Embedding it better...

µTasker Document

µTasker – FAT Emulation

Table of Contents

1. Introduction

FAT (File Allocation Table) has found wide spread use in file systems since its introduction in the late 1970s in the form of FAT12. Although FAT12, as is FAT16, is less common today due to the large size of small and cheap storage media, FAT32 still represents a popular file system used in many computer and embedded systems.

FAT can be considered as a standard file system format that can be used to ensure basic compatibility for almost all devices, even if some devices may also support more advanced modern file systems as well.

In some situations however the overhead of FAT may be a disadvantage. Some storage media also require additional software layers (block management and wear levelling, for example) due to the fact that the properties of their hardware storage elements are not well suited to FAT operation. FAT operations tend to make intensive use of small parts of the storage area which can, if not handled appropriately by additional algorithms to spread the load, lead to premature failure of the part.

The FAT emulation that is described in this document shows how *FAT read compatibility* can be retained while utilising linear storage. The FAT read emulation is included in the ***µTasker*** Mass-Storage module. Although more restrictions exist for FAT write emulation it is also used by the ***µTasker*** Serial Loader when operating as a USB-MSD loading device: http://www.utasker.com/docs/uTasker/uTaskerSerialLoader.PDF

2. Example of Linear Data Storage (Test-Vehicle)

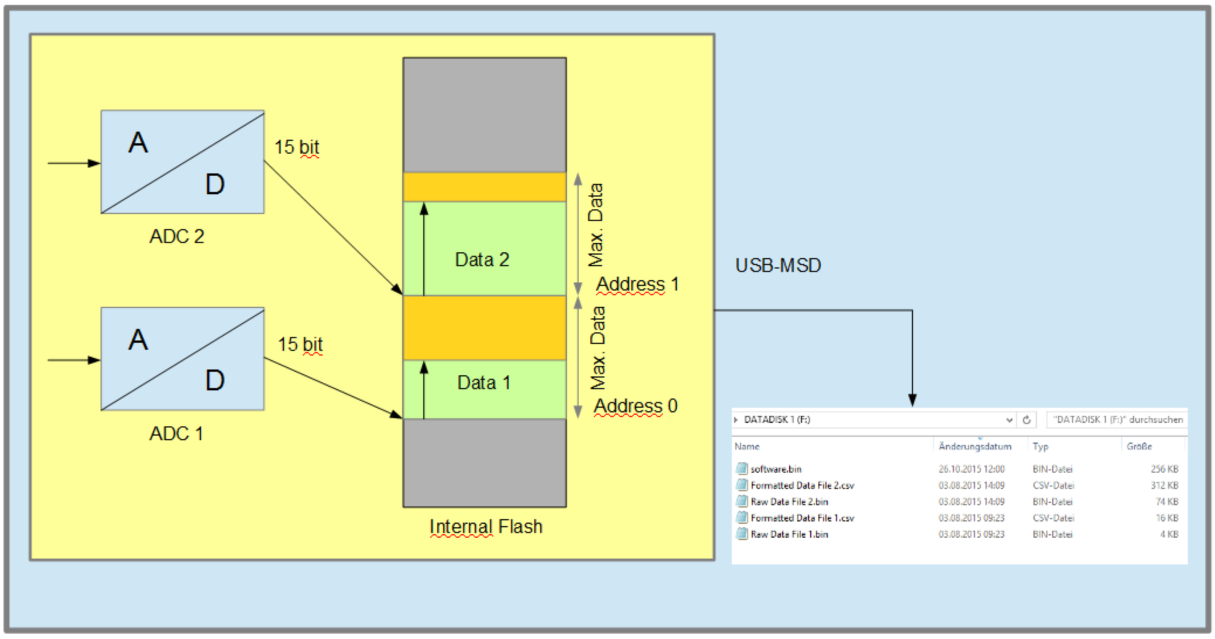
In order to illustrate the advantage that the FAT emulation can offer, a typical application example is used in which data is stored in internal Flash memory in a raw, linear format. This means that the data is stored in its natural binary format in a reserved area of memory (outside the area used by program code or other storage requirements), whereby the first data element is saved at the start of the area and subsequent data elements are stored contiguously after it in that area.

This is the simplest form of data storage, requiring very low software overhead and obtaining both maximum storage efficiency and write speed. It is assumed that the amount of data that has been saved is always known or can be determined (example, by scanning the area for deleted values or by using a simple header in the area maintaining the valid length).

To make the example a little more interesting two such data areas will be assumed, each of the same size and representing two possible data sets (or files).

By utilising the FAT emulation these two files are later to be visualised on a PC host via USB, whereby the natural method is USB-MSD so that the device appears as an external hard drive showing the two files as read-only files that can be copied to the PC.

Although the data elements are stored in their natural format the data read by the PC should be formatted so that it is readable by humans and can easily be input into data analysis tools – CSV (Comma-Separated Values) being a common format.



Name	Änderungsdatum	Typ	Größe
software.bin	26.10.2015 12:00	BIN-Datei	256 KB
Formatted Data File 2.csv	03.08.2015 14:09	CSV-Datei	312 KB
Raw Data File 2.bin	03.08.2015 14:09	BIN-Datei	74 KB
Formatted Data File 1.csv	03.08.2015 09:23	CSV-Datei	16 KB
Raw Data File 1.bin	03.08.2015 09:23	BIN-Datei	4 KB

This image represents the operation whereby there are two areas reserved in Flash for two linear blocks of data. The data arrives from two independent ADCs which may be supplying data to be saved to their respective block.

To simplify the management of the linear data 15 bit signed samples (left aligned) are used with the range -32768 to +32766 (0x8000 to 0x7ffe), meaning that it is easy to know how much data has already been saved to the data block because unused memory in flash memory will be at its erased state (0xffff). The controlling software therefore knows always how much data has been saved in each block, where the two blocks are physically located in its memory map (Address 0 and Address 1) and what is the maximum data that can be saved.

This application is therefore very simple in terms of code requirements and very efficient in use of storage space.

3. Adding Emulated FAT

By enabling USB-MSD device in the ***µTasker*** project the device appears as one or more external hard-drives to a PC host, which is a practical method for viewing and retrieving data from an SD card or other memory sources.

In order to allow the PC host to see a drive for the saved linear data an *emulated drive* is enabled which will supply the PC with information as if it were a FAT formatted disk containing data files detailed by the application. The application thus simple informs the emulated FAT drive of the amount of files that it would like to have displayed, their size, time stamps (if required) and their names.

When the PC reads content from the specified files the emulated FAT will request the application to supply it with the data content that should be returned. This means that the application also has the freedom to return whatever data it requires, either the raw binary content or a formatted representation of it. It is therefore possible to supply larger data content than is physically saved in the Flash, as show by the following example:

Data in Flash:

`0x0000, 0x0001, 0x0002, 0xfffe` [8 bytes physical storage required for 4 x 15 bit samples]

Date file content received by the PC host (CSV with ASCII decimal representation):

```
      0,      1,      2,     -2,
```

There are 8 ASCII characters for each short word of stored data, meaning that the size of the read data is 4 times larger than the physical storage. If FAT were used to save the data in its CSV formatted representation it would thus also require *more* than 4x the physical storage space (considering that the data sets in CSV *also* have additional separators).

4. Software Realisation

The example can be implemented very easily in the ***µTasker*** project by enabling the necessary framework support and then by writing a few functions to control the details required by the application. Rather than actually implement the ADC data capture, which is more an example of a practical application and not fundamental to the emulated FAT operation, the two files are filled automatically with a recognisable data pattern – for readers interested in ADC operation the following document gives details: http://www.utasker.com/docs/uTasker/uTaskerADC.pdf

The FRDM-K64F is used for this example since it is a popular development board with 1MBytes on internal Flash memory, although any other board with USB device and adequate internal memory can be used equivalently. The board is first selected in `config.h`

```
#define FRDM_K64F
```

Since USB-MSD is required to allow a PC host to see the device as an external hard-drive, USB is enabled:

```
#define USB_INTERFACE
```

together with the USB-MSD class:

```
#define USE_USB_MSD
```

Note that other composite USB device classes could be used at the same time by enabling them if desired (eg. `USE_USB_CDC` *for USB-CDC composite).*

The number of disks is selected by

```
#define NUMBER_USB_MSD 1
```

whereby the emulated FAT disk will be the only one used (an SD-card could, for example, be used at the same time as further disk if desired).

These defines enable the USB-MSD class operation, which is controlled in `usb_application.c`

In order for this to be able to use an emulated disk rather than physical disk (like an SD-card) the final configuration is to enable this support using

```
#define FAT_EMULATION
```

which automatically configures the USB-MSD's single disk as an emulated FAT disk, not requiring the full utFAT module.

The emulated FAT disk dimensions and maximum files are controlled by the following defines (more details about choosing the values is given later in the document):

```
#define EMULATED_FAT_DISK_SIZE  (1024 * 1024)                    // 1 MByte disk to be emulated
#define MAXIMUM_DATA_FILES        6 // maximum number of data files that could be used by the disk

#define ROOT_DIR_SECTORS          2 // 2 sectors is adequate for file information (including LFNs)
                                     for maximum file count
```

The USB-MSD will be handled by the USB task (in `usb_application.c`) but requires the application to supply two call-back handlers so that it can deliver information and data to the USB-MSD module when it requires it:

```
extern const unsigned char *fnGetDataFile(int iDisk,
                                          int iDataRef,
                                          EMULATED_FILE_DETAILS *ptrFileDetails);

extern int uDatacopy(int iDisk,
                     int iDataRef,
                     unsigned char *ptrSectorData,
                     const unsigned char *ptrSourceData,
                     int iLength);
```

These callbacks are explained when the complete code is discussed below.

Files to be Displayed

There are two data blocks used for storage of sampled data and these are to be made visible as two files which are *formatted as CSV*.

Furthermore, the two sets of samples are to be viewed as *binary data* too.

The sample data is stored from the address 0x40000 in internal Flash memory. This means that the firmware in the processor is stored in the area below this address. This area is also to be made visible as a raw binary file, which would allow the processor's *firmware* to be copied to the PC host if desired.

Finally an information file is to be displayed which contains *HTML* format for linking to further information on a web site.

The total file count is thus 6 files.

The detail of these files are set by the application when it starts in order to later have a simple way of referencing them when required. The initialisation is as follows:

```
// Prepare data files in linear flash (prime them if not already present)
//
static void fnPrepareEmulatedFAT(void)
{
    dataFile[0].ulDataFileLength = fnGetDataFileLength(LINEAR_DATA_1, DATA_FILE_1_LENGTH, 1);
                     // identify the present raw size of the first data file (prime data if nothing presently exists)
    dataFile[1].ulDataFileLength = fnGetDataFileLength(LINEAR_DATA_2, DATA_FILE_2_LENGTH, 0);
                    // identify the present raw size of the second data file (prime data if nothing presently exists)
    dataFile[0].ptrFileLocation = LINEAR_DATA_1;                // memory mapped address of file's raw data
    dataFile[0].ucFormatType = FORMAT_TYPE_RAW_BINARY;
    dataFile[1].ptrFileLocation = LINEAR_DATA_2;                // memory mapped address of file's raw data
    dataFile[1].ucFormatType = FORMAT_TYPE_RAW_BINARY;
    dataFile[2].ptrFileLocation = dataFile[0].ptrFileLocation;  // memory mapped address of file's raw data
    dataFile[2].ulDataFileLength = dataFile[0].ulDataFileLength +
         ((dataFile[0].ulDataFileLength/(2 * DATA_SET_CONTENT)) * RAW_DATA_SIZE);
                                                                // increase due to data group separators
    dataFile[2].ulDataFileLength *= FORMATTING_FACTOR;          // the increase in size of the file due to
                                                                   the ASCII formatting
    dataFile[2].ucFormatType = FORMAT_TYPE_CSV_FORMATTED;
    dataFile[3].ptrFileLocation = dataFile[1].ptrFileLocation;  // memory mapped address of file's raw data
    dataFile[3].ulDataFileLength = dataFile[1].ulDataFileLength +
         ((dataFile[1].ulDataFileLength/(2 * DATA_SET_CONTENT)) * RAW_DATA_SIZE);
                                                                // increase due to data group separators
    dataFile[3].ulDataFileLength *= FORMATTING_FACTOR;          // the increase in size of the file due to
                                                                    the ASCII formatting
    dataFile[3].ucFormatType = FORMAT_TYPE_CSV_FORMATTED;
    dataFile[4].ptrFileLocation  = FLASH_START_ADDRESS;         // data file locations in memory
    dataFile[4].ulDataFileLength = (unsigned long)(LINEAR_DATA_1 - FLASH_START_ADDRESS); // raw data length
    dataFile[4].ucFormatType = FORMAT_TYPE_RAW_BINARY;
    dataFile[5].ptrFileLocation  = (const unsigned char *)cHTML_link;    // data file locations in memory
    dataFile[5].ulDataFileLength = (sizeof(cHTML_link) - 1);    // raw string content length
    dataFile[5].ucFormatType = FORMAT_TYPE_RAW_STRING;
    #if defined EMULATED_FAT_FILE_DATE_CONTROL
    dataFile[0].usCreationTime = dataFile[2].usCreationTime =
                            (CREATION_SECONDS1 | (CREATION_MINUTES1 << 5) | (CREATION_HOURS1 << 11));
    dataFile[0].usCreationDate = dataFile[2].usCreationDate =
                            (CREATION_DAY_OF_MONTH1 | (CREATION_MONTH_OF_YEAR1 << 5) | (CREATION_YEAR1 << 9));
    dataFile[1].usCreationTime = dataFile[3].usCreationTime =
                            (CREATION_SECONDS2 | (CREATION_MINUTES2 << 5) | (CREATION_HOURS2 << 11));
    dataFile[1].usCreationDate = dataFile[3].usCreationDate =
                            (CREATION_DAY_OF_MONTH2 | (CREATION_MONTH_OF_YEAR2 << 5) | (CREATION_YEAR2 << 9));
    #endif
    #if defined EMULATED_FAT_FILE_NAME_CONTROL
        #if defined FAT_EMULATION_LFN
    dataFile[0].ptrFileName = "Raw Data File 1.bin";
    dataFile[1].ptrFileName = "Raw Data File 2.bin";
    dataFile[2].ptrFileName = "Formatted Data File 1.csv";
    dataFile[3].ptrFileName = "Formatted Data File 2.csv";
    dataFile[4].ptrFileName = "software.bin";
    dataFile[5].ptrFileName = "uTasker.html";
        #else
    dataFile[0].ptrFileName = "DATA-1.BIN";
    dataFile[1].ptrFileName = "DATA-2.BIN";
    dataFile[2].ptrFileName = "DATA-1.CSV";
    dataFile[3].ptrFileName = "DATA-2.CSV";
    dataFile[4].ptrFileName = "SOFTWARE.BIN";
    dataFile[5].ptrFileName = "UTASKER.HTM";
        #endif
    #endif
}
```

Each of the 6 file details are stored in structs so that they can later be accessed with a single file index reference.

```
static DATA_FILE_INFORMATION dataFile[APPLICATION_DATA_FILES] = {{0}};
```

which is defined as

```
typedef struct stDATA_FILE_INFORMATION

{
    unsigned long ulDataFileLength;
    const unsigned char *ptrFileLocation;
    #if defined EMULATED_FAT_FILE_NAME_CONTROL
        const CHAR *ptrFileName;        // short file name (8:3 format) or LFN (when FAT_EMULATION_LFN is enabled)
                                          [leave at 0 for default name]
    #endif
    #if defined EMULATED_FAT_FILE_DATE_CONTROL
        unsigned short usCreationTime;
        unsigned short usCreationDate;  // [leave at 0 for fixed date/time stamp]
    #endif
    unsigned char ucFormatType;
} DATA_FILE_INFORMATION;
```

The `ucFormatType` can be set to one of the following values:

```
#define FORMAT_TYPE_RAW_BINARY    0
#define FORMAT_TYPE_RAW_STRING    1
#define FORMAT_TYPE_CSV_FORMATTED 2
#define FORMAT_TYPE_INVALID_FILE  3
```

which defines how the content will later be copied.

The initialisation therefore consists of filling out the file structs with details about the location of the data belonging to the file in memory, the file's size (either in raw binary size of formatted size) as well as optional time stamp and file name. The routine used `fnGetDataFileLength()` to determine the size of the data files containing reference data and primes the memory with a sample pattern in case it does not yet exist. *This routine is not discussed in any more detail here and it can be assumed that it always return with the a positive file size and ensures that the processor's flash has a reference pattern ready*.

In the case of the files that are formatted their size is to be larger than their raw content size. As seen in the `fnGetDataFile()` code for files 2 and 3, these files sizes are increased by a factor

```
(x + ((x/(2 * DATA_SET_CONTENT)) * RAW_DATA_SIZE)) * FORMATTING_FACTOR)
```

to compensate for their formatted content size.

For the CSV formatting the following defines are used, which control also the number of data values in each data group and the separator character to be used. The generated format can be see later in the document.

```
// Formatting algorithm details
//
#define RAW_DATA_SIZE         2      // data samples are each 2 bytes in size
#define FORMATTING_FACTOR     4      // each short word input is formatted to ASCII decimal "-32000, "
#define DATA_SET_CONTENT      16     // after each 16 formatted values a line feed sequence is inserted "    \r\n"
#define CSV_SEPARATOR_VALUE   ',';
```

In order to control the location of the data in the Flash memory the following defines are used:

```
// Fixed details of test data files in linear memory
//
#define DATA_FILE_1_LENGTH    3722                                  // the number of raw data samples for file 1
#define DATA_FILE_2_LENGTH    74982                                 // the number of raw data samples for file 2
#define MAX_DATA_FILE_LENGTH  (128 * 1024)
#define LINEAR_DATA_1         (const unsigned char *)(256 * 1024) // internal flash address of the start of the
                                                                      file's raw data
#define LINEAR_DATA_2         (const unsigned char *)(LINEAR_DATA_1 + MAX_DATA_FILE_LENGTH)
                                                    // internal flash address of the start of the file's raw data
```

The final file doesn't use data from general Flash but instead used a string (from code space) which will later represent the file content. For completeness the string, which is simple HTML code, is shown here:

```
static const CHAR cHTML_link[] = "<html><head><title>Emulated FAT</title></head><body bgcolor=#d8d8d8
marginheight=30><center><font color=#ff0000 style=font-size:30px><b style='mso-bidi-font-weight:normal'>&micro;Tasker
FAT Emulation</font></b><br><br><br>Full details of this and many other features can be found at <a
href=""http://www.uTasker.com/"">the uTasker web site</a>.</body></html>";
```

Choosing the Value of ROOT_DIR_SECTORS

ROOT_DIR_SECTORS specifies the number of sectors that are required to hold all file objects. In order to best select this value one needs to know what file objects basically are and how many can fit into a sector (a sector is 512 bytes in size). The root directory contains always a volume entry (the name of the disk) plus an optional number of files (0 to MAXIMUM_DATA_FILES) which each have a file object containing their details (name, length, time etc.) and if only short file names are being used the total number of such objects is simply (MAXIMUM_DATA_FILES + 1). A file object is 32 bytes in size and each sector in the root directory can hold 16 objects. The number of sectors required is thus ((MAXIMUM_DATA_FILES + 16)/16).

In the case of long file names the required number of sectors is not so simple to calculate since long file names occupy a number of short file name objects and so the total number will depend also on the actual length of each name being used; the longest allowed long file name (with 255 characters) can occupy as many as 20 short file name objects!

When long file names are being used each will occupy a minimum of 2 short file name objects and increase by one for each additional 13 characters in the name; one can thus estimate the maximum space required, knowing the types of names used and also the number of files that will need to be displayed.

The reason for needing to specify this is so that the emulated FAT can construct the content of the root directory in RAM (of the required size to hold it) at initialisation and then simply return the content each time it is requested. This respresents the simplest and most efficient generation technique but does require this extra RAM buffer space (ROOT_DIR_SECTORS x 512 bytes).

Should RAM utilisation need be kept to an absolute minimum it is possible to remove the definition of `ROOT_DIR_SECTORS` which causes the emulated FAT code to not keep a copy of the root directory content, in which case also no dimensioning of it is necessary.

The trade-off with this option is however that each time the host requests content of the FAT sector(s) representing the root directory the content needs to be recalculated. This requires much of the emulated FAT initialisation to be executed each time it happens and represents a more complicated solution which trades off run-time processing against speed of operation and memory buffers requirement. As the number of root sectors increases so does the calculation effort required to regenerate their content.

Informing the FAT Emulator of File Information

When the FAT Emulator is initialised by the USB-MSD device it calls the routine `fnGetDataFile()` with a file reference for each possible file to be viewed.

Since `MAXIMUM_DATA_FILES` has been set to 6, the routine will be called 6 times with `iDataRef` values of 0 through 5.

```
extern const unsigned char *fnGetDataFile(int iDisk, int iDataRef, EMULATED_FILE_DETAILS *ptrFileDetails);
```

The application's task is to fill out `ptrFileDetails` with details for each file that is to be viewed. The return value for a valid file is a pointer to its start location in the memory map. The following is the complete routine to return the information about the six files.

```
// The application must supply this function when using FAT_EMULATION
// - the FAT emulator calls it to obtain referenced file information (formatted length, location in memory, creation
date/time and name)
//
extern const unsigned char *fnGetDataFile(int iDisk, int iDataRef, EMULATED_FILE_DETAILS *ptrFileDetails)
{
    if (iDisk != 0) {                                                  // data belongs to first disk only
        return 0;
    }
    if ((iDataRef < APPLICATION_DATA_FILES) && (dataFile[iDataRef].ucFormatType != FORMAT_TYPE_INVALID_FILE)) {
        ptrFileDetails->ulFileLength = dataFile[iDataRef].ulDataFileLength; // raw content length of file
    #if defined EMULATED_FAT_FILE_DATE_CONTROL
        ptrFileDetails->usCreationTime = dataFile[iDataRef].usCreationTime;
        ptrFileDetails->usCreationDate = dataFile[iDataRef].usCreationDate; // (leave at 0 for fixed date/time stamp)
    #endif
    #if defined EMULATED_FAT_FILE_NAME_CONTROL
        ptrFileDetails->ptrFileName = dataFile[iDataRef].ptrFileName;
    #endif
        ptrFileDetails->ucValid = 1;                                   // file is valid
        return (dataFile[iDataRef].ptrFileLocation);                   // memory mapped address of file's raw data
    }
    else {
        ptrFileDetails->ucValid = 0;                                   // file is not valid
        return 0;
    }
}
```

Due to the fact that the details had been prepared during the application initialisation it is just a case of coping each file's information based on `iDataRef` used as index.

When there is a single emulated FAT disk `iDisk` will always be 0. Should multiple disks be used the application can define which files are viewed on which disks.

A file marked as invalid will not be make visible by the emulated FAT module. Those marked as valid will be shown, even if their size if zero. In each case, the file's size and validity field is filed out and a pointer is returned to its memory location.

Depending on project options, the application can further detail how the file is displayed:

- `EMULATED_FAT_FILE_DATE_CONTROL` allows the application to set a time stamp for the file. Its format is the same as that used by the FAT file object.

A value of 0 for `ptrFileDetails->usCreationDate` results in the FAT emulator using a default time stamp, as it does for all files when the option is not enabled.

An example of times used in the code above, which is compatible with the FAT file object's usage is:

```
#define CREATION_HOURS1         8

#define CREATION_MINUTES1       23
#define CREATION_SECONDS1       1

#define CREATION_DAY_OF_MONTH1  3
#define CREATION_MONTH_OF_YEAR1 8
#define CREATION_YEAR1          (2015 - 1980)
```

although if the data was saved with a time stamp of its own the actual time/date could be used.

- `EMULATED_FAT_FILE_NAME_CONTROL` allows the application to specify the file name to be used.
  When `FAT_EMULATION_LFN` is also enabled the file name can be a long file name, otherwise it should be set to a short one (8:3 format).
  If `ptrFileDetails->ptrFileName` is set to zero, or the naming option not enabled the FAT emulator will name the files as "**`DATA_000.BIN`**", "**`DATA_001.BIN`**", "**`DATA_002.BIN`**" etc.
  Should any of the files not be valid, its number will not be used; eg. If files 0, 1 and 3 are valid but file 2 not, the enabled files will be named "**`DATA_000.BIN`**", "**`DATA_001.BIN`**", "**`DATA_003.BIN`**" respectively.

Inserting Data into the Emulated Files

When the PC host reads the content of the emulated files the following routine is called, which is provided by the application and needs to fill out the data content to be returned:

```
// The application must supply this function when using FAT_EMULATION
// - the FAT emulator calls it to obtain referenced file content, which can be returned as raw data or formatted
// - this example formats raw binary data to CSV format so that it can be easily opened in various PC programs for
viewing and processing
//
extern int uDatacopy(int iDisk, int iDataRef, unsigned char *ptrSectorData, const unsigned char *ptrSourceData,
                     int iLength)
{
    int iAdded = 0;
    unsigned long ulNextDataSetEnd;
    unsigned long ulDataOffset;
    int iStringLength;
    CHAR *ptrBuf;
    unsigned long ulThisDataLength;
    signed short *ptrRawSource;

    if ((iDisk != 0) || (iDataRef >= APPLICATION_DATA_FILES) ||
        (dataFile[iDataRef].ucFormatType == FORMAT_TYPE_INVALID_FILE)) { // data belongs to first disk only
        return 0;                                                        // no data added
    }
    switch (dataFile[iDataRef].ucFormatType) {
    case FORMAT_TYPE_RAW_BINARY:
    case FORMAT_TYPE_RAW_STRING:
        iAdded = (dataFile[iDataRef].ulDataFileLength - (dataFile[iDataRef].ptrFileLocation – ptrSourceData));
                                                                         // remaining raw content length
        if (iAdded > iLength) {
            iAdded = iLength;
        }
        if (FORMAT_TYPE_RAW_STRING == dataFile[iDataRef].ucFormatType) {
            uMemcpy(ptrSectorData, ptrSourceData, iAdded);      // strings are in code so copy directly from memory
        }
        else {
            fnGetParsFile((unsigned char *)ptrSourceData, ptrSectorData, iAdded); // prepare the raw data
        }
        break;

    case FORMAT_TYPE_CSV_FORMATTED:
        // CSV Formatted files
        //
        ulDataOffset = (ptrSourceData - dataFile[iDataRef].ptrFileLocation); // the physical offset
        ulDataOffset /= (FORMATTING_FACTOR * RAW_DATA_SIZE);             // raw data entry offset
        ulNextDataSetEnd = (ulDataOffset%(DATA_SET_CONTENT + 1));
        ulDataOffset = (((ulDataOffset/(DATA_SET_CONTENT + 1)) * DATA_SET_CONTENT) + ulNextDataSetEnd);
        ptrRawSource = (signed short *)dataFile[iDataRef].ptrFileLocation; // pointer to the raw data in memory
        ptrRawSource += ulDataOffset;
        ptrSourceData = (dataFile[iDataRef].ptrFileLocation + ulDataOffset);
        ulThisDataLength = (dataFile[iDataRef].ulDataFileLength -
                           (ptrSourceData - dataFile[iDataRef].ptrFileLocation)); // the remaining raw data length
        while (iLength > 0) {                                            // while data is to be added
            if (ulThisDataLength == 0) {                                 // no more data to add
                break;
            }
            if (ulNextDataSetEnd == DATA_SET_CONTENT) {
                uMemcpy(ptrSectorData, "      \r\n", 8);
                ulNextDataSetEnd = 0;
            }
            else {
                signed short sValue;
                fnGetParsFile((unsigned char *)ptrRawSource, (unsigned char *)&sValue, sizeof(sValue));
                                                                // get the next raw sample from the linear data area
                ptrBuf = fnBufferDec(sValue, DISPLAY_NEGATIVE, (CHAR *)ptrSectorData);
                                                                // the ASCII decimal representation of the sample
                iStringLength = (ptrBuf - (CHAR *)ptrSectorData);
                if (iStringLength < 6) {
                    ptrBuf = (CHAR *)(ptrSectorData + (6 - iStringLength));
                    uReverseMemcpy(ptrBuf, ptrSectorData, iStringLength);
                    ptrBuf = (CHAR *)ptrSectorData;
                    while (iStringLength++ < 6) {
                        *ptrBuf++ = ' ';
                    }
                    ptrBuf = (CHAR *)(ptrSectorData + 6);
                }
                *ptrBuf++ = CSV_SEPARATOR_VALUE;
                *ptrBuf = ' ';
                ptrRawSource++;
                ulThisDataLength -= RAW_DATA_SIZE;
                ulNextDataSetEnd++;
            }
            iLength -= 8;
            iAdded += 8;
            ptrSectorData += 8;
        }
        break;
    default:
        break;
    }
    return iAdded;                                              // the length added to the buffer
}
```

As in the case of the routine to inform of the data size and location, this routine is also passed a file reference. `PtrSectorData` is a pointer to a buffer which should be filled with `iLength` of data, where this is usually 512 bytes for FAT.

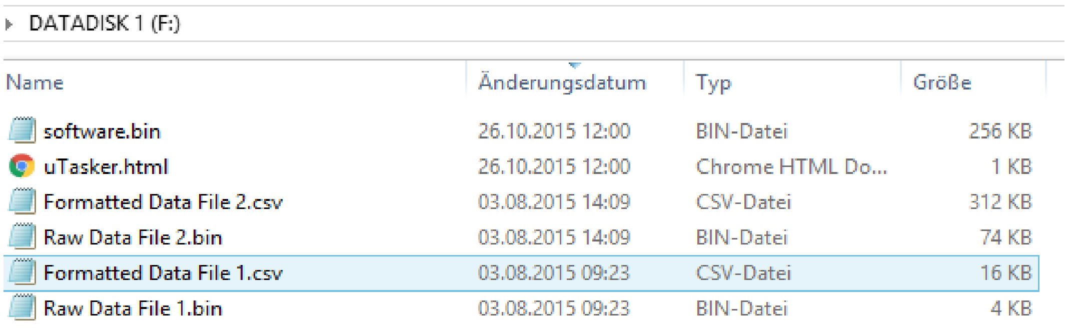
`PtrSourceData` is a pointer to the file's raw data, which may have an offset from its start address. Based on the offset the routine can calculate the location of the first byte to be returned from the memory area.

In the case of the raw binary files the operation is very simple since the code simply collects the data from the storage area (using `fnGetParsFile()` for accesses to Flash (allowing the data to be in any supported Flash type) so that it could be in internal, external SPI Flash, etc.) and puts it into the output buffer. See http://www.utasker.com/docs/uTasker/uTaskerFileSystem_3.PDF for more details about the memory retrieval function.

The formatted case is a little more complicated since the code needs to use an algorithm to work out which data should next be inserted and also needs to format the binary input to the ASCII output. Further complication stems from the fact that the formatted data samples are grouped, which requires a group terminator to be regularly inserted. The shown code allows the CSV file content to be generated when required and other formatting methods could be added here in case ever required.

5. Results

When connected to a PC host via USB the processor appears as an external hard drive with the volume name “**DATADISK 1**”. The drive is write-protected.



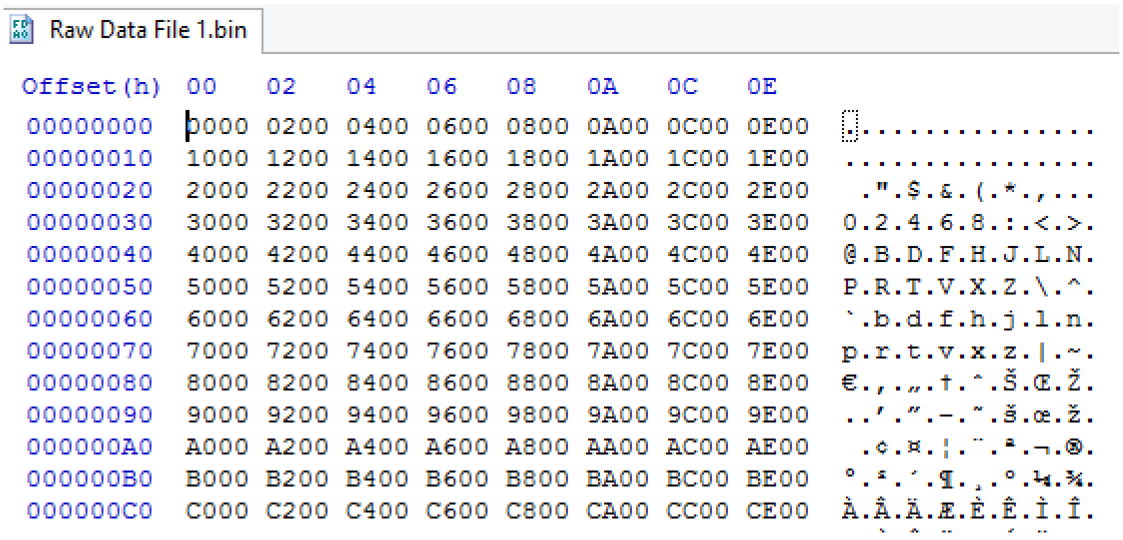
DATADISK 1 (F:)

Name	Änderungsdatum	Typ	Größe
software.bin	26.10.2015 12:00	BIN-Datei	256 KB
uTasker.html	26.10.2015 12:00	Chrome HTML Do...	1 KB
Formatted Data File 2.csv	03.08.2015 14:09	CSV-Datei	312 KB
Raw Data File 2.bin	03.08.2015 14:09	BIN-Datei	74 KB
Formatted Data File 1.csv	03.08.2015 09:23	CSV-Datei	16 KB
Raw Data File 1.bin	03.08.2015 09:23	BIN-Datei	4 KB

The files are visible as defined by the application, whereby the `.csv` files are much larger than their corresponding `.bin` files.

The files can be copied to the PC host or directly opened for viewing.

The following image shows an extract of the content of `Raw Data File 1.bin` as shown in a binary editor:

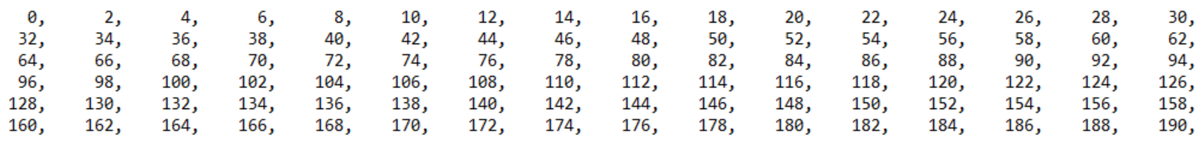


Raw Data File 1.bin

```
Offset(h) 00   02   04   06   08   0A   0C   0E
00000000  0000 0200 0400 0600 0800 0A00 0C00 0E00  ................
00000010  1000 1200 1400 1600 1800 1A00 1C00 1E00  ................
00000020  2000 2200 2400 2600 2800 2A00 2C00 2E00   .".$.&.(.*.,...
00000030  3000 3200 3400 3600 3800 3A00 3C00 3E00  0.2.4.6.8.:.<.>.
00000040  4000 4200 4400 4600 4800 4A00 4C00 4E00  @.B.D.F.H.J.L.N.
00000050  5000 5200 5400 5600 5800 5A00 5C00 5E00  P.R.T.V.X.Z.\.^.
00000060  6000 6200 6400 6600 6800 6A00 6C00 6E00  `.b.d.f.h.j.l.n.
00000070  7000 7200 7400 7600 7800 7A00 7C00 7E00  p.r.t.v.x.z.|.~.
00000080  8000 8200 8400 8600 8800 8A00 8C00 8E00  €.‚.„.†.ˆ.Š.Œ.Ž.
00000090  9000 9200 9400 9600 9800 9A00 9C00 9E00  ..’.”.–.˜.š.œ.ž.
000000A0  A000 A200 A400 A600 A800 AA00 AC00 AE00   .¢.¤.¦.¨.ª.¬.®.
000000B0  B000 B200 B400 B600 B800 BA00 BC00 BE00  °.².´.¶.¸.º.¼.¾.
000000C0  C000 C200 C400 C600 C800 CA00 CC00 CE00  À.Â.Ä.Æ.È.Ê.Ì.Î.
```

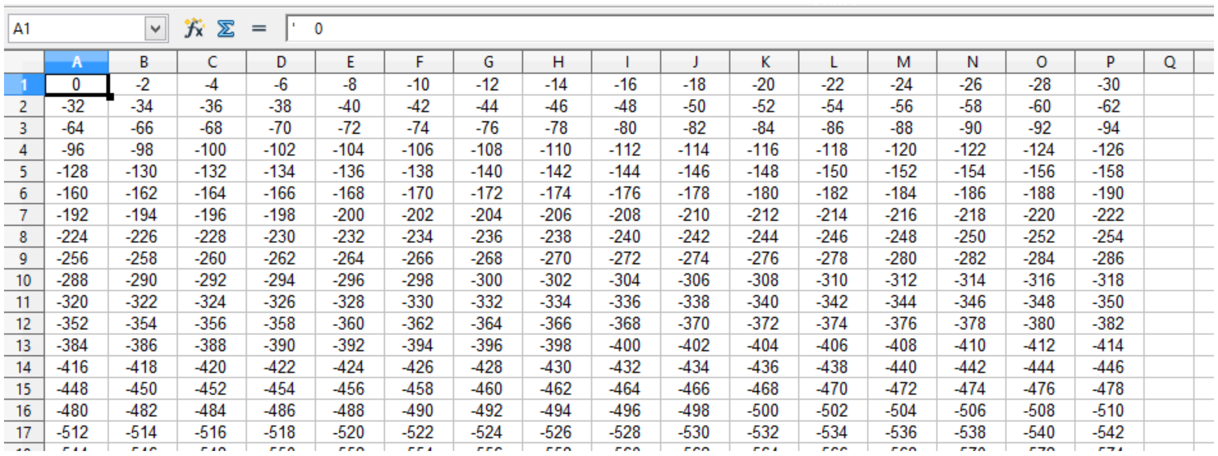
It is seen that the sample values increase 0x0000, 0x0002, 0x0004 etc. (although they are displayed here in the alternative endian-ness).

When `Formatted Data File 1.csv` is opened instead the same values are seen as comma-separated decimal values:



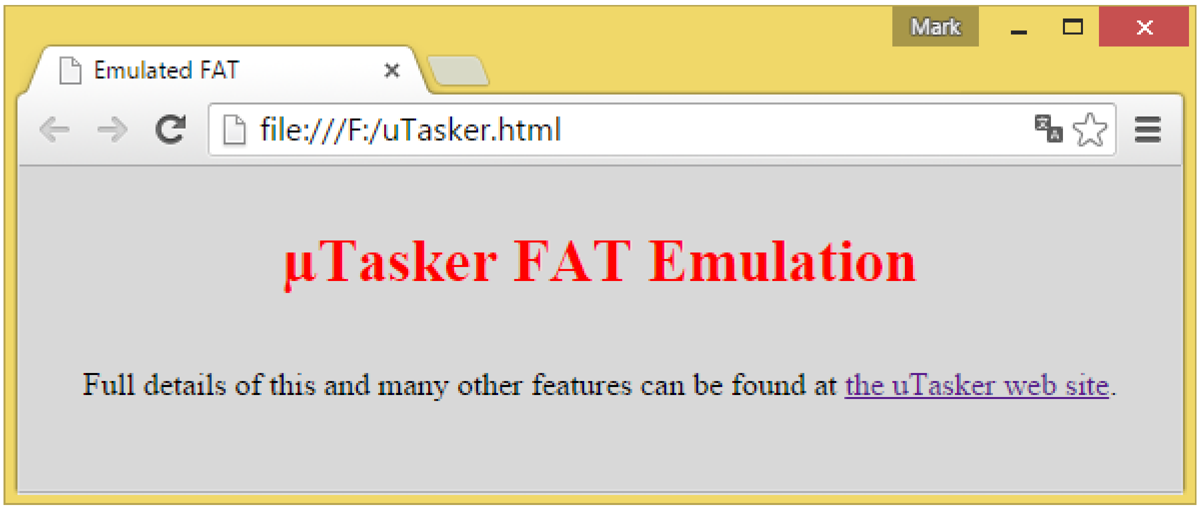
```
   0,     2,     4,     6,     8,    10,    12,    14,    16,    18,    20,    22,    24,    26,    28,    30,
  32,    34,    36,    38,    40,    42,    44,    46,    48,    50,    52,    54,    56,    58,    60,    62,
  64,    66,    68,    70,    72,    74,    76,    78,    80,    82,    84,    86,    88,    90,    92,    94,
  96,    98,   100,   102,   104,   106,   108,   110,   112,   114,   116,   118,   120,   122,   124,   126,
 128,   130,   132,   134,   136,   138,   140,   142,   144,   146,   148,   150,   152,   154,   156,   158,
 160,   162,   164,   166,   168,   170,   172,   174,   176,   178,   180,   182,   184,   186,   188,   190,
```

Since many programs can open or import `.csv` formatted data the final image shows the content of `Formatted Data File 2.csv` opened in a table calculation program, whereby it can be seen that these samples increase in value.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	0	-2	-4	-6	-8	-10	-12	-14	-16	-18	-20	-22	-24	-26	-28	-30	
2	-32	-34	-36	-38	-40	-42	-44	-46	-48	-50	-52	-54	-56	-58	-60	-62	
3	-64	-66	-68	-70	-72	-74	-76	-78	-80	-82	-84	-86	-88	-90	-92	-94	
4	-96	-98	-100	-102	-104	-106	-108	-110	-112	-114	-116	-118	-120	-122	-124	-126	
5	-128	-130	-132	-134	-136	-138	-140	-142	-144	-146	-148	-150	-152	-154	-156	-158	
6	-160	-162	-164	-166	-168	-170	-172	-174	-176	-178	-180	-182	-184	-186	-188	-190	
7	-192	-194	-196	-198	-200	-202	-204	-206	-208	-210	-212	-214	-216	-218	-220	-222	
8	-224	-226	-228	-230	-232	-234	-236	-238	-240	-242	-244	-246	-248	-250	-252	-254	
9	-256	-258	-260	-262	-264	-266	-268	-270	-272	-274	-276	-278	-280	-282	-284	-286	
10	-288	-290	-292	-294	-296	-298	-300	-302	-304	-306	-308	-310	-312	-314	-316	-318	
11	-320	-322	-324	-326	-328	-330	-332	-334	-336	-338	-340	-342	-344	-346	-348	-350	
12	-352	-354	-356	-358	-360	-362	-364	-366	-368	-370	-372	-374	-376	-378	-380	-382	
13	-384	-386	-388	-390	-392	-394	-396	-398	-400	-402	-404	-406	-408	-410	-412	-414	
14	-416	-418	-420	-422	-424	-426	-428	-430	-432	-434	-436	-438	-440	-442	-444	-446	
15	-448	-450	-452	-454	-456	-458	-460	-462	-464	-466	-468	-470	-472	-474	-476	-478	
16	-480	-482	-484	-486	-488	-490	-492	-494	-496	-498	-500	-502	-504	-506	-508	-510	
17	-512	-514	-516	-518	-520	-522	-524	-526	-528	-530	-532	-534	-536	-538	-540	-542	

The final file that appears on the disk is `uTasker.html` which, when double clicked, opens in a browser to give a link to the **µTasker** web site. This is a convenient method of supplying files for users containing brief information and such links to further more extensive documentation, such as user's manuals in the Internet.



µTasker FAT Emulation

Full details of this and many other features can be found at the uTasker web site.

The binary file from this reference is available at the FRDM-K64F page:
http://www.utasker.com/kinetis/FRDM-K64F.html

6. Conclusion

The example has shown how easy it is to add FAT emulation to a project by simply enabling USB-MSD and the emulation module, plus adding suitable interface routines to control the files to be viewed and their content and/or formatting.

µTasker – FAT Emulation not only allows visualisation of data stored in linear memory as if it were FAT formatted but also convenient retrieval by drag-and-drop using the PC host's file manager.

Further advantages of FAT emulation are that the files can be stored in raw format in linear memory, but still be retrieved as if they were saved in a formatted file. This saves space and also complexity compared to using a complete FAT module, and is faster and more convenient for most Flash storage media.

Modifications:

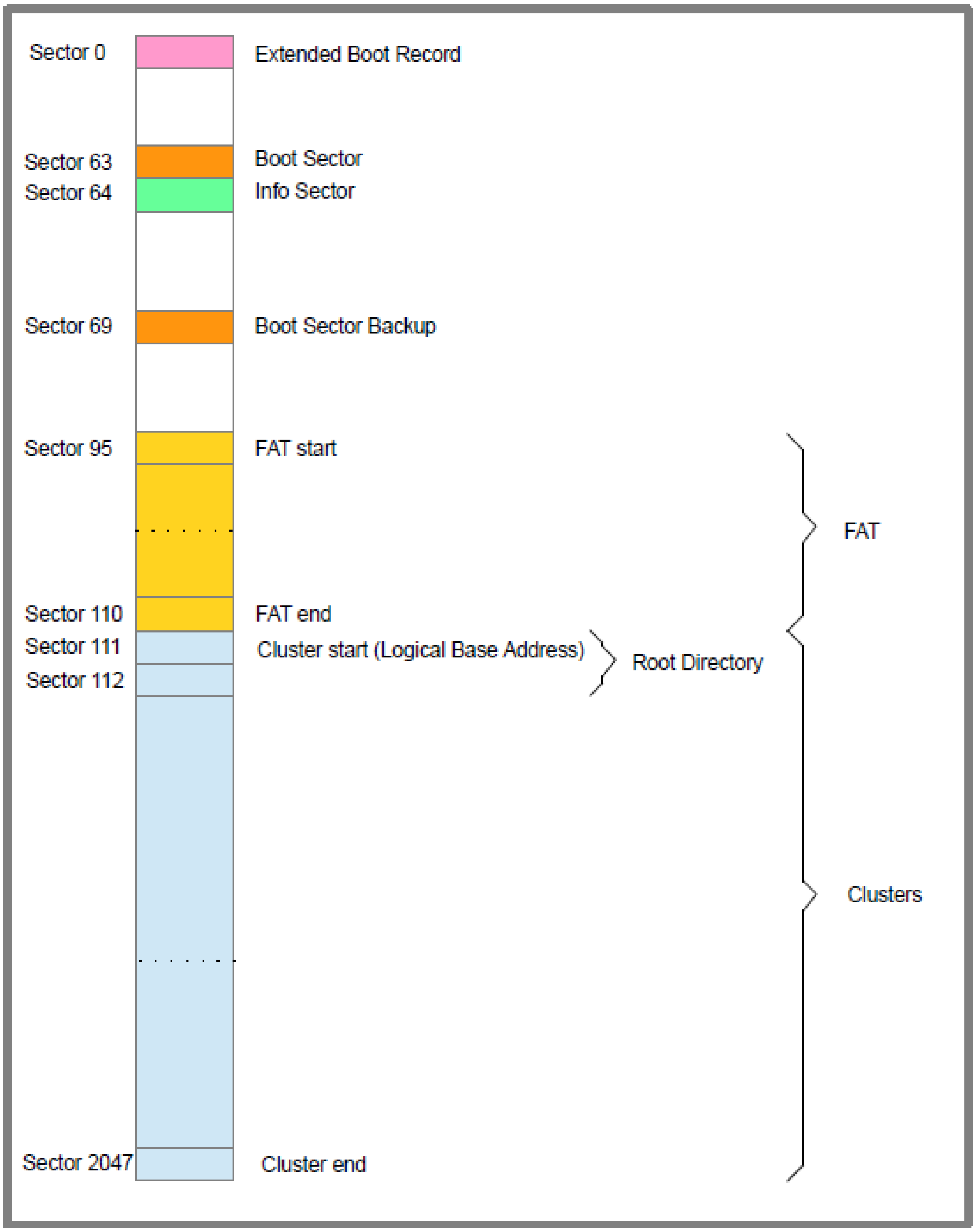
- V1.00 29.10.2015: - First release.
- V1.01 31.10.2015: - Added details about root directory memory and appendix explaining the FAT emulation technique used.

Appendix A – FAT Emulation Operation

A host requests data from a storage medium as one or more sectors (512 bytes in size). Using these sectors content the host can determine the type of file system it uses and all details about the data stored on it. FAT emulation therefore involves returning sector data containing content that makes the host believe that it is FAT, as well as data content when appropriate.

The emulated FAT shares the utFAT code to virtually format the emulated disk according to its size, with a singe partition and extended boot record pointing to a boot sector (which has a single backup) Such a disk then has a number of FATs (the emulated case uses only one FAT since there is no advantage in having a back-up of it), an information sector (when FAT32), some unoccupied sectors and finally clusters, which make up the majority of the storage space. At the start of cluster space is the root directory which contains the details of all files at the highest level on the disk. No directories are used and so all files are in the root directory. For more details about these topics see the utFAT User's Guide - http://www.utasker.com/docs/uTasker/uTasker_utFAT.PDF

The following illustration shows a typical case (1MByte FAT32 formatted disk) where two sectors are required to hold the files in the root directory – in this case each cluster consists of one sector but larger emulated disks may use multiple sectors for each cluster:



The emulation this consists of identifying in which sector is in when read by the host. If it is the Extended Boot sector, the info sector or one of the Boot sectors it returns the same content as it would write to the sector if it were formatting the disk.

When a root directory sector content is will cause the root directory content to be returned (with information about which files are in it, their names, size and data locations).

A request of a FAT sector results in the corresponding FAT content to be returned, which depends on the data content since the FAT area informs of which clusters are free and how clusters are chained (for example the first two clusters are chained since the both belong to the root directory).

A request of clusters that are occupied by the root directory requires the data that occupies them to be returned – this depends again on the files available and the clusters that their content occupy, whereby there are usually multiple chained clusters belonging to each file.

If file data in a sector is less that the sector size, the rest of the returned sector data is set to 0.

Any requests of clusters that are not occupied by file data causes a sector with all 0 content to be returned, as is the case when an unused/reserved sector is requested.