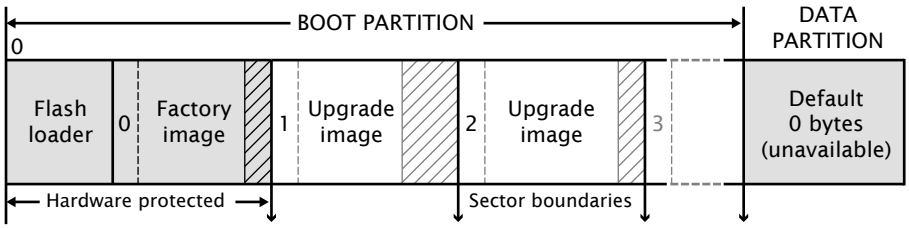
libflash API

IN THIS DOCUMENT

The libflash library provides functions for reading and writing data to SPI flash devices that use the xCORE format shown in the diagram below.

Figure 1: Flash format diagram



All functions are prototyped in the header file <flash.h>. Except where otherwise stated, functions return 0 on success and non-zero on failure.

1 General Operations

The program must explicitly open a connection to the SPI device before attempting to use it, and must disconnect once finished accessing the device.

The functions `fl_connect` and `fl_connectToDevice` require an argument of type `fl_SPIPorts`, which defines the four ports and clock block used to connect to the device.

```
ypedef struct {
in buffered port:8 spiMISO;
out port spiSS;
out port spiCLK;
out buffered port:8 spiMOSI;
clock spiClkblk;
fl_SPIPorts;
```


Publication Date: 2013/11/11 REV C

`int fl_connect(fl_SPIPorts *SPI)`

`fl_connect` opens a connection to the specified SPI device.

`int fl_connectToDevice(fl_SPIPorts *SPI, fl_DeviceSpec spec[], unsigned n)`

`fl_connectToDevice` opens a connection to an SPI device. It iterates through an array of *n* SPI device specifications, attempting to connect using each specification until one succeeds.

`int fl_getFlashType(void)`

`fl_getFlashType` returns an `enum` value for the flash device. The enumeration of devices known to libflash is given below.

```
typedef enum {
  UNKNOWN = 0,
  ALTERA_EPCS1,
  ATMEL_AT25DF041A,
  ATMEL_AT25FS010,
  ST_M25PE10,
  ST_M25PE20,
  WINBOND_W25X40
} fl_FlashId;
```

If the function call `fl_connectToDevice(p, spec, n)` is used to connect to a flash device, `fl_getFlashType` returns the parameter value `spec[`*i*`].flashId` where *i* is the index of the connected device.

`unsigned fl_getFlashSize(void)`

`fl_getFlashSize` returns the capacity of the SPI device in bytes.

`int fl_disconnect(void)`

`fl_disconnect` closes the connection to the SPI device.

2 Boot Partition Functions

By default, the size of the boot partition is set to the size of the flash device. Access to boot images is provided through an iterator interface.

`int fl_getFactoryImage(fl_BootImageInfo *bootImageInfo)`

`fl_getFactoryImage` provides information about the factory boot image.

`int fl_getNextBootImage(fl_BootImageInfo *bootImageInfo)`

`fl_getNextBootImage` provides information about the next upgrade image. Once located, an image can be upgraded. Functions are also provided for reading the contents of an upgrade image.

`unsigned fl_getImageVersion(fl_BootImageInfo *bootImageInfo)`

`fl_getImageVersion` returns the version number of the specified image.

`int fl_startImageReplace(fl_BootImageInfo *, unsigned maxsize)`

`fl_startImageReplace` prepares the SPI device for replacing an image. The old image can no longer be assumed to exist after this call.

Attempting to write into the data partition or the space of another upgrade image is invalid. A non-zero return value signifies that the preparation is not yet complete and that the function should be called again. This behavior allows the latency of a sector erase to be masked by the program.

`int fl_startImageAdd(fl_BootImageInfo*, unsigned maxsize, unsigned padding)`

`fl_startImageAdd` prepares the SPI device for adding an image after the specified image. The start of the new image is at least padding bytes after the previous image.

Attempting to write into the data partition or the space of another upgrade image is invalid. A non-zero return value signifies that the preparation is not yet complete and that the function must be called again. This behavior allows the latency of a sector erase to be masked by the program.

`int fl_startImageAddAt(unsigned offset, unsigned maxsize)`

`fl_startImageAddAt` prepares the SPI device for adding an image at the specified address offset from the base of the first sector after the factory image.

Attempting to write into the data partition or the space of another upgrade image is invalid. A non-zero return value signifies that the preparation is not yet complete and that the function must be called again.

`int fl_writeImagePage(const unsigned char page[])`

`fl_writeImagePage` waits until the SPI device is able to accept a request and then outputs the next page of data to the device. Attempting to write past the maximum size passed to `fl_startImageReplace`, `fl_startImageAdd` or `fl_startImageAddAt` is invalid.

`int fl_writeImageEnd(void)`

`fl_writeImageEnd` waits until the SPI device has written the last page of data to its memory.

`int fl_startImageRead(fl_BootImageInfo *b)`

`fl_startImageRead` prepares the SPI device for reading the contents of the specified upgrade image.

`int fl_readImagePage(unsigned char page[])`

`fl_readImagePage` inputs the next page of data from the SPI device and writes it to the array page.

`int fl_deleteImage(fl_BootImageInfo* b)`

`fl_deleteImage` erases the upgrade image with the specified image.

3 Data Partition Functions

All flash devices are assumed to have uniform page sizes but are not assumed to have uniform sector sizes. Read and write operations occur at the page level, and erase operations occur at the sector level. This means that to write part of a sector, a buffer size of at least one sector is required to preserve other data.

In the following functions, writes to the data partition and erasures from the data partition are not fail-safe. If the operation is interrupted, for example due to a power failure, the data in the page or sector is undefined.

`unsigned fl_getDataPartitionSize(void)`

`fl_getDataPartitionSize` returns the size of the data partition in bytes.

`int fl_readData(unsigned offset, unsigned size, unsigned char dst[])`

`fl_readData` reads a number of bytes from an offset into the data partition and writes them to the array dst.

`unsigned fl_getWriteScratchSize(unsigned offset, unsigned size)`

`fl_getWriteScratchSize` returns the buffer size needed by `fl_writeData` for the given parameters.

```
int fl_writeData(unsigned offset,
                 unsigned size,
                 const unsigned char src[],
                 unsigned char buffer[])
```

`fl_writeData` writes the array `src` to the specified offset in the data partition. It uses the array `buffer` to preserve page data that must be re-written.

3.1 Page-Level Functions

`unsigned fl_getPageSize(void)`

`fl_getPageSize` returns the page size in bytes.

`unsigned fl_getNumDataPages(void)`

`fl_getNumDataPages` returns the number of pages in the data partition.

`unsigned fl_writeDataPage(unsigned n, const unsigned char data[])`

`fl_writeDataPage` writes the array data to the *n*-th page in the data partition. The data array must be at least as big as the page size; if larger, the highest elements are ignored.

`unsigned fl_readDataPage(unsigned n, unsigned char data[])`

`fl_readDataPage` reads the *n*-th page in the data partition and writes it to the array data. The size of data must be at least as large as the page size.

3.2 Sector-Level Functions

`unsigned fl_getNumDataSectors(void)`

`fl_getNumDataSectors` returns the number of sectors in the data partition.

`unsigned fl_getDataSectorSize(unsigned n)`

`fl_getDataSectorSize` returns the size of the *n*-th sector in the data partition in bytes.

`unsigned fl_eraseDataSector(unsigned n)`

`fl_eraseDataSector` erases the *n*-th sector in the data partition.

`unsigned fl_eraseAllDataSectors(void)`

`fl_eraseAllDataSectors` erases all sectors in the data partition.