zilog
Embedded in Life
An IXYS Company

MultiMotor Series

Application Note

Implementing a Data Logger with Spansion SPI Flash

AN036001-0513

Abstract

This application note shows how to implement a data logger to record persistence data for future analysis.

The purpose of a data logger is to automatically collect data toward providing a comprehensive review of conditions upon recording. As an example, the *black box* used in most aircraft records data containing the sequence of events occurring immediately preceding a crash event. By the same token, a weather data logger maintains a record of environmental changes for tracking storms and weather-related conditions.

To see real-world implementations of this data logger application, refer to the MultiMotor Series application notes listed in Table 1.

Table 1. MultiMotor Series Application Notes that apply a Data Logger

Document Number	Document Title
AN0353	3-Phase Sensorless Brushless DC Motor Control Application Note
AN0355	Hall Sensor Sinusoidal PWM Modulation Brushless DC Motor Control
AN0356	3-Phase Hall-Sensor Brushless DC Motor Control

Note: The source code file associated with this application note, AN0360-SC01.zip, is available for download from the Zilog website. This source code has been tested with version 5.0.1 of ZDS II for ZNEO MCUs. Subsequent releases of ZDS II may require you to modify the code supplied with this application note.

Features

This data logger application offers the following features:

- Data packets stored on SPI Flash use a *circular* buffer approach
- Provides capability to retrieve data packets in either forward or reverse direction

Discussion

To implement a data logger, there must be a way to keep track of all of the data and to replace old data with new data while maintaining a history. To accomplish this requirement suggests the use of a Circular Buffer.

A *circular* buffer tracks beginning and ending records; therefore, the starting point may appear anywhere within the blocks. Again, for simplicity, a *Beginning Record* will always point to the start of a sector because erases only occur sector by sector. An *Ending Record* will point to the location of the next record to write to. To dump the records is a matter of reading the Beginning Record and each subsequent record in sequence (wrapping from the top to the bottom). To present it in reverse order, start at *Ending Record – 1* and read each subsequent previous record.

To add the datalogger to any project, add the data logger files to the project, then add the initialization of the Flash memory, a timer variable to track intervals, a command to dump the data, and a routine to pass record information to be written.

Hardware

The datalogger code is written specifically for the Spansion SPI Flash device and Zilog's ZNEO family of microcontrollers. These two devices are the only required hardware items.

SPI Flash

Flash data is written to the Spansion SPI Flash device, part number S25FL032P. This device consists of 64 uniform 64KB sectors with its two (top or bottom) 64KB sectors further segmented into thirty-two 4KB subsectors. This arrangement provides for the ability to erase all data in Flash memory, erase a 64KB sector, erase a 4KB subsector, or erase an 8KB subsector. The addressing method is to use a 24-bit address in Big Endian format in the `0x00000000` to `0x003FFFFF` address range.

The SPI Flash device is controlled through commands as an SPI slave device, using Mode0 or Mode3 of the SPI protocols. This device receives data through a *page write* command; each page is between 1 and 256 bytes in length. A page will remain within the sector of a requested address (i.e., it wraps on its current sector). For example, if you write 256 bytes at address `0x0001FFF0`, the first 16 bytes will be written within the `0x0001FFF0–0x0001FFFF` address range, and the remainder will be written at address `0x00010000`.

The device can read any/all bytes by specific address and continue to read data. As long as the Chip Select pin is Low, every read will return the next byte. This methodology allows a user to read between 1 and 4MB (the entire contents of Flash memory) in one command.

There are a number of other features that are included on the Spansion SPI Flash unit that we have not incorporated into this application, for various reasons. These features include:

- Dual Output and Quad Output SPI protocols
- 16 bytes of One-Time Programmable (OTP) area for permanent, secure identification, as well as 490 bytes of OTP for other permanent information

- Fast reads, up to 104MHz
- Data block protection
- Deep Power Mode

Firmware Implementation

The datalogger firmware only implements datalogger functionality. The interface portions that must be added to your project are discussed below but are not part of the firmware files.

The data logger consists of four files:

- `SPIFlash.c` and `SPIFlash.h` provide a low-level interface to the SPI Flash device, and `datalogger.c`
- `datalogger.h` provide the data logger implementation

Note: This document only discusses the data logger implementation and the code modifications necessary to implement the data logger. To learn more about a real-world implementation of the data logger, see the 3-Phase Sensorless Brushless DC Motor Control Application Note (AN0353).

Data Packets to Store

The data packet that is to be stored in the circular buffer will typically be in the form of a structure. This structure must always be an even divisor of the Flash memory page size; for the Spansion part, this page size is 256 bytes.

A structure in the `datalogger.h` file called RUNNINGDATA defines a 16-byte data packet for storage and retrieval. A discussion of this structure is strictly a demonstration of a specific implementation for clarity, and is beyond the scope of this document.

The RUNNINGDATA structure is defined as follows:

```
struct RUNNINGDATA {
 unsigned char state;         // State of the motor
 unsigned char temperature;   // Temperature
 unsigned char voltage;       // Voltage
 unsigned short int speed;    // Speed of the motor
 struct TIMESTAMP stamp;      // Time stamp variable for tracking
                              // records
 struct TIMESTAMP motorlife;  // Time stamp for tracking motor run
                              // time
 unsigned char reserve;       // Add padding for even divisible of
                              // 256 (16 bytes)
};
```

The datalogger assumes no actual real-time clock, and therefore uses a variable to mimic the RTC. This variable is named `clock`, and it is up to the main application to update it every second. The `clock` variable is initialized to the time stamp of the last record in Flash memory at startup in the `InitDataLogger()` initialization routine.

To conserve space, a time stamp structure (i.e., `struct TIMESTAMP`) is used to maintain the time in a five-byte structure that will extend up to 256 years. This structure allows both a time stamp for a log entry and a time stamp for the total run time of the motor; it also maintains total time information down to 10 bytes, allowing 6 bytes for other motor control information.

An additional *normalize* function acts to normalize the time stamp to keep it valid. For example, if you were to add one second to the time stamp, this normalize function will accordingly adjust to account for the change in hours and years. By using this normalize function in the TIMESTAMP structure, the time stamp can be replaced with any structure, and only the normalize function would need to change to accommodate.

Circular Buffer

The data logger uses a *circular* buffer (also known as a ring buffer) approach to saving records to Flash memory. This circular buffer allows the current data to replace older data after Flash memory is full. The result is that the most recent data is always stored.

A circular buffer has no absolute beginning or ending addresses. The physical buffer does, of course, but the code wraps from the last physical address to the beginning physical address. Due to the circular nature of this type of buffer, the beginning address of the data could be anywhere within the physical buffer. The ending address of the data in the circular buffer could be less than or greater than the physical beginning address. When the circular buffer ending address equals the circular buffer beginning address, the beginning address is incremented and the new data replaces the old data.

Flash memory only allows erasure on a sector-by-sector basis; therefore the implementation will occur by sectors. This implementation will provide 63 sectors of valid data at any time (assuming Flash memory has been filled). The 64th sector is where the current data is being written.

The circular buffer implementation has two global variables: `EndDataAddress` and `BegDataAddress`. The `EndDataAddress` variable always points to the next record to be written. The `BegDataAddress` variable always points to the start of a filled sector. The addresses are physical addresses and, as they are modified, they are wrapped to the beginning if the end has been reached.

Given that the SPI Flash function can perform a subsector erase of two sectors, two macro values would be added to specify the physical addresses of the beginning and ending of Flash memory (i.e., `DATAADDRESSSTART, DATAADDRESSEND`, respectively). These macro values allow a developer to change the physical address of the beginning address to `0x00010000` and thereby use the first sector for other data yet continue with the data logger in the remainder of Flash memory.

Initialization

The initialization of the circular buffer is critical to set `BegDataAddress` to the correct sector of the beginning data and `EndDataAddress` to the next unused record. Because the beginning data could be in any sector (remember that it is circular and therefore contains no absolute start value), the last valid record must be determined before a position can be determined from which to start looking for the beginning record location. Erases can only occur sector by sector; therefore, `BegDataAddress` will always point to the beginning of the sector.

The cycle begins by retrieving the last record of each sector. The first occurrence of an invalid record (i.e., data that was erased) is the current sector to write to. To find the actual record to write to, the algorithm goes back through the records to find the first record with valid data. The `EndDataAddress` will be set to point to the address of the next record.

After the `EndDataAddress` has been found, the `BegDataAddress` can be located. The search starts at the beginning of the next sector and cycles through the first record of every sector until a valid record is found. This record is the address of the `BegDataAddress`.

If the system is shut down during an update, the last record of the sector may possibly have been written, but quite possibly the next sector may have not been erased. This situation prevents finding an open record to start a search from. To account for this issue, the last record of every sector must be checked to find the earliest time stamp. This time stamp will indicate the sector to erase, thereby identifying an `EndDataAddress` location. The `BegDataAddress` will then be the beginning of the next sector.

Update Record

To update a record in the circular buffer, write the data packet to the address pointed to by `EndDataAddress`. Next, increase the `EndDataAddress` by one data packet to point to the next data packet to write to.

However, because this methodology can result in exceeding the buffer's physical boundaries, check to ensure that `EndDataAddress` has not exceeded the maximum Flash address. If it has, reset `EndDataAddress` to the beginning of Flash memory.

Noting when `EndDataAddress` equals `BegDataAddress` must also be considered. When these two address are the same value, the buffer will be full; therefore the sector that the `EndDataAddress` is pointing to must be erased, and `BegDataAddress` must be moved to the next sector. If this next sector should come after the ending physical sector, the `BegDataAddress` is set to the beginning physical address.

Output

This application uses the console to output the data from Flash memory to the console. Start at the `EndDataAddress`, and implement a loop to perform the following procedure.

1. Specify the next previous record. If this record is greater than or equal to `DATAADDRESSSTART`, set the record to `DATAADDRESSEND` minus one record.
2. Read the record. If the record is not valid, exit the loop.

3. Print the record to the screen. Check for a Ctrl-C to see if this print task should be stopped. If there is a Ctrl-C, exit the loop.
4. Call `LogData` so that the log update can continue to furnish information to the the motor at appropriate intervals.
5. If the record equals `BegDataAddress`, the end of the valid data has been reached; therefore, exit the loop.
6. After exiting the loop, return.

Reading and Writing Flash

To read Flash memory, first wait until Flash memory is ready, then read the status register until the busy flag is no longer set or until there are write/erase errors.

Note: If there is a write or erase error, the error will specify that the SPI Flash is either bad or is going bad; the busy flag will not be cleared. In this case, an error code will be returned; this error will never be cleared and Flash memory will be marked as bad.

After the busy flag is cleared, send the `READ` command followed by the 24-bit address, then read data until the number of bytes requested are read.

To write Flash memory, wait until Flash memory is ready (as was done previously), then place Flash memory into write mode by sending a Write Enable command. Next send a Page Write command followed by a 24-bit address, then send the data, one byte at a time, until all bytes requested have been sent.

To erase Flash memory, wait until Flash memory is ready, then place Flash memory into write mode. Next, either send a Bulk Erase (BE) command to erase all Flash memory, or send a Sector Erase (SE) command to erase just a single sector.

Datalogger Implementation

To implement the datalogger into an existing application, observe the following procedure.

1. Include the four datalogger files in the existing project (hereafter referred to as *the project*).
2. In the main function, place the initialization routine, `InitDataLogger()`, after any other initialization required for your project. In the unlikely event that the Flash initialization fails, the Flash should be disabled. For example, an external unsigned char variable `isFlashValid` can be used to ensure that reads or writes do not occur on a Flash unit that has failed.
3. If the RTC is being emulated, add a timer to update the external variable `clock` every second.

4. Add a function to create a data packet to write to Flash, with the data to populate the packet. The actual write operation to the datalogger will typically be written at specific intervals. This function would be required to verify the interval had passed.
5. In the main infinite while loop, add a call to the function to record data to the datalogger.
6. If there is a requirement to output the data to the console, modify the function in the `datalogger.c` file called `OutputDataLogger()` to present the data from the record in a meaningful way.

The following code snippet provides an example of the requirements for implementing the datalogger:

```
// Create local file variable for seconds
static unsigned char second=0;
// Create local file variable to keep track of Flash status
static unsigned char isValidFlash = TRUE;


// In the timer interrupt service routine, update the second
  variable used by logdata() function
 if (count == second)
  ++second;


// Function to write current state record to the datalogger at
  specific intervals

void LogData(void)
{
 struct RUNNINGDATA curState; // Local variable to populate for
                              // writing record

 if (seconds >= INTERVALTOWRITERECORD) // Controls how often data
                              // is recorded
 {
  if (isValidFlash)           // Make sure Flash is valid
  {
  clock.seconds += seconds;   // Emulate RTC
   seconds = 0;
   NormalizeTimeStamp(&clock);// Normalize stamp to account for
                              // new second
  .....  // Add data to curState variable here

   if (UpdateRecord(&curState) != SPI_SUCCESS)
   {
    isValidFlash = FALSE;     // If we fail, there is something
                              // wrong with the Flash part
   }
  }
 }
}

void main(void)
{

 .....  // Initializations
 if (InitDataLogger()) != SPI_SUCCESS)
 {
  isValidFlash = FALSE;
 }

 .....  // Continue with other code

 while(1)                     // Infinite while loop
 {
  LogData();

  .....  // Continue with other code to execute in while loop
 }
}
```

The OutputDataLogger Function

An `OutputDataLogger` function is included in the `datalogger.c` file for convenience. This function handles the retrieval and printing of each data packet to the console.

The function starts with defining a temporary data packet variable, based on the structure defined in the `datalogger.h` file, to allow the structure to change without having to change the code. The function uses `printf()` calls to output messages to the console.

To retrieve data, first compare the values contained in the datalogger variables `BegDataAddr` and `EndDataAddr`. If these two values are the same, there is no data; otherwise, we can step through each record.

Start by initializing the loop address pointer to the associated pointer. To go in reverse order (i.e., present the last record first), set the loop address pointer to `EndDataAddr`; otherwise, set it to `BegDataAddr`. Next, step through each data packet by adding or subtracting the size of the data packet to/from the loop address pointer (i.e., loop to the physical beginning, when required) and reading the data packet at that address. Verify that the record is valid. When the contents of Flash are erased, the values are set to all ones (`0xFF`). By making sure a specific point in the record is always set, such as *state*, the record can be verified as valid when it is not set to `0xFF`.

Assuming the data packet is valid, then it is printed out to the console via a `printf()` call with a formatted string.

After printing the data packet, check the UART for the Ctrl-C character (`0x03`). If this character is found, exit the loop; otherwise, continue stepping through the data packets until encountering an invalid packet, or return to the location of the beginning address.

This kind of output can take a long time. Because we do not want to stop logging data while these packets are being displayed, call the `LogData()` function in the main application to continue updating the data accordingly.

Summary

The application implements a data logger using a Spansion SPI Flash device. This data logger keeps track of the motor state and logs onto Flash memory in a circular buffer type implementation, and is segregated to allow it to be used elsewhere with minimal implementation requirements.

References

The documents associated with this application note are listed below and are available free for download from the Zilog website unless otherwise noted.

- 3-Phase Sensorless Brushless DC Motor Control Application Note (AN0353)
- 32 Mbit CMOS 3.0 V Flash Memory with 104 MHz SPI Multi I/O Bus Data Sheet (S25FL032P_00), available on the Spansion website

Customer Support

To share comments, get your technical questions answered, or report issues you may be experiencing with our products, please visit Zilog's Technical Support page at http://support.zilog.com.

To learn more about this product, find additional documentation, or to discover other facets about Zilog product offerings, please visit the Zilog Knowledge Base at http://zilog.com/kb or consider participating in the Zilog Forum at http://zilog.com/forum.

This publication is subject to replacement by a later edition. To determine whether a later edition exists, please visit the Zilog website at http://www.zilog.com.

Warning: DO NOT USE THIS PRODUCT IN LIFE SUPPORT SYSTEMS.