Embedding it better...

µTasker Document

µFileSystem – Revision 1

Table of Contents

1. Introduction

This document details a revision made to the way that the uFileSystem (and low level accesses) work with storage medium that can be of varying device type and/or have multiple devices.

The background to the revision is that the storage types covered have grown (eg. internal Flash, external SPI Flash, external I2C EEPROM etc.) and in some cases there are multiple devices of the same type that need to be controlled. With increasing device types the code has grown to be less understandable and with multiple devices there have been some problems with writing and reading over device boundaries that needed to be solved for each individual case.

The revision attempts to improve the code to be more structured in order to handle the various situations in a more generic manner and also to add new devices without its complexity each time increasing again.

The revised operation is enabled by the define `STORAGE_REV_1`. Although it is advised to work with the new strategy the user can disable it by disabling this define in `config.h` in case of any problems encountered. *Once the revised operation has been fully verified the method will be used exclusively*.

2. Examples of what should be possible

1) When reading a block of data to a buffer the starting memory location can be in any storage medium. The read length can extend over multiple devices, requiring the read to be correctly managed over multiple devices of the same or differing types.

2) When writing a block of data from a buffer the starting memory location can be in any storage medium. The write length can extend over multiple devices, requiring the write to be correctly managed over multiple devices of the same or differing types.

3) When erasing a part of memory the initial section can be in any storage medium. The erase length can extend over multiple devices, requiring the erase to be correctly managed over multiple devices of the same or differing types.

In each case the user doesn't necessarily need to know in what type of storage medium the destination area is in to be able to perform the operations.

A further advantage of the revised technique is that the user can add or remove memory areas from the system at run-time, allowing the memory dimensions to be adjusted depending on what devices are physically available.

3. Demonstration Project

The uTasker demonstration project includes (among others, and depending on the project status) support of internal flash, external SPI based flash, external I²C based EEPROM and external parallel flash. The uFileSystem is linear, beginning in internal flash (if used) and extending into other media types.

The project demonstrates SPI Flash use (a single device or multiple devices of the same type) as an extension to the file system or as a replacement to the use of internal Flash. This is enabled by the define `SPI_FILE_SYSTEM` and the corresponding dimensioning of the µFileSystem and µParameterSystem. The demonstration project assumes that the declared number of SPI Flash devices are always present and the file granularity used by internal and external Flash should be the same.

The project demonstrates I²C EEPROM use (a single device or multiple devices of the same type) as an extension to the file system or as a replacement to the use of internal Flash. This is enabled by the define `I2C_EEPROM_FILE_SYSTEM` and the corresponding dimensioning of the µFileSystem and µParameterSystem. The demonstration project allows the number of devices attached to be determined at run time and so the size of the file system changes accordingly. The file granularity used by internal Flash and external EEPROM should be the same.

The project demonstrates external parallel Flash use (a single device or multiple devices of the same type) as an extension to the file system or as a replacement to the use of internal Flash. This is enabled by the define `EXT_FLASH_FILE_SYSTEM` and the corresponding dimensioning of the µFileSystem and µParameterSystem. The demonstration project assumes that the declared number of external parallel Flash devices are always present and the file granularity used by internal and external Flash should be the same.

Although the demonstration project doesn't show the use of a mixture of the storage media the revision 1 project adaptation would allow this to be configured with little extra effort.

4. Memory Mapping

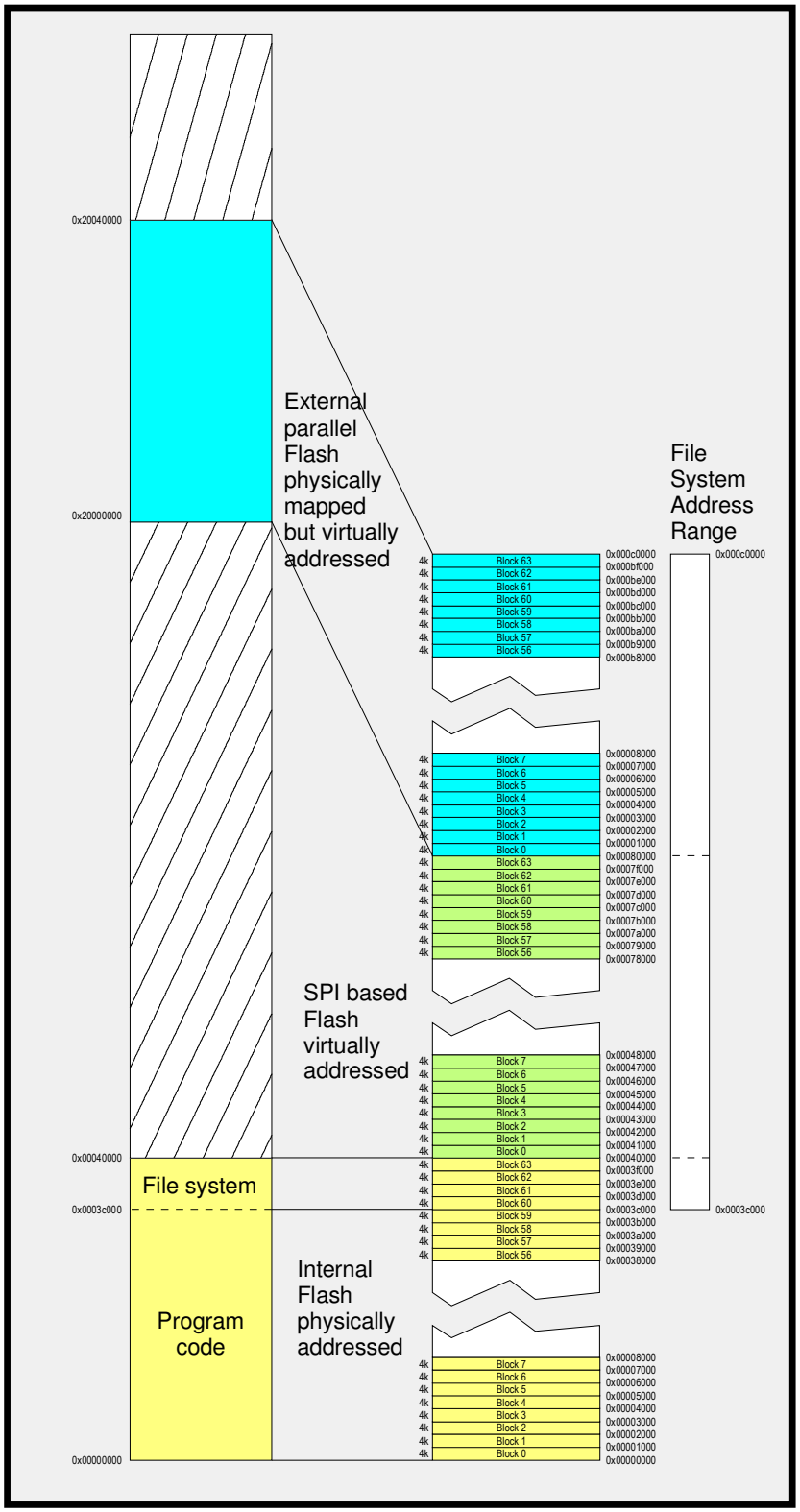

Figure 1 – File System Memory Mapping

Figure 1 shows how the file system memory mapping operates. It is based on the fact that the user sees the file system as a contiguous block of memory in the memory space. All memory access using the file system methods also accesses the data according to this strategy.

The example shows three types of memory being used: internal Flash, external SPI Flash and external parallel Flash. In each case the three memory types have the same physical size (for simplicity) but each show different characteristics:

- The internal Flash is directly memory mapped. The file system uses a part of the available area, starting at the physical address location of the part of the Flash that is used by the file system.
- The external SPI Flash is mapped virtually into the otherwise free physical memory space following the internal Flash. Since the SPI Flash (or multiple devices) doesn't actually have a memory mapped location the interface performs the mapping function.
- The external parallel Flash memory does have a physical location in the system's memory map but this is not suitable (in this case) for use as a member of the contiguous file system space and so it is virtually addressed (mapped between its physical location and its file system location).

5. Modifications

There are primarily three routines that are affected by the revision:

```
extern void fnGetParsFile(unsigned char *ParLocation, unsigned char
*ptrValue, MAX_FILE_LENGTH Size);

extern int fnWriteBytesFlash(unsigned char *ucDestination, unsigned
char *ucData, MAX_FILE_LENGTH Length);

extern int fnEraseFlashSector(unsigned char *ptrSector,
MAX_FILE_LENGTH Length):
```

These routines are a part of the hardware file belonging to each processor. The use of the routines by user code and/or uFileSystem code don't need any modification itself.

Originally the routines checked for the area of memory that each access is for and used the appropriate hardware details to read, write or delete the content. With an increasing number of storage media this code had become difficult to maintain. These routines now serve more as cover functions to the hardware related details by first calling a new local function called `fnGetStorageType()` which performs the decision as to which of the storage areas the access starts in. In addition it determines in which device in the storage area the access is to when multiple devices exist (eg. there may be 4 SPI Flash devices used to create an SPI Flash storage area). Finally it limits the length of the access to within the range of a single device. If the function (read, write or erase) is to be performed in multiple storage types or in multiple devices the cover functions include a loop which handles this in a generic manner.

The result is that it is simple to add different hardware level handling to the driver without needing to fundamentally modify the cover function's details and the new hardware details can then be cleanly added in their own sub-routines, aiding readability and maintenance.

A new detail is that the `fnGetStorageType()` operation is not based on fixed storage media locations but on a list of storage location tables, where each storage type has its own entry. The details list initially contains typically only internal Flash and the list can be adapted dynamically (when needed) to enable run-time decisions to take place and the storage space to be suitably adapted.

The internal flash is initially the only table entry:

```
STORAGE_AREA_ENTRY *UserStorageListPtr = (STORAGE_AREA_ENTRY *)&internal_flash;

static const STORAGE_DAREA_ENTRY default_flash = {
    0,                                      // end of list
    (unsigned char *)(FLASH_START_ADDRESS) // start address of internal flash
    (unsigned char *)(FLASH_START_ADDRESS + SIZE_OF_FLASH),  // end of internal
                                                               flash
    _STORAGE_INTERNAL_FLASH,                // type
    0                                       // not multiple devices
    };
```

Code 1

Drivers automatically insert their own table entry into the list by changing `UserStorageListPtr` to point to their entry and then setting their next entry pointer to the original flash entry.

```
typedef struct stSTORAGE_AREA_ENTRY{
void          *ptrNext;          // pointer to the next storage type available, or
                                    zero at end of list
unsigned char *ptrMemoryStart;   // pointer to the first address in the storage
                                    area
unsigned char *ptrMemoryEnd;     // pointer to last location at the end of the
                                    storage area (total area when multiple devices)
unsigned char ucStorageType;     // the storage type, such as internal Flash, SPI
                                    Flash etc.
unsigned char ucDeviceCount;     // the number of individual devices the storage
                                    area is made up of (it is assumed that all
                                    devices are the same type and size)
} STORAGE_DEVICE_ENTRY;
```

Code 2

In the case of storage areas made up of multiple devices it is to be noted that the `ptrMemoryStart` and `ptrMemoryEnd` pointers refer to the complete storage area range. The `ucDeviceCount` is set to the number of devices making up the area. This means that only one storage area entry is needed for all these devices.

It is to be noted that the reason why the address of last byte is used, and not following address, is to allow the storage area to end at the last possible address in 32 bit memory space; otherwise there would be an overflow to zero.

6. Example

The following is an example of configuring external parallel flash memory to operate as extension to internal flash.

The internal flash is 512k bytes in size and starts at the address `0x00000000`. The final 128k of the space is to be used by the uFileSystem.

The external parallel flash area is 2Meg bytes in size and is memory mapped to the physical address `0xffe00000`. It is made up of two chips of each 1 Meg bytes in size.

The configuration of the two storage areas may be fully automated in the processor file but it could also be added in new cases by users, whereby only access to the internal flash's default configuration block is required as follows:

```
STORAGE_AREA_ENTRY *UserStorageListPtr = (STORAGE_AREA_ENTRY *)&default_flash;
                                                          // default entry
```

Code 3

`UserStorageListPtr` is available globally and will be initialised as show in code 1.

The external parallel flash will usually be connected to an external memory interface allowing fast access and control of its location in memory (chip select control). Since there are two physical chips involved in the example the interface will be configured so that the first chip

select will be asserted when accesses are made to the range `0xffe00000..0xffefffff` and the second when the access in the range `0xfff00000..0xffffffff`.

The storage area for this medium is then inserted into the storage media list as follows:

```
static const STORAGE_AREA_ENTRY external_flash = {
    (void *)&default_flash,             // inserted before the internal flash
    (unsigned char *)(EXTERNAL_FLASH_START_ADDRESS), // start address of external
                                                        flash
    (unsigned char *)(EXTERNAL_FLASH_START_ADDRESS + (SIZE_OF_EXTERNAL_FLASH - 1)),
                                                      // end of external flash
    _STORAGE_PARALLEL_FLASH,                          // type
    EXTERNAL_FLASH_DEVICE_COUNT                       // number of devices
};
```

Code 4

The dimensions are set by

```
#define EXTERNAL_FLASH_START_ADDRESS (0xffe00000)
#define SIZE_OF_EXTERNAL_FLASH_CHIP   (1024 * 1024)
#define EXTERNAL_FLASH_DEVICE_COUNT   2
#define SIZE_OF_EXTERNAL_FLASH
                          (EXTERNAL_FLASH_DEVICE_COUNT*SIZE_OF_EXTERNAL_FLASH_CHIP)
```

Code 5

To complete the insertion of the parallel flash into the storage media list the following operation is performed:

```
    UserStorageListPtr = (STORAGE_AREA_ENTRY *)&external_flash;
```

Code 6

This modifies the list starting with the external parallel flash area, followed by the original internal flash area.

In order to map the external flash's physical location to be contiguous with the end of the internal flash the following define is used:

```
#define EXT_PARALLEL_FLASH_OFFSET
          ((FLASH_START_ADDRESS + SIZE_OF_FLASH) - EXTERNAL_FLASH_START_ADDRESS)
```

Code 7

This define is used by the external flash driver to allow it to shift its virtual location to suit the file system.

The file system area now looks like a block of internal flash from 0x00000000..0x7ffff, followed immediately by external flash between 0x80000..280000. These addresses are used when reading, writing or erasing memory using the routines described at the start of this document.

It is assumed that the file granularity can be chosen to be suitable for the flash granularity of internal and external flash. The file system itself is declared to start at 0x60000 and be 0x220000 bytes in size; using a large 64k file granularity it looks as follows:

```
#define SINGLE_FILE_SIZE         (64 * 1024)   // files are made up of 64k blocks
#define uFILE_START              (0x60000)     // first file starts here
#define FILE_SYSTEM_SIZE         (0x220000)    // 2176k reserved for file system
```

Code 8

This doesn't include any area for the uParameterSystem but this can also be included either in internal or external flash memory if required.

The flash drivers for the various storage areas, combined with the control functions used by the interface routines, allow reads, writes or deletes to be made from any areas in the storage area space. The user doesn't need to know what memory is being used and operations spanning multiple storage media types and multiple individual chips are also handled transparently.

7. Conclusion

This document has detailed the reasons for the implementation of a revision for the µFileSystem access methods which ensures that the operation of various storage media, optionally based on multiple devices, is handled in a generic manner to ensure reliable operation as well as improved project maintenance.

See also the µFileSystem and µParameterSystem document for details of their operation and features: http://www.utasker.com/docs/uTasker/uTaskerFileSystem_3.PDF

Modifications:
V0.00 07.10.2011: Initial draft