Embedding it better...

µTasker Document

µFileSystem and µParameterSystem

Table of Contents

1. Introduction

µTasker is an operating system designed especially for embedded applications where a tight control over resources is desired along with a high level of user comfort to produce efficient and highly deterministic code.

This document discusses the implementation and use of the µTasker file system – known as **µFileSystem**. *This document is valid for µTasker releases from V1.2.008 but is also suitable as an introduction for older releases. Note that the **µParameterSystem** was decoupled from the **µFileSystem** during the further development of the V1.3 project and so this must also be considered when working with older versions, where the **µFileSystem** size counted also any **µParameterSystem** memory.*

Some devices can be classified as containing FLASH with large granularity (larger than the size of typical files within a file system, with the desire to be able to save multiple files in one FLASH block). For details about the **µFileSystem** extension for large-FLASH granularity types please consult the document:
http://www.utasker.com/docs/STR91XF/FileSystemSTR91X.PDF

For details of extended support for SPI FLASH see the document:

http://www.utasker.com/docs/uTasker/uTaskerFileSystemSPI_FLASH_003.PDF

This revision of the µFileSytem documentation (revision 5 and above) includes additional, optional features allowing a larger number of individual files to be used (`EXTENDED_UFILESYSTEM`) as well as an optional capability to append data to existing files (`UFILESYSTEM_APPEND`). These additional features are discussed in new chapters, whereby the rest of the chapters are generally left to represent the case without these capabilities.

From revision 8 the extended file names has changed from “ZXXX” to ‘zXXX’ [`uFile.c` from 04.10.2014]. This change was to enable better use of maximum µFileSystem size before the extended area starts.

2. First Things First

The **µFileSystem** is rather special. It was designed to enable such services as FTP, HTTP and parameter support to be integrated into embedded systems with a minimum amount of resources but with a useable level of comfort and flexibility. It operates in internal FLASH - either in linear or paged systems – and in external SPI EEPROMs/ SPI FLASH. It can in addition be configured to work in small memory environments up to 64k (file system size and individual file sizes) or larger memory environments (file system and/or individual files of larger than 64k).

The **µFileSystem** was not designed to be a copy of other file systems, so don't be disappointed when you find it is not FAT32 conformant. This is not a goal and also not necessarily a benefit for file operation in a small embedded system – the µTasker project also contains a FAT32 file system (utFAT) which can be located in SD cards or other memories such as NAND Flash – see its user's guide for more details:
http://www.utasker.com/docs/uTasker/uTasker_utFAT.PDF

You will however find that the µFileSystem is very easy to use, reliable and all that is often needed for the more important higher level tasks and protocols in real-world projects based on the supported devices.

3. Configuring µFileSystem

There are several parameters which need to be configured so that the **µFileSystem** operates with size constraints suiting your project. Such things are:

- The structure of the FLASH within the device (its granularity) – a SPI EEPROM doesn't have the same restrictions but uses an emulation technique where this parameter is also specified.
- The block size of the files which you want to use in your project. (*This cannot be smaller than the FLASH granularity itself*).
- The number of files which the file system should support (equivalent to its total size).
- The size of parameters which should also be saved within the µParameterSystem (*if used*) and whether there should be a swap block.
- The maximum single file size within the file system (basically whether smaller than 64k or not).
- Whether the total file system space is larger than 64k or not.

Some parameters are limited by the physical properties of the devices used and so are predefined and not real user parameters in this sense.

Before going in to details about each of these parameters, let's start with a graphical representation of a typical **µFileSystem** in the internal FLASH of a small device:

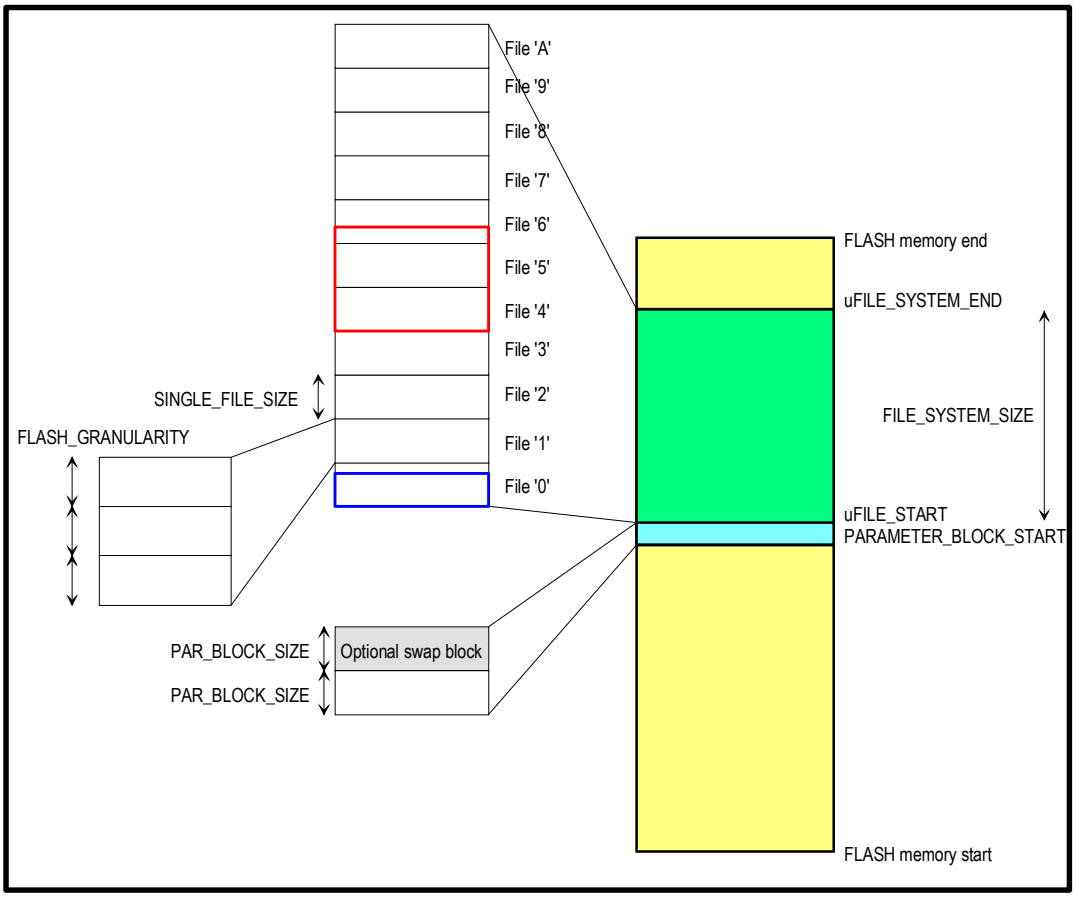

Figure 1.Typical **µFileSystem** block positioned in the on-chip FLASH of a device

Here is an example configuration of the **µFileSystem** in the hardware configuration file (app_hw_xxxx.h, where xxxx is the target processor used):

```
#define uFILE_START 0x18000                        // FLASH location at 96k start
#define SINGLE_FILE_SIZE (4*FLASH_GRANULARITY)     // each file a multiple of 1k
#define FILE_SYSTEM_SIZE (32*SINGLE_FILE_SIZE)     // 32k reserved for file system
                                                      (including parameter blocks)
```

This example assumes a FLASH_GRANULARITY of 256 bytes.

It can be seen that the start of the block can be set to a fixed address, which must obviously be in the real FLASH address range. The FLASH granularity is defined by the device used (here 256 byte granularity – or FLASH sector size [smallest block which can be individually erased] is assumed and the SINGLE_FILE_SIZE is set 4 times larger to achieve 1k file blocks. *The FLASH granularity is usually a constant defined by the chip used – see the header file belonging to the hardware device, eg. \Hardware\XXXX\xxxx.h, where XXX is the target used.*

The complete size of the FLASH memory used by the file system is defined as 32k. This will thus result in space for 32 files of 1k size, or correspondingly less when using multiples of 1k.

There are a few simple rules involved, as follows:

- File locations are specified by names, which are essentially one number or letter (or the first number or letter in a longer name). The file locations count from '0' to '9', and then from 'A' to 'Z', and then from 'a' to 'z'. This gives a maximum of 62 individual files in all. The actual amount useable is given by the total size defined and the SINGLE_FILE_SIZE used.
- A file smaller that SINGLE_FILE_SIZE occupies still SINGLE_FILE_SIZE space in the FLASH, where SINGLE_FILE_SIZE can be made up of more than one FLASH sector (FLASH_GRANULARITY).
- A file larger than SINGLE_FILE_SIZE occupies as many SINGLE_FILE_SIZE blocks as necessary to fit in to the file system. *These blocks are contiguous and thus, e.g. file 'A' may also occupy space for 'B' and 'C' etc.*
- The files system is flat, meaning that it doesn't support directories.
- When copying files to the file system (for example via FTP) only the first letter is observed, but certain extensions can be optionally respected. The following are examples of valid names: MyFile.txt; MyFile.TXT; MyFile.001.txt [This file will be copied to the 'M' space in the file system and it will be understood as a text file] {Note that myFile.txt uses a different case and so is positioned at another address – it is not the same file!}

From these first rules it is clear that the location of a file in the **µFileSystem** is directly defined by its name. It also becomes clear that the user must thus know which files are to be saved so that they actually fit (this can be controlled by the names given to the files to be saved). For example, it is not possible to write a file requiring more than one file block to the last block in the file system – *it will just not fit!*

The **µFileSystem** interface routines coordinate reading files and writing them and also handles the following cases appropriately, giving the next important rule:

- If a file is written to a file block which is already occupied, the new file takes preference. This means that the existing file will first be deleted from all blocks which it may occupy.

Now this may sound quite strange at first but as I said before, the **µFileSystem** is not designed to be a copy of other file systems, but designed to allow useful things to be performed in a very efficient manner. The application software designer is however involved a little since he/she must in fact also design the use of the available file system space.

The µTasker demo project is delivered with web pages which are loaded by FTP. Check out the names of the files to see where they are located. Check out the size of the files and you will see that these also dictate the names of the web pages, since they must fit without clobbering existing ones when loaded (see last rule above). But it does work and for such tasks this system is very useable.

See also the following folder in the µTasker demo project
`\Applications\uTaskerV1.x\WebPages\WebPages`*`HWType`*`\FileSystem`

It contains a word document used as a template for designing the positioning of files within the file system for the project and can be used as a template for further work. It should now also be easily understandable after this introduction.

Before moving to more details about how the files are built up and how to use the access routines, it is important to quickly mention two `typedefs` in the project. These are found in `types.h` [belonging to the application] and the two which interest us here are `MAX_FILE_LENGTH` and `MAX_FILE_SYSTEM_OFFSET`.

These control the use of variables (basically `unsigned short` and `unsigned long` types) which are used within the routines controlling the files. The value `unsigned short` restricts values from 0x0000 to 0xffff (64k) and so also limits the size of the file system and file offsets within. If you require a file system length of more than 64k it is necessary to ensure that the `typedef` for `MAX_FILE_SYSTEM_OFFSET` to be `unsigned long` so that this will work properly. This will allow a larger file system length and individual files to 64k in length. If also *individual* files are required to be longer than 64k, then also `MAX_FILE_LENGTH` should be specified to be an `unsigned long`.

Here's a table for clarity:

	`MAX_FILE_SYSTEM_OFFSET`	`MAX_FILE_LENGTH`
File system of less that 64k	`typedef unsigned` **`short`**	`typedef unsigned` **`short`**
File system larger than 64k but individual files smaller than 64k	`typedef unsigned` **`long`**	`typedef unsigned` **`short`**
File system larger than 64k **AND** individual files larger than 64k	`typedef unsigned` **`long`**	`typedef unsigned` **`long`**

Of course all projects could use `unsigned long typedefs` and no one would have to decide, but this is of course a waste of resources in systems where it is not necessary, making the code also less efficient than it could be as a 16 bit device may have to mess around with 32 bit words for no real reason at all. So best to be able to tune it…the demo project is set up to suit the particular target used and so may set the most common settings for that device.

4. File Contents

The individual files within the system are copied and read from fixed locations in memory which are referenced by file name. This means that there doesn't have to be any central file management keeping track of where they are (*for example a File Allocation Table*).

The files themselves do however have some characteristics otherwise it would not be possible to know whether they are valid and how long their contents actually are.

The **µFileSystem** doesn't try to do anything about file access rights or file date and times – if an application requires such support (which is rather unlikely in most small embedded systems) this will have to be handled somewhere else – like in the application.

The result of this simplification is that also the files themselves require very little information saved. The files are built up with a very short header made up of the file length (which will be either 2 bytes or 4 bytes in length depending on the setting of `MAX_FILE_LENGTH` (see above)). This also operates as a file validity check since blank FLASH will return a file length of 0xffff or 0xffffffff, which is interpreted as zero length. This also means that there is in fact an additional restriction to the real file content length since space for the header is required, but this is only of the order of 2 or 4 bytes here.

It is often also interesting to know what type of data has been saved in a file (*.txt, *.GIF, *.JPG etc.) and this may also be optionally saved by setting the define `SUPPORT_MIME_IDENTIFIER` in `config.h`

When using `SUPPORT_MIME_IDENTIFIER` the FTP server will, for example, display the file extensions and the HTTP server will better be able to work with images, CSS and Java etc. since it will know not to try to interpret the wrong types of data. Its use increases the file header by just one byte and is recommended!

When working with file extensions the user has the capability to control exactly which types are supported in the project. This can be set `in config.h` in the following table:

```
const CHAR *cMimeTable[] = {
    (const CHAR *)"HTM",    // HTML file - will be interpreted by web server
    (const CHAR *)"JPG",    // JPG image
    (const CHAR *)"GIF",    // GIF image
    (const CHAR *)"CSS",    // CSS Cascading Style Sheets
    (const CHAR *)"JS",     // Java script
    (const CHAR *)"BIN",    // Binary data file
    (const CHAR *)"TXT",    // Text data file
    (const CHAR *)"???",    // all other types will be displayed as unknown
};
```

The number of extensions can be reduced or extended by removing or adding them from/to the list. The extensions are not case sensitive so `test.htm` and `test.HTM` will be identified as being equivalent – in the FTP server they will however be displayed as `HTM`, as defined in the list.

The entry for HTM is very useful since it is used by the HTML parser in the web server to identify which types of files are to be parsed – it should therefore be present to enable this to operate correctly (unless `SUPPORT_MIME_IDENTIFIER` is not activated).

Note also that longer file names, such as `html` (a typical extension also used for HTML files) will also be interpreted as equal to `HTM`. Such a file will later be displayed by the FTP server as HTM – from the list.

5. Interface Routines

Since the basic workings of the **µFileSystem** should be quite clear by now, it is worth looking at how the application can make use of it through its interface routines. These interface routines are hardware independent and so are portable between processor types and storage locations (internal or external memory). The prototypes for the routines are found in `driver.h`.

Note that the FTP and HTTP servers make use of the various file interface routines and so are a good practical reference. The application can of course also make use of the calls if required.

```
extern MEMORY_RANGE_POINTER*uOpenFile(CHAR *ptrfileName);
```

This call passes a pointer to a file name and returns a pointer to its contents. The pointer is to the memory location of the file according to its name and doesn't indicate that the file is valid or not empty. To obtain the length of the file (which will be 0 if the file is not present) the routine `uGetFileLength()` [see below] can be used.
Example:

```
MEMORY_RANGE_POINTER ptrFile = uOpenFile("MyFile.txt");
```

`uOpenFile("M")` is also adequate to access the same named file. The pointer to the contents can be later used for content retrieval and modification purposes.

```
extern MAX_FILE_LENGTH uGetFileLength(unsigned char *ptrFile);
```

This call passes a pointer to a file name and returns the file content length.
Note that a file pointer to an area of previous file content will return a random vale (depending on file content) and so it is generally recommended to scan the µFileSystem content form the start to accurately obtain available files and their lengths.

Example:

```
MAX_FILE_LENGTH file_length = uGetFileLength(ptrFile);
```

```
extern MAX_FILE_LENGTH uGetFileData
                (unsigned char *ptrFile,
                MAX_FILE_SYSTEM_OFFSET FileOffset,
                unsigned char *ucData,
                MAX_FILE_LENGTH DataLength);
```

This call retrieves the specified amount of bytes content from the offset in the file referred to by its file pointer. It returns the number of bytes copied, which will be the same as the number specified (`DataLength`) if there are enough remaining or else the number of bytes which could be copied. Using this function it is not actually necessary to know the file length before accessing.

Example: (the complete contents of the file are retrieved in packets of 50 bytes each. After the retrieval `file_offset` also contains the file length)

```
MAX_FILE_LENGTH file_offset = 0;
MAX_FILE_LENGTH bytes_retrieved = 0;
unsigned char ucBuffer[50];
do {
```

```
        bytes_retrieved  = uGetFileData(ptrFile, file_offset,
                                        ucBuffer, 50);
        file_offset += bytes_retrieved;
    } while (bytes_retrieved < 50);
```

```
extern MAX_FILE_LENGTH uFileWrite(unsigned char *ptrFile,
                unsigned char *ptrData,
                MAX_FILE_LENGTH DataLength);
```

This routine writes the specified amount of data to the file. Note that only one user can be writing to a file at one time (single-user write). Each call to the write function appends the data to the previous write position (that is, in a linear fashion starting at the beginning of the file on the first write and progressing through the file system space on each subsequent write). The write is terminated after all data has been copied by using the close command as detailed below.

The routine returns the total length of data copied since the first call. Writes and the total length are restricted to within the reserved file system space to avoid writing outside of this space.

The routine automatically deletes any files which may already occupy the space required by the write, meaning that the user doesn't have to know whether the space is free or not beforehand – it will be "bulldozed" free if needed…

Example: This routine copies two blocks of data to the file, where they will be programmed to the memory (usually FLASH) resulting in the sequence 100 x 0x5a : 100 x 0xb3 – 200 bytes in total. Note that the file contents is not valid until the file has been closed (see next routine)

```
unsigned char ucData[100];
uMemset(ucData, 0x5a, 100);
uFileWrite(ptrFile, ucData, 100);
uMemset(ucData, 0xb3, 100);
uFileWrite(ptrFile, ucData, 100);
```

```
extern MAX_FILE_LENGTH uFileClose(unsigned char *ptrFile);
```

This routine is used to close a file whose contents have been fully written. It returns the length of the file which has been in the process of being written. Note that this is only used when extensions are not supported – when using `SUPPORT_MIME_IDENTIFIER` the following close routine must be used instead.

Example: The file written in the last routine example is closed. It will return the length 200 and is now valid for reading – its name and length are valid.

```
if (uFileClose(ptrFile) != 200) {
    // Error
}
```

```
extern MAX_FILE_LENGTH uFileCloseMime(unsigned char *ptrFile,
                unsigned char *ucMimeType);
```

This routine is used to close a file whose contents have been fully written. It returns the length of the file which has been in the process of being written. This version of close is used always when `SUPPORT_MIME_IDENTIFIER` is defined so that the file extension can be set.

Example: The file written in the last routine example is closed. It will return the length 200 and is now valid for reading – its name, length and extension type are valid.

```
unsigned char ucMimeType = MIME_BINARY;
if (uFileCloseMime(ptrFile, &ucMimeType) != 200) {
    // Error
}
```

```
extern int uFileErase(unsigned char *ptrFile,
                      MAX_FILE_LENGTH FileLength);
```

This routine is used to erase an area in FLASH. It returns a zero if the erase was successful or a negative value if not. The erase length is the length of the file to be erased, or can also be a value equivalent to a number of file granularity sizes (which is automatically rounded up to suit).

Example: this example shows how one file is deleted by passing its length to the function.

```
MAX_FILE_LENGTH Length = uGetFileLength(ptrFile);
uFileErase(ptrFile, Length);
```

```
extern CHAR uGetFileName(unsigned char *ptrFile)
```

This routine is passed the name (single digit) of the file. Note that this also returns the name of the file position pointed to even when the file is not present.

Example:

```
CHAR cFileName = uGetFileName(ptrFile)
```

```
extern unsigned char fnConvertName(CHAR cName);
```

This routine takes a file name (0..9, ‘A’..’Z’ or ‘a’..’z’ and converts it to a file offset number from 0 (for ‘0’) to 61 (for ‘Z’). This routine can also be useful for similar signalling methods often used in the µTasker web interface for control purposes.

Example:

```
unsigned char ucValue = fnConvertName(*ptrFileName);
```

```
extern MEMORY_RANGE_POINTER uOpenNextMimeFile(
     MEMORY_RANGE_POINTER ptrfileLocation,
     MAX_FILE_LENGTH *FileLength,
     unsigned char *ucMimeType,
     unsigned char ucSubfile);

extern MEMORY_RANGE_POINTER uOpenNextFile (
     MEMORY_RANGE_POINTER ptrfileLocation,
     MAX_FILE_LENGTH *FileLength,
     unsigned char ucSubfile)
```

These routines return a pointer to the next file in the µFileSystem and writes the file's length and MIME types (when a file exists). When `SUPPORT_MIME_IDENTIFIER` is enabled the routine `uOpenNextMimeFile()` is used or else `uOpenNextFile()` is used. The parameter `ucSubFile` is used only when `SUB_FILE_SIZE` is defined (large granularity memory).

`0` is returned if no (next) file is found; this means that there are no (further) files with content.
`ptrfileLocation` set to `0` returns the first existing file (equivalent to calling with the value set to `uFILE_SYSTEM_START`)

Examples (not large granularity memory and with MIME type):

```
MAX_FILE_LENGTH FileLength;
unsigned char ucMimeType;
MEMORY_RANGE_POINTER ptrNextFile;
MEMORY_RANGE_POINTER ptrFirstFile = uOpenNextMimeFile(0,
&FileLength, &ucMimeType); // get pointer to first existing file
if (ptrFirstFile != 0) {   // if there is a first file found
    ptrNextFile = uOpenNextMimeFile(ptrFirstFile,
        &FileLength, &ucMimeType);  // get pointer to next
                                       existing file
}
```

See the use in `ftp.c` for more examples, including handling of `ucSubfile`.

6. Low-Level FLASH Routines

In addition to the file system interface, there are user accessible low level routines supported by the FLASH driver of each device. The prototypes to these functions are in `hardware.h`.

These routines are used by the file system for accessing the memory and their operation details are generally hidden from the user. The functions can however be used if there is a special need to do so.

```
extern int  fnEraseFlashSector(unsigned char *ptrSector);
```

This routine is passed a pointer to any location within a sector of FLASH. The individual sector is then erased and the return value 0 indicates that the erase completed successfully.

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

This routine copies a buffer of data to the memory starting at the destination address. It has to be noted that the caller must ensure that the memory has been deleted previously since it is only possible to program bits from '1' to '0'. In some cases, depending on the device, it is possible to program individual bits and in some cases a minimum word size is imposed. Such details can be found in the user's manual to the specific device.

The routine will check to ensure that important programming rules are observed but may choose to not write if there is an invalid type of access. This is not signalled as an error but rather the particular write (possibly within a buffer) is silently ignored.

The value 0 is returned when no programming errors occurred.

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

This routine is used to retrieve the data directly from the device. It does no further checking.

7. µParameterSystem

The **µFileSystem** supports an optional parameter system allowing user data to be saved and restored as required. This is very useful for application variables which must be preserved when the system is powered off (examples are MAC and IP addresses, password information, serial interface settings, and many other possibilities).

The parameter system has also an option to use a 'swap-block' which will ensure that the parameters cannot be lost when the system is powered down just then as new values were being saved and also that previous parameters can be retrieved if necessary. The µTasker demo project makes use of this possibility when saving new IP configuration values which are used temporarily until the user confirms that they are good, or a trial-period times out with no confirmation. Since the original parameters are still available, they can be re-activated so that a device doesn't remain unreachable due to inappropriate settings.

The **µParameterSystem** can be activated by setting the define `USE_PARAMETER_BLOCK` in `config.h` and the 'swap-block' by setting the define `USE_PAR_SWAP_BLOCK` in `config.h`.

The size of the parameter block is defined by `PARAMETER_BLOCK_SIZE (`in `app_hw_xxxx.h`), which must be a multiple of the FLASH sector size (`FLASH_GRANULARITY`) as is the case for sizes in the file system. When `USE_PAR_SWAP_BLOCK` is activated, two such blocks are reserved and used.

The location of the **µParameterSystem** in memory is defined by `PARAMETER_BLOCK_START (`in `app_hw_xxxx.h`). This is generally independent of the **µFileSystem** and can this also be in different memory type (internal FLASH rather than external SPI FLASH, for example) but its location should be at a lower physical address value than the **µFileSystem** address space**.**

Here are the important differences between using the **µParameterSystem** with and without a 'swap-block'

- When no 'swap-block' is present, the block has to be deleted before modifying parameters. There is a risk of losing parameters if the system crashes or is powered off at this instant.
- A 'swap-block' requires twice the FLASH memory space since the parameters are present twice.
- Previous parameters can be recovered after testing temporary values when the 'swap-block' is activated.

The following are generally valid.

- The space for user parameters is the size of the parameter block minus 2 to 8 bytes (these 2 to 8 bytes are used for block management and the exact size depends on the hardware involved).
- If a device doesn't support "accumulative writes" it is necessary to use one FLASH word (size of word depends on device) per variable byte. An example of this is the M9S12NE64, which uses 16 bit FLASH words and cannot perform accumulative writes. When a block of user parameters is saved, each of the bytes in the block of variables is saved in an individual word. A parameter block of 512 bytes (the FLASH granularity in the NE64) can therefore save (512 – 4)/2 bytes [254] of user data. Other FLASH devices supporting accumulative writes with the same granularity can save (512 – 2) [510] bytes of user data. This means the limit is given by the chip and it is always best to know such details… There is a define in the header file belonging to the device called `MAX_SECTOR_PARS` which gives the limit for a parameter block of the size `FLASH_GRANULARITY`, which may be helpful…

The parameter block management is very simple but it ensures that it is possible to identify if a block is valid, as opposed to deleted or corrupted by a power cycle during writing. It also can identify a block with temporarily valid content which may require validation before being accepted for general use.

- A blank parameter block is a block of FLASH memory completely filled with the value 0xff.
- A block being written but not valid is a block with the first two entities (size of the entity is a byte where possible but can be up to an `unsigned long` if the FLASH doesn't support byte write) with the value 0xff (up to 0xffffffff). The rest of the data is presently being written with parameters and some locations will no longer be 0xff.
- A valid parameter block is identified by the first entity being set to 0x55 (up to 0x55555555). By simply checking the first entity for this pattern it is clear whether it contains valid data or not. When using "swap blocks" it is possible to have two valid blocks although only one of them is validated (see next paragraph). The valid entity is already written to the FLASH block after all data has successfully been saved – it therefore ensures that it is never possible to have a valid block with garbage in it due to a system crash.
- A validated block, as opposed to a temporary valid block, has the value 0x55 (up to 0x55555555) at the second entity location. By using the valid entity and also the validated entity it is possible to distinguish between the validated block and the temporary block.

The µTasker demo project makes use of the **µParameterSystem** and supports both single and swap blocks. Should the parameter block support be deactivated, the project will always run with default setting which means that the desired values (such as IP address) have to be hard coded and can then not be changed by the normal user.

Although the swap block can be deactivated, it is advised also to use it since its advantages are very useful in real-world projects.

The µTasker demo project provides additional cover functions which use the **µParameterSystem** interface routines but add a higher level of user comfort (examples are `fnGetEthernetPars()`, `fnGetOurParameters()` and `fnSaveNewPars()`). These serve as useful examples for your own application and can of course be used as they are in your own project if you wish.

The routine **`int`** `fnSaveNewPars(`**`int`**`)`, although an example cover function, is described here in a little more detail since it shows how the **µParameterSystem** is used in a typical application – this has in fact been used in many projects and so why not use its basic idea in your own? In the following examples it is assumed that a "swap block" is also being used. To get to know exactly how it works it is advised to use the µTasker simulator and set a break point in the function. The details become clear when the code is stepped though.

```
fnSaveNewPars(SAVE_NEW_PARAMETERS);
```

Here user parameters are saved from an application structure to the swap block. The swap block will be set as valid and also as validated while any existing block will be deleted.

```
fnSaveNewPars(SAVE_NEW_PARAMETERS_VALIDATE);
```

Here user parameters are saved from an application structure to the swap block. The swap block will be set as valid but not be validated. The original block (if existing) will therefore still contain the generally valid data. This new block can be validated at a later time (for example after verification that they are indeed suitable) and then used as general parameters rather than just temporary ones.

```
fnSaveNewPars(SAVE_NEW_PARAMETERS_CHECK_CRITICAL);
```

Here user parameters are saved only when no critical data has been changed. If critical data has indeed been modified, the user could then receive a warning before they are really saved.

8. µParameterSystem Routines

The **µParameterSystem** supports the following interface functions for retrieval, deleting and modifying information in the parameter system. Their prototypes are defined in `driver.h`.

```
extern int fnGetPar(unsigned short usParameterReference,
                    unsigned char *ucValue,
                    unsigned short usLength);
```

This function retrieves `usLength` bytes of data from the parameter block and copies them into the buffer pointed to by `ucValue`. `usParameterReference` is the offset to the start of data to be retrieved from the beginning of the data. The function returns a zero on success or a negative value if the requested parameters do not exist.

Either the validated parameter set or the temporary parameter set can be specified as shown in the following example:

```
unsigned char ucBuffer[10];
if (fnGetPar((TEMPORARY_PARAM_SET | 0) , ucBuffer, 10) < 0) {
    if (fnGetPar(0 , ucBuffer, 10) < 0) {
        // no parameters available, use defaults
    }
}
```

In the example, 10 bytes are retrieved from the start of the parameter block. First an attempt to retrieve them from the temporarily valid parameter block is made, followed by the validated block. If both fail it means that no parameters are available and so default values should be used.

```
extern int fnSetPar(unsigned short usParameterReference,
                    unsigned char *ucValue,
                    unsigned short usLength);
```

This function saves `usLength` data from the buffer `ucValue` to the parameter block at the offset `usParameterReference` from its start. Note that the caller should carefully manage the use of this function since it is only possible to change values in FLASH when they are previously deleted. Therefore this can be called a number of times to save blocks of data at various FLASH locations but a repeat of a save to certain block is likely to fail.

The routine returns a zero when the write was successful.

Examples:

```
unsigned char ucBuffer[10] = {0,1,2,3,4,5,6,7,8,9};
fnSetPar((TEMPORARY_PARAM_SET | 0) , ucBuffer, 10);
```

The data bytes 0..9 are written to the start of the temporary parameter block – if the temporary parameter block is initially empty it will be marked as temporarily valid.

```
unsigned char ucBuffer[10] = {0,1,2,3,4,5,6,7,8,9};
fnSetPar(0 , ucBuffer, 10);
```

The data bytes 0..9 are written to the start of the valid parameter block – if there is initially no valid parameter block (empty FLASH) a block will be marked as valid after this call.

```
extern int fnDelPar(unsigned char ucDeleteType)
```

This routine is used to delete and manipulate the parameter block(s). It returns a zero when successful.

Examples:

```
fnDelPar(INVALIDATE_PARAMETER_BLOCK);
```

All data will be deleted from the valid parameter block and the block will become invalid.

```
fnDelPar(INVALIDATE_TEST_PARAMETER_BLOCK);
```

All data will be deleted from the temporary parameter block and the block will become invalid.

```
fnDelPar(SWAP_PARAMETER_BLOCK);
```

All data will be deleted from the valid parameter block and the block will become invalid. A valid temporary parameter block will be set as validated block. This call is used to swap between a backup and a new parameter set.

9. Practical Example of a User Parameter Block

```
typedef struct stPARS
{
    unsigned char  ucParVersion;     // Versions number to avoid loading incorrect
                                        structure after major changes
    unsigned short telnet_timeout;   // TELNET timeout in seconds (0xffff means no
                                        timeout) - max 18 hours
    unsigned short usSerialMode;     // serial settings
    unsigned short usTelnetPort;     // the port number of our TELNET service
    unsigned char  ucServers;        // active servers
    unsigned char  ucSerialSpeed;    // Baud rate of serial interface
    unsigned char  ucTrustedIP[IPV4_LENGTH]; // Trusted IP address
    CHAR           cUserName[8];     // The single user administrator
    CHAR           cUserPass[8];     // The administrator's password
    CHAR           cDeviceIDName[21]; // 20 characters for recognition plus null
                                        terminator
    unsigned char  ucFlowHigh;       // High water flow level for XOFF/CTS (%)
    unsigned char  ucFlowLow;        // Low water flow level for XON/CTS (%)
    unsigned char  ucUserOutputs;    // ports set as outputs
    unsigned char  ucUserOutputValues; // port values
} PARS;
```

The µTasker demo project uses a structure containing various settings which can be modified and saved by the user. This structure is saved along with network parameters whenever the parameters are saved to FLASH using the **µParameterSystem** system.

This structure can be easily modified or extended to suit your own purposes and then the changed or extended parameters will also be saved.

Note that the first location is defined as a version number. This is very helpful when working on a project where the parameters are occasionally being modified since it allows the routine retrieving the parameters to be sure that the contents are compatible with the structure in the software.

Imagine that the variable `ucSerialSpeed` were to be removed. This would mean that the variables below it in the structure will move up by a byte and when retrieving a copy of the structure from FLASH which was saved using the previous version, they would then be shifted by one byte.

However if the version number is incremented every time a critical change is made, the retrieval routine can easily notice that the contents of FLASH are no longer valid and use the defaults instead. This avoids mistakes being made due to incompatible parameters and is recommended for general use.

10. Extended µFileSystem

In situations where the number of individual files supported by the basic µFileSystem is not adequate - either the number of files needs to be more that 62 or the size of memory available cannot be optimally utilised with the low quantity – the Extended µFileSystem option can be of use.

The extended operation is enabled in the project by the define `EXTENDED_UFILESYSTEM`, which is set to a length depicting the length of the file referencing as show below.

- If `EXTENDED_UFILESYSTEM` is set to 1 the file 'z' becomes equivalent to "z0".
- If `EXTENDED_UFILESYSTEM` is set to 2 the file 'z' becomes equivalent to "z00".
- If `EXTENDED_UFILESYSTEM` is set to 3 the file 'z' becomes equivalent to "z000".

The value informs of the number of digits that the final file in the basic µFileSystem space is extended by. Following the file "z0" are further possible files "z1", "z2", "z3", "z4" etc. up to the largest digit that can be displayed by the extension.

In the case of a three digit extension there can be extended files from "z000" up to "z999".

A few initial rules can already be defined (a three digit extension is used as basis for the examples):

- The size of the standard µFileSystem must be less than 62 file blocks so that the extended operation can automatically start operating once this is exceeded. This is because the 62nd file block is equivalent to the µFileSystem 'z'. If the standard size is set to more than 61 the code will generate an error message when compilation takes place. If this appears it means that the standard area size should be re-dimensioned accordingly.
- The file 'z' is the same as "z000". *FTP will however always display it with full extension digits*.
- 'z' can also be reference as "z0", "z00" or "zZ000". *The zero digits are assumed and automatically extended*.
- The same is true for general extended files. "z062" and "z62" are, for example, equivalent.
- The rules for storage in the extended area are otherwise the same as for storage in the non-extended area; files occupy as many file blocks are necessary to store their complete length; writing files will cause space for them to be made (bull-dozing) if necessary by deleting complete files occupying their required space.

Additional defines when working with the Extended µFileSystem:

- `EXTENSION_FILE_SIZE`
  this is required and defines the size of the file granularity in the extended file area. This value should be set to `FILE_GRANULARITY` at the time of writing. That means that the files in the standard area and in the extended area use the same file granularity.
  This may be changed in the future to allow different values to be used.
- `EXTENSION_FILE_COUNT_VARIABLE`
  this is optional and allows the size of the extended area to be configured at run-time based on the size of the memory detected. It allows the standard part of the

µFileSystem to exist in memory that is always available (such as in internal FLASH) and the extended area to be made up of optional extension memory which may or may not be present.
The memory driver used needs to also support this as an option so that it can enter the extended memory size at initialisation.

Finally note that the standard µFileSystem area need not consist of an area with the maximum 61 µFileSytsem files. It may, for example, only have a range of '0'..'9' if only 10 file blocks are used. The extended area always begins at "z000" in this case and all file names in between ('A', B' etc.) are equivalent to "z000"!

The following call shows an extended file being opened:

```
unsigned char *ptrFile = uOpenFile("z065.txt");
```

An example of a configuration for an extended area is shown below:

```
#define EXTENDED_UFILESYSTEM  3    // extend file names to "zXXX"
#define EXTENSION_FILE_SIZE   FILE_GRANULARITY
                                   // file granularity used in the extension memory
#define EXTENDED_FILE_SYSTEM_SIZE  (100 * 4 * 1024)
```

This example shows the extended area being enabled with a size of 400k (100 extended files assuming 4k Flash granularity). The standard area is defined in the normal manner and is not affected by the extended area definition; if the extended area is disabled the standard area is still available with the same characteristics.

11. Appending Data to Existing µFileSystem Files

When the optional define UFILESYSTEM_APPEND is enabled a further method is available which allows existing files to be extended. This can be useful in situations where the file's content is a result of logging data which grows in size as the operation continues or as system events occur.

The use of this extension is simple and involves a file to be opened with uOpenFileAppend() rather than with uOpenFile(). Writing data to this file now appends the data to the end of the file rather than writing a new file from the beginning. When the file is closed (using uFileClose() or uFileCloseMime()) this causes the file's header to be updated accordingly so that the file now contains its new length.

```
extern unsigned char *uOpenFileAppend(CHAR *ptrfileName);
```

This call passes a pointer to a file name and returns a pointer to its contents.

Example:

```
unsigned char *ptrFile = uOpenFileAppend("MyFile.txt");
```

uOpenFileAppend("M") is also adequate to access the same named file. The pointer to the contents can be later used for content retrieval and modification purposes.

When using the append feature there are a few important principles that need to be understood. The most important being the fact that it is generally not possible to change the file length without having to first perform a delete of the first FLASH sector in the file (an exception is when the µFileSystem is operating in EEPROM based memory where individual bytes can be modified). Adding data to an existing file in FLASH thus involves first writing the new data to the end of the file, then making a backup in SRAM of the initial FLASH sector of the file, deleting the first Flash sector, changing its file length value in the image and then writing back the complete first Flash sector.

The implications of the file length update in Flash are the following:

- Each time additional data is written and the file closed there is a delete/programming cycle performed at the file Flash section in the file area. This can potentially cause a high level of wear on this Flash sector if it is performed a large number of times.
- Since the content of the first Flash sector needs to be deleted there is a risk that the file length and first sector length of original data will be lost if the system crashes (due to a SW problem or power failure) during the time that the original data is in SRAM.
- If a device has a large Flash granularity it may not be realistic to use this method since the amount of data to be backed up in order to change the file length could be too large. In some systems with large Flash granularity the sector delete time can also be long and so cause additional difficulties with erase times.

Some Flash media doesn't allow single bytes to be written to empty Flash. In these cases it is also necessary to add append data first to an SRAM buffer so that the first append Flash sector can also be deleted and modified as new data is added.

Such details are handled in the µFileSystem code as required but the user should generally be aware of the fact that extra memory may be in use (the worst case is one Flash granularity sized backup buffer) and that such swap operations may be taking place.

As additional data is added to an existing file the µFileSystem automatically verifies if there are existing files occupying the area that it grows into. If files are found they are automatically deleted (bull-dozing) to make space for the growing file.

If the file grows to the end of the available file system space no more data will be added (return length of write indicates that not all data could be written) and the file is thus limited to the maximum possible size.

The following shows a typical use of the append method, whereby the file may or may not already exist:

```
unsigned char ucMimeType = MIME_BINARY;
MAX_FILE_LENGTH file_length;
unsigned char ucData[100];
unsigned char *ptrFile = uOpenFileAppend("MyFile.txt");
uMemset(ucData, 0x5a, 100);
uFileWrite(ptrFile, ucData, 100);
uFileCloseMime(ptrFile, &ucMimeType);
ptrFile = uOpenFileAppend("MyFile.txt");
uMemset(ucData, 0xb3, 100);
uFileWrite(ptrFile, ucData, 100);
uFileCloseMime(ptrFile, 0);
file_length = uGetFileLength(ptrFile);
```

The following details are to be noted:

- The open with append is possible whether the file initially exists or not.
- First 100 bytes are added to the file and it is closed.
- The file is opened for append a second time and a further 100 bytes of data are added to it before being closed again.
- The first write sets the file mime type to be a binary file. This ensures that the mime type is set when the file doesn't already exist. (`uGetFileLength(ptrFile)` could of course have been used to identify whether this was really the case).
- The second time that the file is closed a zero pointer is passed as mime type parameter, which means that the original mime type is retained. *This is only allowed when working with the append method!*
- Finally, the file length is checked, whereby it should be 200 bytes long if the file previously didn't exist, or 200 bytes longer than the original length as long as the µFileSystem's available area didn't restrict its growth.

12. Conclusion

The **µFileSystem** and **µParameterSystem** have been introduced, including a discussion of their advantages and restrictions in embedded projects. Using the guidelines contained here it should be possible to utilise their capabilities to achieve very efficient file and parameter operation in most real embedded projects.

The user interface routines for generic access to storage space (which can be either internal or external, or even a mixture of both in the same address space) have been detailed and practical examples given to increase the understanding of their use in real projects.

Optional extensions to allow an increased number of files in the µFileSystem or to enable appending data to existing files have been detailed in their own chapters.

It has been pointed out that the µTasker project includes also a FAT32 compatible file system (utFAT) for use with SD cards or other medium such as NAND Flash. Its user's manual can be consulted for complete details:
http://www.utasker.com/docs/uTasker/uTasker_utFAT.PDF

Modifications:
V0.02 5.1.2007:
- updated to include parameter description and be valid for *µTasker* release V1.3

V0.03 13.3.2008:
- Corrected file name ordering (0..9, A..Z, a..z)

V0.04 22.2.2009:
- Parameter block separated from µFileSystem. Corrected 96k address (`0x118000` to `0x18000`), some grammatical improvements and new cover sheet. Add conclusion and link to large-granularity document.

V0.05 25.09.2011:
- Reference to utFAT added. Optional extensions `EXTENDED_UFILESYSTEM` and `UFILESYSTEM_APPEND` added in new chapters.

V0.06 21.12.2011:
- Cleaned up some textual errors and modified configuration of the extended file system size to use `EXTENDED_FILE_SYSTEM_SIZE`.

V0.07 21.09.2013:
- Added notes that `uOpenFile()` always returns a memory pointer, even when the file doesn't exist (`uGetFileLength()` is used to check that the file has content). Added `uOpenNextMimeFile()` / `uOpenNextFile()` descriptions.

V0.08 04.10.2014:
- Specify that the extended file system requires the standard uFileSystem size to be smaller than 62 file blocks (latest code generates an error message when this is not respected). The extended file name has been changed from 'ZXXX' to 'zXXX' so that the extended file system can be used with maximum uFileSystem size.