and the USB subsystem, depending on USB connection status.

NAND flash has high memory density, but inherently has defects. ICs can be shipped from the factory with sections of memory that will produce uncorrectable errors (so-called bad blocks). The technology also “wears out” as it is erased and rewritten, eventually developing more bad blocks. Therefore, it is advantageous to distribute erase/write cycles over the entire memory array rather than having them concentrated on one block.

2 Our solution

The solution we have adopted uses a flash translation layer to distribute erase/write cycles and to manage bad blocks. One can think of the firmware as being composed of several layers, each providing an abstraction to the layer above it. From highest level to lowest, the layers are:

- The logger application and USB mass storage controller code
- FAT filesystem, provided by STMicroelectronics as part of STM32CubeMX middleware. This code is not included in the zip file that New Horizons has shared.
- FAT-dhara adapter
- Dhara: a flash translation layer
- Dhara-NAND adapter
- Generic driver to work with STM32CubeMX and multiple flash ICs
- Our wrappers around the STM32 hardware abstraction layer

- STM32Cube hardware abstraction layer and FreeRTOS, also not included in the code that New Horizons has shared.

We describe some of these components in the sections below. The system is more complex than it might be if it has been written from scratch for a single application. It was derived from a code base designed to allow adaptation to different microcontrollers, hardware abstraction layers, and NAND flash ICs.

2.1 FAT-dhara adapter

These functions ensure that dhara works with the FAT filesystem. Two directories/files in the repository contain the relevant functions:

File **src/fatfs/dhara_diskio.c** contains functions to adapt between those expected by the FAT filesystem and those provided by the flash translation layer.

File **stm32cubemx/sl2/FATFS/Target/user_diskio.c**. This file is *not* in the released code because it is part of STMicroelectronics' CubeMX middleware code and covered by their "Ultimate Liberty" license, which is not really about liberty. This automatically generated file contains locations for the user to insert calls to the user's memory implementation. In our case, we inserted calls to the functions that are in **dhara_diskio.c**.

2.2 Dhara

This is an open-source library written by Daniel Beer that provides a flash translation layer and bad block management. It is incorporated into the Demo EMS Logger code in directory **third_party/dhara**. The file **map_internals.txt** contains an explanation of how some parts of it work. It stores the mapping between logical page numbers and physical page numbers in metadata in the main flash memory bytes (not the 64 "spare" bytes per page that the IC has for error-correction and other functions).

Dhara requires that the user provide implementations of the functions prototyped in file **third_party/dhara/dhara/dhara_nand.h**. These functions should provide operations on the specific NAND IC (reading and writing pages, checking bad blocks, etc.).

2.3 Dhara adapter

This is part of the Demo EMS Logger code that provides the functions needed by dhara, translating between dhara and the NAND driver. The relevant files are in directory **src/dhara**.

File **dhara_nand.c** has the functions needed by dhara to read and write from the NAND IC, a Winbond W25N01GWZEIG. The read implementation buffers a full page so that consecutive reads from the same page do not need to access the NAND.

One consideration for this IC is that dhara requires an `is_free()` function to denote whether a page has been programmed or not since erasure. This IC does not natively have that capability, so we are using one of the “spare” bytes as a “programmed” marker. See the comment within function `dhara_nand_prog()`. In reviewing the code for this writeup, I notice that there are some incorrect comments. For instance, the IC has only 64 extra bytes per page, not 127.

File **dhara_utils.c** has static data structures to hold information about the NAND, the microcontroller’s SPI peripheral, as well as buffers to hold data read to and from it. It also has some utility functions to initialize on boot and finalize the NAND after write operations.

File **dhara_metadata_cache.c** provides a cache of dhara’s metadata for the last few pages that have been read. Having this cache can reduce the number of reads that we need to perform from the NAND IC. Since each read comes with an energy cost, this minimizes overall power consumption.

File **dhara_nand_timer.c** is a mechanism to reduce flash power consumption by turning it off after periods of inactivity. The `nand_enable()` function applies power to the flash, does some preliminary communication, and starts a FreeRTOS timer. If the timer expires, the callback function removes power from the flash. Firmware can also use the `nand_hold_active()` function to override the timeout in case there are long-running operations that are expected to access the NAND again, but with a longer delay than the timeout.

File **dhara_nand_timer_none.c** is a version of the above file that is linked when compiling the bootloader, but is not used by the main application. Since the bootloader operates briefly, there is no need for timer functions, and their implementations in this file are empty.

2.4 Generic driver to work with STM32CubeMX and multiple flash ICs

This driver is in file **src/drivers/nand/nand.c**. It handles generic flash functions like reading a page from the NAND IC’s flash into its page buffer and reading that data into the microcontroller. It is configured by defined constants and by supplying functions that work with the particular hardware/middleware platform the driver is running on.

That “platform” code is in directory **src/drivers/nand/** in files **nand_platform.h**, **nand_platform_stm_spi.h**, and **nand_platform_stm.h**. These functions send commands

and data over SPI using the STM32 HAL and, in some cases, our wrappers (next section), and receive data responses.

2.5 Our wrappers around the STM32 hardware abstraction layer

We have written some wrappers around the STM32CubeMX HAL to allow operation in different conditions: whether FreeRTOS is running, interrupts are enabled, and whether direct memory access is desired. The functions for SPI are in file **src/utls/task_io_stm/task_io_spi.c**.