Blackbook5 API

TinyOS Cross-Platform Compatible
Flash File System

David Moss
Rincon Research Corporation
dmm@rincon.com

Introduction

The Blackbook File System was developed to meet several needs:

1. Develop a TinyOS flash file system that can be compatible with various microcontrollers and flash types.
    a. Target release platforms include Crossbow's Mica- type motes with an AT45DB flash, and Moteiv's Tmotes with an ST M25P80 flash.
    b. Blackbook should remain compatible and easily portable to other flash types as well, including NAND flash and other types of NOR flash with varying erase sizes and characteristics.
    c. The existence of internal flash in the microcontroller itself should not be assumed; Blackbook should exist entirely in the non-volatile memory provided by the external flash chip on the mote.
    d. The behavior and interfaces to the Blackbook file system should remain constant across all platforms, regardless of the underlying hardware.
    e. Explicitly support large flash erase sizes.

2. Provide maximum data throughput, natural wear leveling, minimum metadata flash usage, minimum RAM footprint, minimal energy consumption, and full fault-tolerance with catastrophic recovery.

3. Provide all the basic binary file functionality including read, write, delete, and dir. Random read access is supported.

4. Provide access to the underlying garbage collector to allow an application to clean the flash during downtime.

5. Provide functionality to save and manage settings and information to non-volatile memory through Dictionary files, similar to a database.

6. Allow application-managed continuous data logging.

Interfaces

Blackbook Interfaces

The main configuration file, BlackbookC, provides the following external interfaces to Blackbook:

- BBoot
  - Provides information about the Boot status of Blackbook. The file system cannot be accessed until Blackbook has fully booted.
- BClean
  - Provides access to Blackbook's garbage collector functionality. This alerts applications as to when a sector is going to be erased, and when the garbage collector is done running. It also allows an application to suggest to Blackbook when to perform a garbage collection, so the process can complete during non-critical times.
- BDictionary
  - Provides access to the Dictionary functionality of Blackbook. This allows applications to open up Dictionary type files and insert, retrieve, and remove key-value pairs of information. This interface is parameterized to allow multiple Dictionary files to be opened at once.
- BFileDelete
  - Allows an application to delete a file from the Blackbook file system.
- BFileDir
  - Allows an application to find general information about the file system and the files contained within.
- BFileRead
  - Open and read data from a file. This allows random read access. The interface is parameterized to allow an application to have multiple files open for reading at once.
- BFileWrite
  - Open and write data into a file through appends. This interface is parameterized to allow an application to have multiple files open for writing.

BBoot Interface

Applications should not attempt to interact with the Blackbook file system until it has finished booting. The BBoot interface alerts application components when Blackbook is ready to use.

Commands

`command bool isBooted();`

```
Use this to determine if Blackbook has been booted.

@return TRUE if the file system has been booted
```

Events

`event void booted(uint16_t totalNodes, uint8_t totalFiles, result_t result);`

```
Signaled when the file system is finished booting.  After
this event occurs, it is safe to use the Blackbook file
system.

@param totalNodes
      The total number of nodes found on flash.

@param totalFiles
      The total number of files found on flash.

@param result
      SUCCESS if the file system is ready for use.
      FAIL if something went wrong and you shouldn’t use
      the file system because you might corrupt something.
      Most likely this error would be the result of more
      nodes and files existing on flash than are currently
      allocated by the installed Blackbook settings from
      BlackbookConst.h.
```

BClean Interface

During downtime, applications can periodically access the BClean interface to request or force garbage collection. Because a sector on flash can take 1 second or longer to erase, it's best not to have to erase sectors during time-critical operations. Sectors are erased automatically when not enough space exists to meet the minimum requested size of a new file, and when a sector contains all delete file information.

Commands

`command result_t performCheckup();`

```
If the free space on the file system is over a 75%
threshold, then we should go ahead and garbage collect.
This should be run periodically when the mote has some time
and energy to spare.

@return SUCCESS if the file system will perform a garbage
collection
```

`command result_t gc();`

```
Force the garbage collection, erasing any sectors that
contain any data with 0 valid nodes.

@return SUCCESS if the garbage collector is run
```

Events

`event void erasing();`

```
The garbage collector is erasing a sector.  Erasing may
take awhile, so this serves as a warning.
```

`event void gcDone(result_t result);`

```
Garbage Collection is complete

@return SUCCESS if any sectors were erased.
```

BDictionary Interface

The Dictionary functionality was first created to support fault-tolerance recovery through a Checkpoint mechanism, but quickly extended to provide the ability to save and load general application settings through the use of key-value pairs. Different components can manage their settings and keys in separate files. Of course, keys stored in separate files can overlap without conflicts. The Dictionary can even provide binary file support, which may be used in future versions of Blackbook. This topic is discussed elsewhere.

Commands

```
command result_t open(char *fileName, uint32_t minimumSize);

      Open a Dictionary file. If the file does not exist on
      flash, the minimumSize will be used to set the length of
      the file.

      @param *fileName
            Name of the Dictionary file to open

      @param minimumSize
            The minimum reserved size for the file on flash.

      @return SUCCESS if the file will be opened


command result_t close();

      Close any opened Dictionary files

      @return SUCCESS because after this command, the Dictionary
      file will definitely be closed.


command result_t insert(uint32_t key, void *value, uint16_t
valueSize);

      Insert a key-value pair into the opened Dictionary file.
      This will invalidate any old key-value pairs using the
      associated key.

      @param key
            The key to use. Any value previously associated with
            this key will be lost.

      @param *value
            Pointer to a buffer containing the value to insert.

      @param valueSize
            The amount of bytes to copy from the buffer.

      @return SUCCESS if the key-value pair will be inserted
```

command result_t retrieve(uint32_t key, void *valueHolder, uint16_t maxValueSize);

Retrieve a key from the opened Dictionary file.

@param key
The key to retrieve

@param *valueHolder
Pointer to the memory location to store the retrieved value.If this is a struct, for example, after retrieval the struct will have been loaded with data.

@param maxValueSize
Used to prevent buffer overflows incase the recorded size of the value does not match the space allocated to the valueHolder

@return SUCCESS if the key will be retrieved.

command result_t remove(uint32_t key);

Remove a key from the opened dictionary file

@param key
The key to remove

@return SUCCESS if the attempt to remove the key will proceed

command result_t getFirstKey();

This command will signal event nextKey when the first key is found.

@return SUCCESS if the command will be processed.

command uint32_t getLastKey();

Obtain the last key written to the Dictionary file.

@return the last key, or (-1) if it doesn’t exist.

command result_t getNextKey(uint32_t presentKey);

Get the next recorded key in the file based on the current key. You can traverse through all recorded keys in the file by obtaining the first key, and then calling getNextKey(..) from the nextKey(..) event until you get a failure.

@return SUCCESS if the command will be processed

command result_t isFileDictionary(char *fileName);

Find out if a given file is a valid dictionary file.

@return SUCCESS if the command will be processed

Events

event void opened(uint16_t totalSize, uint16_t remainingBytes, result_t result);

A Dictionary file was opened successfully.

@param totalSize
The total amount of flash space dedicated to storing key-value pairs in the file

@param remainingBytes
The remaining amount of space left to write to

@param result
SUCCESS if the file was successfully opened.

event void closed(result_t result);

The opened Dictionary file is now closed

@param result
SUCCESS if there are no open files

event void inserted(uint32_t key, void *value, uint16_t valueSize, result_t result);

A key-value pair was inserted into the currently opened Dictionary file.

@param key
The key used to insert the value

@param *value
Pointer to the buffer containing the value.

@param valueSize
The amount of bytes copied from the buffer into flash

@param result
SUCCESS if the key was written successfully.

event void retrieved(uint32_t key, void *valueHolder, uint16_t valueSize, result_t result);

```
A value was retrieved from the given key.

@param key
      The key used to find the value

@param *valueHolder
      Pointer to the buffer where the value was stored

@param valueSize
      The actual size of the value.

@param result
      SUCCESS if the value was pulled out and is
      uncorrupted
```

event void removed(uint32_t key, result_t result);

```
A key-value pair was removed

@param key
      The key that should no longer exist

@param result
      SUCCESS if the key was really removed
```

event void nextKey(uint32_t nextKey, result_t result);

```
The next key in the open Dictionary file after a call to
getFirstKey() or getNextKey(..).

@param nextKey
      The next key in the Dictionary file.

@param result
      SUCCESS if this is the really the next key, FAIL if
      the presentKey was invalid or there is no next key.
```

event void fileIsDictionary(bool isDictionary, result_t result);

```
Event signaled after a call to isFileDictionary(..).

@param isDictionary
      TRUE if the file is a valid dictionary file.

@param result
      SUCCESS if this is the really the next key, FAIL if
      the presentKey was invalid or there is no next key.
```

BFileDelete Interface

Files deleted from Blackbook are not erased from flash until the garbage collector, BClean, erases the sector.Deleting a file will allow the sector to be erased, though, and will also free up space in RAM for other files to exist.

Commands

`command result_t delete(char *fileName);`

```
Delete a file.

@param *fileName
      The name of the file to delete

@return SUCCESS if Blackbook will attempt to delete the
file.
```

Events

`event void deleted(result_t result);`

```
A file was deleted

@param result
      SUCCESS if the file was deleted from flash
```

BFileDir Interface

The BFileDir interface provides general information about the file system.

Commands

`command uint8_t getTotalFiles();`

```
@return the total number of files in the file system
```

`command uint16_t getTotalNodes();`

```
@return the total number of nodes in the file system
```

`command uint32_t getFreeSpace();`

```
@return the approximate free space on the flash
```

`command result_t checkExists(char *fileName);`

```
@param *fileName
      Check for the existence of this file.

@return TRUE if the file exists, FALSE if it doesn't
```

`command result_t readFirst();`

```
An optional way to read the first filename of the system.
This is the same as calling BFileDir.readNext(NULL)

@return SUCCESS if readNext will be signaled.
```

`command result_t readNext(char *presentFilename);`

```
Read the next file in the file system, based on the current
filename. If you want to find the first file in the file
system, pass in NULL.

If the next file exists, it will be returned in the
nextFile event with result SUCCESS

If there is no next file, the nextFile event will signal
with the filename passed in and FAIL.

If the present filename passed in doesn't exist, then this
command returns FAIL and no signal is given.

@param *presentFilename
      The name of the current file, of which you want to
      find the next valid file after.
```

command uint32_t getReservedLength(char *fileName);

Get the total reserved bytes of an existing file. The reserved length is how much space on flash is reserved for data in the file, not how much data is actually written to the file.

@param *fileName
    The name of the file to pull the reservedLength from.

@return the reservedLength of the file, 0 if it doesn't exist

command uint32_t getDataLength(char *fileName);

Get the total amount of data written to the file with the given fileName.

@param *fileName
    Name of the file to pull the dataLength from.

@return the dataLength of the file, 0 if it doesn't exist

command result_t checkCorruption(char *fileName);

Find if a file is corrupt. This will read each node from the file and verify it against its dataCrc. If the calculated data CRC from a node does not match the node's recorded CRC, the file is corrupt. This process can take a long time and use quite a bit of energy for large files.

@param *fileName
    Name of the file to check for corruption.

@return SUCCESS if the corrupt check will proceed.

Events

event void corruptionCheckDone(char *fileName, bool isCorrupt, result_t result);

The corruption check on a file is complete

@param *fileName
    The name of the file that was checked

@param isCorrupt
    TRUE if the file's actual data does not match its CRC

@param result
    SUCCESS if this information is valid.

event void existsCheckDone(char *fileName, bool doesExist, result_t result);

```
The check to see if a file exists is complete

@param *fileName
      The name of the file

@param doesExist
      TRUE if the file exists

@param result
      SUCCESS if this information is valid
```

event void nextFile(char *fileName, result_t result);

```
This is the next file in the file system after the given
present file.  This is signaled after a readFirst() or
readNext(..) command.

@param *fileName
      Name of the next file

@param result
      SUCCESS if this is actually the next file, FAIL if
      the given present file is not valid or there is no
      next file.
```

BFileRead Interface

The BFileRead interface allows an application to read back data that is written to a file.Random read access is also supported.

Commands

```
command result_t open(char *fileName);

      Open a file for reading

      @param *fileName
            Name of the file to open

      @return SUCCESS if the attempt to open for reading proceeds



command result_t close();

      Close any currently opened file

      @return SUCCESS because the file for the current
      parameterized interface will always be closed after this
      command is called.


command result_t read(void *dataBuffer, uint16_t amount);

      Read a specified amount of data from the open file into the
      given buffer.

      @param *dataBuffer
            The buffer to read data into

      @param amount
            The amount of data to read

      @return SUCCESS if the command goes through
```

command result_t seek(uint32_t fileAddress);

```
Seek a given address to read from in the file.

This will point the current internal read pointer to the
given address if the address is within bounds of the
file.When BFileRead.read(...) is called, the first byte of
the buffer will be the byte at the file  address specified
here.

If the address is outside the bounds of the data in the
file, the internal read pointer address will not change.

@param fileAddress
      The address to seek in the file.

@return SUCCESS if the read pointer is adjusted, FAIL if
the read pointer didn't change
```

command result_t skip(uint16_t skipLength);

```
Skip the specified number of bytes in the file

@param skipLength
      Number of bytes to skip

@return SUCCESS if the internal read pointer was adjusted,
FAIL if it wasn't because the skip length is beyond the
bounds of the file.
```

command uint32_t getRemaining();

```
Get the remaining bytes available to read from this file.
This is the total size of the file minus your current
position.

@return the number of remaining bytes in this file
```

Events

event void opened(char *fileName, uint32_t amount, result_t result);

```
      A file has been opened

      @param *fileName
            Name of the opened file

      @param amount
            The total amount of data to read in the file

      @param result - SUCCESS if the file was successfully opened
```

event void closed(result_t result);

```
      Any previously opened file is now closed

      @param result
            SUCCESS if the file was closed properly
```

event void readDone(char *fileName, void *dataBuffer, uint16_t amount, result_t result);

```
      File read complete

      @param *fileName
            The name of the file being read

      @param *dataBuffer
            A pointer to the data buffer containing the data that
            was read

      @param amount
            The length of the data read into the buffer

      @param result
            SUCCESS if there were no problems reading the data
```

BFileWrite Interface

The BFileWrite interface allows an application to append data to a binary file.

When no more files can be opened for writing, that means your flash is filled up, your RAM can no longer support any more files, the file being opened for writing is already in use, or the parameterized interface opening the file already has a different file open.

Commands

`command result_t open(char *fileName, uint32_t minimumSize);`

```
Open a file for writing.

The minimumSize must be specified to ensure enough memory
exists in flash for the operation.  For example, if you
want to create a new file and write 0x1000 bytes to it,
then make the minimumSize 0x1000.  The total data length of
the file will always meet the minimum size requirements, or
open will fail.  This means that if a file already exists
with 0x1000 bytes, and you want to re-open it to append
0x1000 more bytes, the minimum size of the file will be
0x2000.  You can open it for appending 0x1000 more bytes by
taking advantage of the BFileDir interface:

      call BFileWrite.open("myFile", call
            BFileDir.getDataLength("myFile") + 0x1000);

@param *fileName
      Name of the file to open for writing

@param minimumSize
      The minimum requested amount of total space to
      reserve in the file.
```

`command result_t close();`

```
Close any currently opened write file.  The file is always
automatically saved before being closed.

@return SUCCESS because the file will always be closed
after this command is called.
```

command result_t save();

Save the current state of the file, guaranteeing the next time we experience a catastrophic failure, we will at least be able to recover data from the open write file up to the point where save was called.

If data is simply being logged for a long time, use save() periodically but probably more infrequently.

You do not need to call save() right before you call close(). In fact, doing so will waste flash space, time, and batteries.

@return SUCCESS if the currently open file will be saved.

command result_t append(void *data, uint16_t amount);

Append the specified amount of data from a given buffer to the open write file.

@param *data
    Pointer to the buffer of data to append

@param amount
    The amount of data in the buffer to write.

@return SUCCESS if the data will be written, FAIL if there is no open file to write to.

command uint32_t getRemaining();

Obtain the remaining bytes available to be written in this file. This is the total reserved length minus your current write position.

@return the remaining length of the file that is writable.

Events

event void opened(char *fileName, uint32_t len, result_t result);

Signaled when a file has been opened

@param *fileName
The name of the opened write file

@param len
The total reserved length of the file

@param result
SUCCESS if the file was opened successfully

event void closed(result_t result);

Signaled after the close command.

@param result
SUCCESS if the file was closed properly, which should be always.

event void saved(char *fileName, result_t result);

Signaled when this file has been saved.

@param *fileName
Name of the open write file that was saved

@param result
SUCCESS if the file was saved successfully

event void appended(char *fileName, void data, uint16_t amountWritten, result_t result);

```
Signaled when data is written to flash. On some media, the
data is not guaranteed to be written to non-volatile memory
until save() or close() is called.

@param *fileName
      Name of the file that was appended to

@param data
      The buffer of data appended to flash

@param amountWritten
      The amount of data actually written to flash

@param result
      SUCCESS if there were no problems appending data
```

Contact

Bug reports, comments, and questions about Blackbook are welcome!
David Moss
dmm@rincon.com