

## 14.2 libflash Library API

XFLASH uses the libflash library to read and write data to the flash device. Developers who need to provide field upgrades to the flash can access this library from their applications.

### 14.2.1 General Operations

The program must first call `fl_connect()` or `fl_connectToDevice()`. The program must call `fl_disconnect()` when it has finished accessing the flash device.

The identity of the FLASH chip and its characteristics can be queried when it is connected using `fl_getFlashType()`.

The boot partition size must be set using `fl_setBootPartitionSize()` unless the default value (64KB) is being used.

By default, function calls operate in a synchronous mode where calls return only when the required operation is complete. Functions can be made to return as soon as possible by switching to asynchronous mode using `fl_setAsynchronousMode()`. In asynchronous mode applications can check whether a previous operation is still completing using `fl_getBusyStatus()`.

```
int fl_connect(fl_PortHolderStruct* pHolder)
    Connects to the SPI device defined in the fl_PortHolderStruct
    structure in the libflash library. Returns 0 on success.
```

```
int fl_connectToDevice(fl_PortHolderStruct* pHolder,
    fl_DeviceSpec* deviceSpecPtr, unsigned int specCount)
    Connects to a device description defined in the fl_DeviceSpec
    structure. Returns 0 on success.
```

```
int fl_disconnect()
    Disconnects from the SPI device. Returns 0 on success.
```

```
int fl_getFlashSize()
    Returns the capacity (in bytes) of the flash device.
```

```
int fl_getFlashType()
    Returns an enum value for the flash device.
    The enumeration of known flash devices is given below.
```

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

### Port Holder descriptor

```
typedef struct
{
    in buffered port:8 spiMISO;
    out port spiSS;
    out port spiCLK;
    out buffered port:8 spiMOSI;
} fl_PortHolderStruct;
```

### Asynchronous Mode Functions

```
int fl_setAsynchronousMode(int state)
    Set state to 0 to put libflash in synchronous mode. Non-0 values put the library in asynchronous mode. Returns 0 on success.
```

```
int fl_getBusyStatus()
    Returns 0 if the previous operation has completed. No other libflash function can be called until the device is ready.
```

```
unsigned int fl_getFullStatus()
    Returns the full content of the device's status register.
```

### 14.2.2 Read/Write/Erase Functions

All flash devices are assumed to have uniform page sizes but are not assumed to have uniform sector sizes. Read/write operations occur at the page level, and erasure occurs at the sector level. Page level reading is provided with the function `fl_readPage()` and writing with the function `fl_writePage()`.

#### Device level functions

```
int fl_eraseAll()
    Deletes all data from the flash device. Returns 0 on success.
```

```
int fl_setProtection(int protect)
    Protects the device as much as possible. Returns 0 on success.
```

#### Sector level functions

```
int fl_getNumSectors()
    Returns the number of sectors on the device.
```

```
int fl_getSectorSize(int sectorNum)
    Returns the size (in bytes) of a particular sector.
```

```
int fl_getSectorAddress(int sectorNum)
    Returns the address of a particular sector.
```

```
int fl_eraseSector(int sectorNum)
```

Erases a sector. Returns 0 on success.

```
int fl_setSectorProtection(int sectorNum, int protect)
```

Set protect to 1 to protect the sector or 0 to unprotect it. Returns 0 on success.

### Page level functions

```
int fl_getNumPages()
```

Returns the number of pages.

```
int fl_getPageSize()
```

Returns the page size (in bytes).

```
int fl_writePage(unsigned int address, const unsigned char data[]);
```

Programs a page at the given address. The data array must be at least as big as the page size. Returns 0 on success.

```
int fl_readPage(unsigned int address, unsigned char data[]);
```

Reads a page at the given address. The data array must be at least as big as the page size. Returns 0 on success.

### 14.2.3 Boot and Store Functions

Access to boot images is provided through an iterator-like interface. The function `fl_getFirstBootImage` is used to find the factory image, and the function `fl_getNextBootImage` finds the next image. The start address in flash, the size and a version identifier for each image must be provided.

Once located, an image can be erased using `fl_eraseNextBootImage()` or a new image added after it using `fl_addBootImage()`, providing sufficient room is available. An image can be added in single pages using `fl_startAddingBootImage()`, `fl_addPageToNewBootImage()` and `fl_addFinalPageToNewBootImage()`.

#### Basic information

```
unsigned int fl_setBootPartitionSize(unsigned int s)
```

Sets the size of the boot partition.

```
unsigned int fl_getBootPartitionSize()
```

Returns the size of the boot partition.

```
unsigned int fl_getStorePartitionBase()
```

Returns the base of the persistent store partition.

```
unsigned int fl_getStorePartitionSize()
```

Returns the size of the persistent store partition.

### Query and modify the boot partition functions

`int fl_getFirstBootImage(fl_BootImageInfo* bootImageInfo)`  
Returns information about the first boot image (the factory image). Requires a pointer to an `fl_BootImageInfo` structure which the calls populate. Returns 0 if an image is found.

`int fl_getNextBootImage(fl_BootImageInfo* bootImageInfo)`  
Returns information about the next boot image. Requires a pointer to an `fl_BootImageInfo` structure which the calls populate. Returns 0 if an image is found.

`int fl_eraseBootImage(fl_BootImageInfo* bootImageInfo)`  
Erases a boot image. Returns 0 on success.

`int fl_compactBootImages()`  
Compacts the boot image space to remove erasure gaps.

`int fl_addBootImage( fl_BootImageInfo* bootImageInfo, unsigned int imageSize, unsigned int version, unsigned int (getData)(void*, unsigned int, unsigned char*), void* userPtr );`  
Adds a new boot image after the supplied one. The function pointer is used as a callback. The value of `userPtr` is passed as the first parameter to `getData()` on every call. Returns 0 on success.

`int fl_startAddingBootImage( fl_BootImageInfo* bootImageInfo, unsigned int imageSize, unsigned int version, int stage );`  
`int fl_addPageToNewBootImage( unsigned char page[] );`  
`int fl_addFinalPageToNewBootImage( unsigned char page[] );`  
Adds a new boot image after the supplied one. An alternative to `fl_addBootImage` that does not use a callback or cause the caller to spend a long time waiting inside libflash. `fl_startAddingBootImage()` should be called first with a value of 0 for stage. If it returns <0 then it has encountered an error. If it returns >0 it should be called again but with the previous return code as the final parameter. When it returns 0 all the necessary work has been completed. Thereafter, `fl_addPageToNewBootImage()` should be called for each page of data in the image except for the final page for which `fl_addFinalPageToNewBootImage()` should be called. `fl_addPageToNewBootImage` and `fl_addFinalPageToNewBootImage` return 0 on success.

### Boot image descriptor

```
typedef struct {
    unsigned int startAddress;
    unsigned int size;
    unsigned int version;
} fl_BootImageInfo;
```

#### 14.2.4 Persistent Storage Functions

Data in the persistent storage is accessed by its address (offset from the start of the persistent store) and size. Reads can be made from any location and specified with any size using `fl_readStore()`. Flash devices do not support erasure of arbitrary ranges, so some RAM must be used to preserve the content of the erasure unit (generally a sector) during the read/erase/write cycle. The function `fl_getWriteScratchSize` is used to obtain the necessary size of this buffer for any given write operation. The function `fl_writeStore()` is used to write to the store.

Persistent store handling is not robust against partially complete writes, for example due to power failure.

The libflash library does not support features to obtain the data stream from an external device. The API does not impose any structure on the persistent storage, so any file system, database or atomic-update must be user-defined and layered on top.

```
int fl_readStore(unsigned int offset, unsigned int size, unsigned
char buffer[]);
```

Reads an arbitrary number of bytes from the store. Returns 0 on success.

```
int fl_getWriteScratchSize(unsigned int offset, unsigned int
size)
```

Returns the scratch buffer size needed to use `fl_writeStore()` with the given parameters.

```
int fl_writeStore(unsigned int offset, unsigned int size,
unsigned char buffer[], unsigned char tmpStore[]);
```

Writes an arbitrary number of bytes to the store. Returns 0 on success.

### 14.2.5 Example field upgrade program

The program below shows how to use the libflash API with standard IO operations to retrieve an upgrade boot image. Usually this image would come from a network connection or SD card.

Required ports are declared in an XC file:

```
#include <platform.h>
#include <flashlib.h>

/* Declaration of port structure for fl_connect. */
fl_PortHolderStruct spiPorts =
{
    PORT_SPI_MISO,
    PORT_SPI_SS,
    PORT_SPI_CLK,
    PORT_SPI_MOSI,
};
```

The file I/O is written in a separate C program:

```
#include <stdio.h>
#include <stdlib.h>
#include <flashlib.h>

extern fl_PortHolderStruct spiPorts;

/* This speeds up file IO over gdb */
#define FILE_BUF_SIZE 16384
char file_buf[FILE_BUF_SIZE];

/*
 * User supplied function which is called-back by flashlib
 */
unsigned int supplyData(void* userPtr, unsigned int numBytes,
    unsigned char* dstBuf)
{
    return( fread(dstBuf, 1, numBytes, (FILE*)userPtr) );
}

int main()
{
    FILE* inFile = NULL;
    int imageSize;

    /* Open the input file and use a big read buffer */
    inFile = fopen("upgrade.bin", "rb");
    if( inFile == NULL )
    {
        fprintf(stderr, "Error: Cannot open input data file.\n");
        exit(1);
    }
    setvbuf(inFile, file_buf, _IOFBF, FILE_BUF_SIZE);
```

```
/* Discover the size of the image */
if( 0 != fseek( inFile, 0, SEEK_END ) )
{
    fprintf(stderr,"Error: Cannot discover input data file size.\n");
    exit(1);
}
imageSize = (int)ftell( inFile );
if( 0 != fseek( inFile, 0, SEEK_SET ) )
{
    fprintf(stderr,"Error: Cannot input file pointer.\n");
    exit(1);
}
printf("Image size is %d\n",imageSize);

/* Connect to the flash chip */
if( 0 != fl_connect(&spiPorts) )
{
    fprintf(stderr,"Error: Cannot recognise attached flash device.\n");
    fclose(inFile);
    exit(1);
}

/* Locate the factory boot image */
fl_BootImageInfo bii;
if( 0 != fl_getFirstBootImage( &bii ) )
{
    fprintf(stderr,"Error: Cannot locate factory boot image.\n");
    fclose(inFile);
    fl_disconnect();
    exit(1);
}

/*
 * Add a boot image after the factory one
 * using data supplied by supplyData() function
 */
if( 0 != fl_addBootImage( &bii, imageSize, 0x1, &supplyData,
    (void*)inFile ) )
{
    fprintf(stderr,"Error: Cannot add new boot image.\n");
    fclose(inFile);
    fl_disconnect();
    exit(1);
}
printf("Boot image added.\n");

/*
 * Reset the file pointer to the start of the file
 */
if( 0 != fseek( inFile, 0, SEEK_SET ) )
{
```

```
        fprintf(stderr,"Error: Cannot input file pointer.\n");
        exit(1);
    }

    /* Locate our newly added boot image. */
    if( 0 != fl_getFirstBootImage( &bii ) )
    {
        fprintf(stderr,"Error: Cannot locate factory boot image.\n");
        fclose(inFile);
        fl_disconnect();
        exit(1);
    }
    if( 0 != fl_getNextBootImage( &bii ) )
    {
        fprintf(stderr,"Error: Cannot find added boot image.\n");
        fclose(inFile);
        fl_disconnect();
        exit(1);
    }
}

/*
 * Read back new boot image and check itís the same as the file
 */
unsigned int upgradeAddr = bii.startAddress;
unsigned char checkBuf[256];
unsigned char fileBuf[256];
unsigned int checkPos = 0;
int gotError = 0;
while( checkPos < imageSize )
{
    int thisSize =
        ((imageSize-checkPos)>256) ? 256 : (imageSize-checkPos);
    fl_readPage( checkPos+upgradeAddr, checkBuf );
    fread( fileBuf, 1, 256, inFile );
    int i;
    for( i=0; i<thisSize; i++ )
    {
        if( checkBuf[i] != fileBuf[i] )
        {
            printf("Error: verification mismatch at 0x%08x (file:0x%02x,
                flash:0x%02x).\n", checkPos+i, fileBuf[i],checkBuf[i]);
            gotError = 1;
        }
    }
    checkPos += 256;
    if( gotError )
    {
        exit(1);
    }
}

/* Close down and exit. */
```

```
fclose( inFile );  
inFile = NULL;  
fl_disconnect();  
printf("Verification complete.\n");  
return( 0 );  
}
```