CODE TIME TECHNOLOGIES

Abassi RTOS

Media I/F

Disclaimer

Table of Contents

List of Figures

List of Tables

1 Introduction

This document provides a description and explains how to set-up the Media Interface module used in Abassi[1] [R1] (including mAbassi [R2] and µAbassi [R3]). The Media Interface is the layer located between the file system stacks (e.g. FAT 32 file system) and the Abassi drivers alike QSPI and SD/MMC. The Media Interface consists in fact of two layers. The lower one is a single file common to all file system stacks and it is used as a uniform interface between media stack specific interface file and Abassi's drivers. The upper layer is the media specific stack file that interfaces the file system stack and the common Media Interface layer.

The API won't be described because each media stack uses their proprietary API to interface with the media and there are no reasons to directly use the common interface layer. If a new media stack is desired to be added the common interface layer, the API is well described in the headers of the few functions and any of the already supported media stack API with the common interface can be used as a template. The file system stack specific code is quite small in fact and around half the file is comments.

1.1 Distribution Contents

The Media Interface module consists of the following files:

Table 1-1 Distribution

File Name	Description
`MediaIF.h`	Include file associated with `MediaIF.c`
`MediaIF.c`	"C" file for the lower layer media Interface
`Media_FatFS.c`	"C" file to interface the FatFS file system with `MediaIF.c`
`Media_FullFAT.c`	"C" file to interface the FullFAT file system with `MediaIF.c`
`Media_ueFAT.c`	"C" file to interface the ueFAT file system with `MediaIF.c`

The media stack specific interface files (`Media_FatFS.c`, `Media_FullFAT.c` and `Media_ueFAT.c`) above are available at the time of writing. Over time it is likely that extra ones being added.

1.2 System Call Layer

The Media Interface is designed for, so it is is tightly coupled with the System Call layer [R4] and very little extra set-up is needed to achieve this. All there is to do use the System Call Layer with media accesses is to include in the app are the following files:

- `SysCall_`*`STACK`*`.c` File system stack specific system call layer
- `SysCall_`*`COMPILER`*`.c` Non-GCC compiler specific system call layer I/F
- `Media_`*`STACK`*`.c` File system stack specific Media I/F
- `MediaIF.c` common Media I/F

If multiple File System stacks are to be used together, then `SysCall_MultiFS.c` needs to be omcluded and all the related `SysCall_`*`STACK`*`.c` & `Media_`*`STACK`*`.c` files. A small number of build options specific to the System Call layer must de defined, refer to [R4] for all details.

[1] When Abassi is mentioned in this document, unless explicitly stated, it always means Abassi, mAbassi and µAbassi.

2 Model

The common lower layer of the Media I/F is the same across all target platforms and compilers as Abassi driver's API is kept the same across all target platforms and compilers. There are a lot of build options (Section 3) because the Media I/F has been created to be as versatile as possible. All the build options specified here apply to the `MediaIF.c` file and not the file system specific layer file (i.e. `Media_`*`STACK`*`.c`). The file system stack specific Media I/F files rely on the file system stack definitions when required. Most of the build options are used to map the physical media storage devices to a device number (this is what is called the drive number and it's the number used when mounting / accessing the storage device through the application). Drive numbers in an application starts at 0 and increment in a contiguous manner; no drive number can be skipped.

The media interface common layer supports these devices:

- SD/MMC: up to 2 physical devices
- QSPI: up to 4 physical devices
- Memory drive: only 1

The build options used to map the physical devices to the drive number used by the application have these suffixes:

`_IDX`: build options with the suffix `_IDX` defines the drive number

`_DEV`: build options with the suffix `_DEV` specify the driver device #, i.e. the number used in the drivers to identify the controller number.

`_SLV`: build options with the suffix `_SLV` are only used for QSPI devices and they specify the slave number (chip select line) on which the QSPI chip is connected to the QSPi controller. This is the number used by the QSPI driver.

The simplest way to use the Media I/F is to not specify any `MEDIA_????` build options and let it map the drives according to the available media devices on the target board. The information used to perform this mapping is extracted from the file `Platform/inc/Platform.h` and it relies on the definitions of `SDMMC_DEV`, `QSPI_DEV` and `QSPI_SLV`. The application drive numbers are assigned from 0 and up in this order: 1st SD/MMC, 2nd QSPI. So when the device mapping is automatic, up to 2 mass storage devices can be automatically supported; it could be less as not all demo boards have SD/MMC and/or QSPI media. Other physical devices can be added over the automatically selected ones as long as the device numbers (`_IDX` values, see next paragraph) don't equate or exceed the total number of devices, If any of the specified indexes are 0 or 1, the automatic mapping always skips the assigned indexes done with the definition of the `MEDIA_?????_IDX` build options.

If the automatic mapping is not desired, then the build option `MEDIA_AUTO_SELECT` must be defined and set to a value of 0. The mass storage devices are defined with pairs of build options with the suffix `_IDX` and `_DEV` (plus `_SLV` for QSPI). For example, if there are 3 QSPI devices (devices:slaves - 0:0, 0:1, 1:0) on the target platform, 2 SD/MMC (devices - 0, 1) and a memory drive are to be used, then one mapping could be for example:

- Drive #0 SD/MMC controller #1:

    `MEDIA_SDMMC0_IDX = 0` - SD/MMC mapped to drive #0

    `MEDIA_SDMMC0_DEV = 1` - SD/MMC controller #1 mapped to drive #0

- Drive #1 QSPI controller #0 / Slave #1:

    `MEDIA_QSPI0_IDX = 1` - QSPI mapped to drive #1

    `MEDIA_QSPI0_DEV = 0` - QSPI controller #0 mapped to drive #1

    `MEDIA_QSPI0_SLV = 1` - QSPI controller #0 / Slave #1 mapped to drive #1

- Drive #2 QSPI controller #0 / Slave #0:

`MEDIA_QSPI1_IDX = 2` - QSPI mapped to drive #2

`MEDIA_QSPI1_DEV = 0` - QSPI controller #0 mapped to drive #2

`MEDIA_QSPI1_SLV = 0` - QSPI controller #0 / Slave #0 mapped to drive #2

- Drive #3 QSPI controller #1 / Slave #0:

`MEDIA_QSPI2_IDX = 3` - QSPI mapped to drive #3

`MEDIA_QSPI2_DEV = 1` - QSPI controller #1 mapped to drive #3

`MEDIA_QSPI2_SLV = 0` - QSPI controller #1 / Slave #0 mapped to drive #3

- Drive #4 SD/MMC controller #0:

`MEDIA_SDMMC1_IDX = 4` - SD/MMC mapped to drive #4

`MEDIA_SDMMC1_DEV = 0` - SD/MMC controller #0 mapped to drive #4

- Drive #5 Memory Drive:

`MEDIA_MDRV_IDX   = 5` - Memory drive mapped to drive #5

`MEDIA_MDRV_SIZE = 0` - Size & base address of the drive is provided by the linker

These restrictions must be followed when mapping the physical media devices to the application device numbers:

1- no two build options with the suffix `_IDX` can be assigned the same numerical value. This would map 2 different physical media on the same drive number

2 – for each build option with the suffix `_IDX` defined, there must be a corresponding `_DEV` (and for QSPI media also `_SLV`) build option defined.

3 - If a number of ***N*** build options with the suffix `_IDX` are defined, then the values assigned to these build options must be within the range of 0 to ***N***-1, no "holes" are permissible.

Error messages during compile time are issued if any of the restrictions are not respected. If a `_DEV` and/or `_SLV` is defined with no corresponding `_IDX`, then these `_DEV` / `_SLV` definitions are ignored. If a build option `_IDX` or `_DEV` or `_SLV` is defined and assigned a negative value, it is the same as if it hasn't been defined.

The name of these build options is always named using the following consturct, excluding the suffix:

`MEDIA __ STORAGE_TYPE  NUMBER __`

`STORAGE_TYPE` is `SDMMC`, `QSPI`, or `MDRV`.

`NUMBER` is a unique identifier with no relationship to the controller number (device number used by the Abassi's drivers) or the drive number it is mapped to; `A B C` could have been used… but numbers it is. It's not necessary to start using `NUMBER` at zero (0) nor use contiguous values for `NUMBER`.

3 Build Options

The build options supported by the Media I/F are shown in the following table:

Table 3-1 Build Options

Token Name	Default	Description
MEDIA_AUTO_SELECT	!=0 (enable)	Controls if the mapping between device and drive is done automatically or not
MEDIA_SDMMC_SECT_SZ	512	Declares a different sector size than the real physical sector size for all SD/MMCs
MEDIA_SDMMC0_IDX	undefined	Drive number for a SD/MMC identified as 0 (0 has no relationship to the drive, device or controller numbers)
MEDIA_SDMMC0_DEV	undefined	SD/MMC controller number (Abassi's driver device number) to map to the drive number specified by MEDIA_SDMMC0_IDX
MEDIA_SDMMC0_SECT_SZ	MEDIA_SDMMC_SECT_SZ	Declares a different sector size than the physical sector size for the SD/MMC card mapped to the drive number specified by MEDIA_SDMMC0_IDX and MEDIA_SDMMC0_DEV
MEDIA_SDMMC0_FIRST	0	First block of 512 bytes to use on the SD/MMC card mapped to the drive number specified by MEDIA_SDMMC0_IDX & MEDIA_SDMMC0_DEV
MEDIA_SDMMC0_SIZE	undefined	Size in multiple of 512 bytes to use on the SD/MMC card mapped to the drive number specified by MEDIA_SDMMC0_IDX & MEDIA_SDMMC0_DEV
MEDIA_SDMMC1_IDX	undefined	Drive number for a SD/MMC identified as 1 (1 has no relationship to the drive, device or controller numbers)
MEDIA_SDMMC1_DEV	undefined	SD/MMC controller number (Abassi's driver device number) to map to the drive number specified by MEDIA_SDMMC1_IDX
MEDIA_SDMMC1_SECT_SZ	MEDIA_SDMMC_SECT_SZ	Declares a different sector size than the physical sector size for the SD/MMC card mapped to the drive number specified by MEDIA_SDMMC1_IDX & MEDIA_SDMMC1_DEV
MEDIA_SDMMC1_FIRST	0	First block of 512 bytes to use on the SD/MMC card mapped to the drive number specified by MEDIA_SDMMC1_IDX & MEDIA_SDMMC1_DEV
MEDIA_SDMMC1_SIZE	undefined	Size in multiple of 512 bytes to use on the SD/MMC card mapped to the drive number specified by MEDIA_SDMMC1_IDX & MEDIA_SDMMC1_DEV
MEDIA_QSPI_SECT_SZ	512	Declares a different sector (minimum erase size) than the real ones for all QSPI devices
MEDIA_QSPI0_IDX	undefined	Drive number for a QSPI identified as 0 (0

		has no relationship to the drive, device, slave or controller numbers)
MEDIA_QSPI0_DEV	undefined	QSPI controller number (Abassi's driver device number) to map to the drive number specified by MEDIA_QSPI0_IDX
MEDIA_QSPI0_SLV	undefined	QSPI slave number (chip select) to map to the drive number specified by MEDIA_QSPI0_IDX
MEDIA_QSPI0_SECT_SZ	MEDIA_QSPI_SECT_SZ	Declares a different sector size than the smallest erase size of the QSPI chip mapped to the drive number specified by MEDIA_QSPI0_IDX & MEDIA_QSPI0_DEV & MEDIA_QSPI0_SLV
MEDIA_QSPI0_FIRST	0	First block of 512 bytes to use on the QSPI chip mapped to the drive number specified by MEDIA_QSPI0_IDX & MEDIA_QSPI0_DEV & MEDIA_QSPI0_SLV
MEDIA_QSPI0_SIZE	undefined	Size in multiple of 512 bytes to use on the QSPI chip mapped to the drive number specified by MEDIA_QSPI0_IDX & MEDIA_QSPI0_DEV & MEDIA_QSPI0_SLV
MEDIA_QSPI1_IDX	undefined	Drive number for a QSPI identified as 1 (1 has no relationship to the driver, device, slave or controller numbers)
MEDIA_QSPI1_DEV	undefined	QSPI controller number (Abassi's driver device number) to map to the drive number specified by MEDIA_QSPI1_IDX
MEDIA_QSPI1_SLV	undefined	QSPI slave number (chip select) to map to the drive number specified by MEDIA_QSPI1_IDX
MEDIA_QSPI1_SECT_SZ	MEDIA_QSPI_SECT_SZ	Declares a different sector size than the smallest erase size of the QSPI chip mapped to the drive number specified by MEDIA_QSPI1_IDX & MEDIA_QSPI1_DEV & MEDIA_QSPI1_SLV
MEDIA_QSPI1_FIRST	0	First block of 512 bytes to use on the QSPI chip mapped to the drive number specified by MEDIA_QSPI1_IDX & MEDIA_QSPI1_DEV & MEDIA_QSPI1_SLV
MEDIA_QSPI1_SIZE	undefined	Size in multiple of 512 bytes to use on the QSPI chip mapped to the drive number specified by MEDIA_QSPI1_IDX & MEDIA_QSPI1_DEV & MEDIA_QSPI1_SLV
MEDIA_QSPI2_IDX	undefined	Drive number for a QSPI identified as 2 (2 has no relationship to the driver, device, slave or controller numbers)
MEDIA_QSPI2_DEV	undefined	QSPI controller number (Abassi's driver device number) to map to the drive number specified by MEDIA_QSPI2_IDX
MEDIA_QSPI2_SLV	undefined	QSPI slave number (chip select) to map to the

		drive number specified by MEDIA_QSPI2_IDX
MEDIA_QSPI2_SECT_SZ	MEDIA_QSPI_SECT_SZ	Declares a different sector size than the smallest erase size of the QSPI chip mapped to the drive number specified by MEDIA_QSPI2_IDX & MEDIA_QSPI2_DEV & MEDIA_QSPI2_SLV
MEDIA_QSPI2_FIRST	0	First block of 512 bytes to use on the QSPI chip mapped to the drive number specified by MEDIA_QSPI2_IDX & MEDIA_QSPI2_DEV & MEDIA_QSPI2_SLV
MEDIA_QSPI2_SIZE	undefined	Size in multiple of 512 bytes to use on the QSPI chip mapped to the drive number specified by MEDIA_QSPI2_IDX & MEDIA_QSPI2_DEV & MEDIA_QSPI2_SLV
MEDIA_QSPI3_IDX	undefined	Drive number for a QSPI identified as 3 (3 has no relationship to the driver, device, slave or controller numbers)
MEDIA_QSPI3_DEV	undefined	QSPI controller number (Abassi's driver device number) to map to the drive number specified by MEDIA_QSPI3_IDX
MEDIA_QSPI3_SLV	undefined	QSPI slave number (chip select) to map to the drive number specified by MEDIA_QSPI3_IDX
MEDIA_QSPI3_SECT_SZ	MEDIA_QSPI_SECT_SZ	Declares a different sector size than the smallest erase size of the QSPI chip mapped to the drive number specified by MEDIA_QSPI3_IDX & MEDIA_QSPI3_DEV & MEDIA_QSPI3_SLV
MEDIA_QSPI3_FIRST	0	First block of 512 bytes to use on the QSPI chip mapped to the drive number specified by MEDIA_QSPI3_IDX & MEDIA_QSPI3_DEV & MEDIA_QSPI3_SLV
MEDIA_QSPI3_SIZE	undefined	Size in multiple of 512 bytes to use on the QSPI chip mapped to the drive number specified by MEDIA_QSPI3_IDX & MEDIA_QSPI3_DEV & MEDIA_QSPI3_SLV
MEDIA_QSPI_SECT_BUF	65536	Size of temporary buffers used with QSPI chips
MEDIA_QSPI_OPT_WRT	1 (enable)	Controls if QSPI erasures /write are minimized when possible. Applies to all QSPI drives
MEDIA_QSPI_CHK_WRT	0 (disable)	Controls if QSPI are read back after writing and how many time to retry upon mismatch. Applies to all QSPI drives
MEDIA_MDRV_IDX	undefined	Drive number for the memory drive
MEDIA_MDRV_SIZE	0	Select if the size of the memory drive is provided by the linker or reserved at compile time
MEDIA_ARG_CHECK	0	Boolean to enable / disable the checks on the

		validity of the API function arguments
`MEDIA_DEBUG`	0	Boolean controlling the sending of progress / debug messages to `stdout`.

Grouping is done in the following sub-sections to eliminate redundant descriptions. When the abbreviation `MEDIA_`*NNNN*`_...` is used, it means `MEDIA_SDMMC`, `MEDIA_QSPI`, or `MEDIA_MDRV`. When the abbreviation `MEDIA_`*NNNN#*`_...` is used, it means `MEDIA_SDMMC0`, `MEDIA_SDMMC1`, `MEDIA_QSPI0`, `MEDIA_QSPI1`, `MEDIA_QSPI2`, `MEDIA_QSPI3`, or `MEDIA_MDRV`.

3.1.1 MEDIA_AUTO_SELECT

The build option `MEDIA_AUTO_SELECT` informs the Media I/F if it assigns or not the media device numbers according to the information on the target platform SDMMC and QSPI media storage devices. This information is located in the file `Platform/inc/Platform.h`. When `MEDIA_AUTO_SELECT` is not defined, or is defined and set to a non-zero value, it selects the auti-mapping of the physical media based on the information in `Platform.h`.: this is the default value. If it is defined, and set to a value of 0, it does not use the information from `Platform.h`, and all media to access must be defined through the build options with prefixes `_IDX`, `_DEV` (and `_SLV` for QSPI).

In `Platform.h` if `SDMMC_DEV` is defined for the target platform, the specified controller (Abassi's driver device number) will be mapped to drive #0. If it is not defined, then no SD/MMC is mapped by default. If `QSPI_DEV` and `QSPI_SLV` are defined for the target platform, then depending if a SD/MMC controller is mapped by default or not, the QSPI controller number (Abassi's driver device number) / slave number (Abassi's driver slave number) will either be mapped to drive #0 (no SD/MMC controller) or to drive #1 (SD/MMC controller present). If one or more `MEDIA_`*NNNN#*`_IDX` build options are defined in the application, these drive numbers are skipped by the automatic mapping as they take precedence, e.g. if the application defines `MEDIA_QSPI3_IDX` and sets its value to 0, the automatic mapping will start at 1, and not 0, when mapping the devices defined in `Platform.h`.

3.1.2 MEDIA_*NNNN*#_IDX

The build options `MEDIA_`*NNNN#*`_IDX` are use to associate a drive number with all the related `MEDIA_NNNN#` build options. For example if `MEDIA_SDMMC1_IDX` is defined and set to 2 it will associate the following build options to drive number 2: `MEDIA_SDMMC1_DEV`, `MEDIA_SDMMC1_SECT_SZ`, `MEDIA_SDMMC1_FIRST`, and `MEDIA_SDMMC1_SIZE`. By default, none of the option `MEDIA_NNNN#_IDX` build options are defined.

3.1.3 MEDIA_*NNNN*#_DEV & MEDIA_QSPI#_SLV

The build options `MEDIA_`*NNNN#*`_DEV`, and for QSPI, `MEDIA_QSPI#_SLV`, are use to specify which media controller (Abassi's driver device number), and for QSPI the slave number are accessed through the drive number specified by `MEDIA_`*NNNN#*`_IDX.` For example if `MEDIA_QSPI2_DEV` is defined and set to 1, and `MEDIA_QSPI2_SLV` is defined and set to 2, then the QSPI chip attached to controller #2 / slave #0 is accessible as the drive number specified by the build option `MEDIA_QSPI2_IDX`.

By default, none of the option `MEDIA_NNNN#_DEV` and `MEDIA_NNNN#_SLV` build options are defined.

3.1.4 MEDIA_*NNNN*_SECT_SZ

Devices block sizes are highly variable across media. For example, SD/MMC can have block sizes of 512, 1024, 204, or 4096 bytes. QSPI block sizes, which are the smallest erasure size, can be between 128 to 256K bytes. Files systems typically rely on the device block size to select the sector sizes or when they are set-up for a fixed sector size. When the sector size needs to be different from the device block size, the build option `MEDIA_`*NNNN*`_SECT_SZ` can be used to overload the physical device block size. When defined and set to a positive value, all media block size reported to the file system stack is the specified value. The value assigned to `MEDIA_`*NNNN*`_SECT_SZ` applies to all media devices of the *NNNN* category. Individual device sector size can be set with the `MEDIA_NNNN#_SECT_SZ` build option.

By default, `MEDIA_`*`NNNN`*`_SECT_SZ` is set to a value of 512 because it's a sector size supported by all file system stack.

3.1.5 MEDIA_*NNNN*#_SECT_SZ

By default, all media devices sector size in a category can be overloaded with the build option `MEDIA_`*`NNNN`*`_SECT_SZ`. It is possible to overload individual devices sector size with the use of the build option `MEDIA_`*`NNNN#`*`_SECT_SZ`. Everything described for `MEDIA_`*`NNNN`*`_SECT_SZ` applies, except it only applies to the specific drive indicated by the *`NNNN#`* in the build option. If both `MEDIA_`*`NNNN`*`_SECT_SZ` and `MEDIA_`*`NNNN#`*`_SECT_SZ` are defined and set to values greater than 0, then `MEDIA_`*`NNNN#`*`_SECT_SZ` takes precedence.

3.1.6 MEDIA_QSPI_SECT_BUF

QSPI sub-sector sizes (minimum erasure size) are commonly 4096 bytes. When the file system sector size is smaller than the QSPI sub-sector size, it becomes necessary, when writing a sector to a QSPI sub-sector, to read a full sub-sector and deposit the data in a temporary buffer, erase the sub-sector, insert the file system sector to write in the temporary buffer and write the updated temporary buffer to the QSPI chip. The build options `MEDIA_QSPI_SECT_BUF` set the size of the temporary buffers (there is one buffer per QSPI drives) and they are set by default to 65536 (64KB) as this seems to be the largest QSPI sub-sector. There shouldn't be any needs to change the value of this build option unless data memory is short, or a QSPI chip with a larger sub-sector than 64K (i.e. the very few parts with 256K sub-sectors) is used.

NOTE: It is not advisable to use QSPI parts with sub-sector size that are greater than 4K if the QSPi chip will go through many writes as there risks to be an excessive number of erase that will be performed and QSPI chips have a finite number of erasure cycle they be submitted to.

3.1.7 MEDIA_QSPI_OPT_WRT

QSPI chips wears out over time due the to erasures. The build option `MEDIA_QSPI_OPT_WRT` changes the way sector writing is performed for all QSPI devices. When this feature is enabled (the build option set to a non-zero value) the Media I/F analyzes the data in the data to write to see if it is possible to skip the erasure. When writing on a QSPI chip a 1 can be changed to a 0 without erasure. The sub-sector to write to is first read from the QSPI and the QSPI contents compared to the data to write. If only bit transitions from 1 to 0 are required, the erasure step is skipped. Also checked is to see if the data to write is identical to the QSPI data at the beginning and at the end of the sector. When there are continuously identical data at the beginning and/or at the end, the writing of these is skipped for the contiguous identical data.

For the QSPI optimized writing to be effective, QSPI chips should be completely erased before being formatted. The more files get written to the QSPI file system will over time reduce the efficiency of QSPI write optimization because less and less "virgin" sub-sectors remain. The QSPI optimize writing is enable by default. This build option applies to all QSPI devices.

3.1.8 MEDIA_QSPI_CHK_WRT

As a reliability mechanism, it is possible to check if the data written on the QSPI chips is correct. By default, the data held in the QSPI chip after a write is not counter-checked against the data that was written. To enable the checking of the data written, the build option `MEDIA_QSPI_CHK_WRT` has to be set to a positive value and the value specifies the maximum number of retries that can be performed when the data written in the chip does not match what it's supposed to be. Upon mismatch, a value of 1 will try writing again only once, a value of 2 will try it writing a maximum of 2 times. This build option applies to all QSPI devices.

3.1.9 MEDIA_*NNNN*#_FIRST & MEDIA_*NNNN*#_SIZE

It is possible to use part of a media device using the combination of the build options `MEDIA_`*`NNNN#`*`_FIRST` and `MEDIA_`*`NNNN#`*`_SIZE`. By default, none of these are defined and the whole area of the media device is accessed. When `MEDIA_`*`NNNN#`*`_FIRST` is defined and set to a positive value, it specifies the first address in the media device to use, any addresses below the value specified are left inaccessible and untouched. The value specified is in block of 512 bytes, therefore specifying value of 1000 leaves the lower 512000 bytes of the media device untouched. When `MEDIA_NNNN#_SIZE` is specified and set to a positive value, it redefines the total size of the media device. `MEDIA_NNNN#_SIZE` value is also the number of 512 bytes blocks.

It is not possible at build time to determine if the resulting size using of the build options exceeds the size of the media device itslef. During run time, if `MEDIA_NNNN#_FIRST` refers to a memory address located past the size of the media device, the initialization of the media device will fail. In the case of `MEDIA_NNNN#_SIZE`, if the resulting upper address is higher than the real upper address of the media device, the size used is the available area (the size is `Media_Size-512*MEDIA_NNNN#_FIRST`) and these is no report (except in the debug info when `MEDIA_DEBUG` is >1). The resulting upper address specified by the 2 build options is `512*(MEDIA_NNNN#_FIRST+MEDIA_NNNN#_SIZE)`.

There are multiples reasons why blocks of 512 bytes was chosen as the unit for the build options `MEDIA_NNNN#_FIRST` and `MEDIA_NNNN#_SIZE`. First, the minimum block size / sector size supported by all file systems is 512 bytes so using 512 bytes makes sense. Most QSPI chips sub-sector erase sizes are 4096 bytes therefore using 512 bytes guarantees data alignment on QSPI sub-sector boundaries. SD/MMC size exceed 4 GB, so it would have been imperative to add `LL` in the build option values to make sure the pre-processor value can exceed 2^32-1. (LL is easy to forget and could lead to unknowingly corrupt area in the media device that should have been left untouched.

It is not necessary to define both build options if the area to leave untouched is the lower part of the memory or the upper area of the memory. To protect the low memory from addresses 0 to `512*MEDIA_NNNN#_FIRST`, it's only necessary to specify `MEDIA_NNNN#_FIRST` as the Media I/F will access all the remainder of the media device memory. To protect the upper memory from addresses `512*MEDIA_NNNN#_SIZE` to the end of the memory, it's only necessary to specify `MEDIA_NNNN#_SIZE` as the Media I/F will only access the memory from 0 to `512*MEDIA_NNNN#_SIZE-1`.

These build options don't apply to the Memory-Drive; `MEDIA_MDRV_SIZE` has its own definition.

3.1.10 MEDIA_MDRV_SIZE

The Media I/F supports a memory drive; i.e. a drive located in the memory space of processor. When `MEDIA_MDRV_IDX` is defined and set to a non-negative value, then `MEDIA_MDRV_SIZE` must be defined defined set to a non-zero value. If `MEDIA_MDRV_SIZE` is positive, the value it is set to defines the size of the memory reserved during compilation, i.e. it defines the size (number of bytes) of an array of type `uint8_t`. When it is set to a negative value, it informs the Media I/F the memory drive memory is allocated during link time. In the linker, the base address of the memory must be the symbol `G_MemDrvBase` and the symbol `G_MemDrvEnd` must be the location immediately after the last memory reserved for the memory drive (the size of the memory drive is then `G_MemDrvEnd – G_MemDrvBase`). For more details, refer to any demo linker script files that are provided as they all have support memory drive.

If `MEDIA_MDRV_IDX` is not defined or is defined with a negative value then `MEDIA_MDRV_SIZE` is ignored and no memory drive is used.

3.1.11 MEDIA_ARG_CHECK

The build options `MEDIA_ARG_CHECK` controls if the driver checks the validity of the API function arguments or not. This build option is a Boolean; when set to a non-zero value, the driver checks the validity of the arguments and returns an error code when the arguments are invalid. When set to a zero value, it does not check the validity of the arguments.

3.1.12 MEDIA_DEBUG

The build options `MEDIA_DEBUG` controls the printout of progress and error messages to `stdout`. This build option can have three set-ups; when set to a value of zero or less, no messages are sent to `stdout`. When set 1, it sends over `stdout` the set-up information used during initialization and causes of error during the operation. When set to a value greater than 1, it prints on `stdout` all operations and causes of errors.

4 Files

This section provides a list of the the files to be included in the application when using the Media I/F with or without the system call layer. It also indicates some key build options to set.

The Media I/F relies on Abassi's drivers to read and write SD/MMC and QSPI media. Therefore, if SD/MMC is to be access, the file `???_sdmmc.c` has to be included in the build and / or if QSPI is to be accessed then the file `???_qspi.c` has to be included in build.

Demos #20 to #29 can be looked at for more details on how to use the system call layer with one or multiple file system stacks.

In the next subsections, *`FSSTACK`* is to be replaced by `FatFS`, `FullFAT` and/or `ueFAT`. , and COMPILER with `ARMCC`, `CCS` and/or `IAR`.

4.1 Media I/F alone

To use a file system stack directly, the file system stack and the Media I/F files are required, so the files that from the Media I/F must be included in the build

- `Drivers/src/MefiaIF.c`	Common media API
- `Driver/src/Media_`*`FSSTACK`*`.c`	File system stack specific API (*`FSSTACK`* is the name of the file system stack).
- `Abassi/Abassi_`*`FSSTACK`*`.c`	File System stack specific API for exclusive access protection. The protection is provided through Abassi's mutex.
- `Platform/src/???_SysCall.c`	Access to the real-time clock on the target platform. It is target platform specific and the file provided in `Platform/src` only supports the target platform used in the demos. Is needed if the file system stack is set-up for time stamping new files.

4.2 System Call Layer & Media I/F

To have access to the media through the standard "C" system calls, e.g. `fopen()`, `fprintf()`, `fclose()`, the system call layer needs to be added on top of the Media I/F alone set of files. Reference [R4] provides all the details on the System Call Layer. The files for the Media I/F standalone (Section 4.1) and the followings need to be added in the build:

- `Abassi/SysCall_`*`FSSTACK`*`.c`	File System stack specific system call layer code. This file is the layer between the standard UINX system call and the File System Tack API.
- `Abassi/SysCall_`*`COMPILER`*`.c`	Non-GCC compilers require a complementary file to interface between the compiler system call API and UNIX system call API.
- `Abassi/SysCall_MultiFS.c`	Only needed when multiple File System are used in the same application.

The System Call Layer has a few build options described in [R4]. At minimum, the build option `OS_SYS_CALL` must be defined and set to a non-zero value

4.3 Fat-FS

The files needed by Fat-FS are the following *(###* is the version of the file system stack) and they all need to be included in the build to use the Fat-FS file system stack:

`Share/inc/ffconf.h`	Fat-FS configuration file used by the demos

`FatFS-###/inc/`	Path for the include files used by FatFS.
`FatFS-###/src/ff.c`	
`FatFS-###/src/ffunicode.c`	

If re-using the configuration file (`Share/inc/ffconf.h`) provided with the demos, there are 3 key build options to set with Fat-FS, namely `FF_VOLUMES` (previously `_VOLUMES`), `FF_FS_LOCK` (previously `_FSLOCK`), and `FF_FS_NORTC` (previously `_FS_NORTC`). The build option `FF_VOLUMES` must be set to a value greater or equal to the maximum number of the media devices that are supported in the application. `FF_FS_NORTC` when set to zero, enables Fat-FS to use an on-board RTC for time stamping files. The file `Platform/src/SysCall_???.c` likely needs to be customized for the target board. For `FF_FS_LOCK`, this build options defines the maximum number of files and directories that can be opened at the same time.

4.4 FullFAT

The files needed by Full-FAT are the following *(###* is the version of the file system stack) and they all need to be included in the build to use the Full-FAT file system stack:

`Share/inc/ff_conf.h`	Full FAT configuration file used by the demos
`FullFAT-###/src/`	Path for the include files used by Full-FAT it is `src`, not `inc`)
`FullFAT-###/src/ff_blk.c`	
`FullFat-###/src/ff_crc.c`	
`FullFat-###/src/ff_dir.c`	
`FullFat-###/src/ff_error.c`	
`FullFat-###/src/ff_fat.c`	
`FullFat-###/src/ff_file.c`	
`FullFat-###/src/ff_format.c`	
`FullFat-###/src/ff_hash.c`	
`FullFat-###/src/ff_ioman.c`	
`FullFat-###/src/ff_memory.c`	
`FullFat-###/src/ff_string.c`	

If re-using the configuration file (`Share/inc/ff_conf.h`) provided with the demos, there are a single key build options to set with Full-Fat, namely `FF_TIME_SUPPORT`. When it is not defined it informs Full FaAT to not rely in a RTC for time stamping files. When defined, it enables Full-FAT to use an on-board RTC for time stamping files. The file `Platform/src/SysCall_???.c` likely needs to be customized for the target board.

NOTE: Full-FAT has a reported issue and it cannot format media drives. Another File System stack must be used if the application needs to perform the formatting of media devices.

4.5 ueFAT

The files needed by FatFS are the following *(###* is the version of the file system stack) and they all need to be included in the build to use the FatFS file system stack:

`Share/inc/fat_opts.h`	Ultra Embedded FAT configuration file used by the demos
`ueFAT-###/`	Path for the include files used by ueFAT
`ueFAT-###/fat_access.c`	
`ueFAT-###/fat_cache.c`	
`ueFAT-###/fat_filelib.c`	
`ueFAT-###/fat_format.c`	

`ueFAT-###/fat_misc.c`

`ueFAT-###/fat_string.c`

`ueFAT-###/fat_table.c`

`ueFAT-###/fat_write.c`

If re-using the configuration file (`Share/inc/fat_opts.h`) provided with the demos, there are a single key build options to set with ueFAT, namely `FATFS_INC_TIME_DATE_SUPPORT`. When defined and set to a non-zero value it enables ueFAT to use an on-board RTC for time stamping files. The file `Platform/src/SysCall_???.c` likely needs to be customized for the target board.

NOTE: ueFAT can only access a single drive. To access multiple drives it is necessary to go through un-mount and mount. Also, ueFAT does not use the FAT info for the number of bytes per sectors, instead it operates with the sector size defined through the build option `FAT_SECTOR_SIZE`.

5 References

[R1] Abassi RTOS – User Guide, available at http://www.code-time.com
[R2] mAbassi RTOS – User Guide, available at http://www.code-time.com
[R3] µAbassi RTOS – User Guide, available at http://www.code-time.com
[R4] Abassi – System Call Layer, available at http://www.code-time.com