CODE TIME TECHNOLOGIES

Abassi RTOS

System Calls Layer

Disclaimer

Code Time Technologies Inc. provides this document “AS IS” without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.
Code Time Technologies Inc. does not warrant that the contents of this document will meet your requirements or that the document is error-free. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the document. Code Time Technologies Inc. may make improvements and/or changes in the product(s) and/or program(s) described in the document at any time. This document does not imply a commitment by Code Time Technologies Inc. to supply or make generally available the product(s) described herein.

Table of Contents

List of Figures

List of Tables

1 Introduction

This document describes the System Calls layer provided for Abassi[1] [R1] (including mAbassi [R2] and µAbassi [R3]). The System Calls layer is the way Code Time Technologies provides a solution to support in an easy manner the standard "C" library system calls, and it encapsulates at the same time the board support package (BSP). It is a plug-and-play set of files, where the standard "C" library input and output (I/O) becomes fully supported, irrelevant of the target platform. With it, all stdio (stdin, stdout, stderr) exchanges are done on the selected platform UART. When one or more mass storage interfaces are available, then it becomes possible to use functions like `fopen()`, `fprintf()`, `fseek()`. In supplement, the System Calls layer is designed to provided most of the UNIX system calls, like `open()`, `read()`, `dup()`. The System Calls layer also supports virtual devices that can be accessed through a `/dev/xxx` file naming, making the BSP fully transparent to the user. It also supports the use of multiple file system stacks when more than one file system format is needed. This means an application can be created that will, for example, handle at run time NTFS, FAT32, and exFAT, relying on different File System stacks.

1.1 Distribution Contents

There are many files supplied with the distribution for the purpose of implementing the System Calls layer. This is mainly due to the number of file system stacks and "C" libraries supported. The general file naming and use are listed in Table 1-1 below:

Table 1-1 Distribution

File Name	Description
`SysCall.h`	Include file for the System Calls.
`SysCall_<FS>.c`	"C" file implementing the System Calls layer for the file system stack *<FS>*. In the file name, *<FS>* is the short-name of the specific file system library. NOTE: Except for Newlib (GCC), two files are always required: `SysCall_<LIB>.c` and `SysCall_<FS>.c`. Newlib (GCC) is an exception as it only requires `SysCall_<FS>.c`.
`SysCall_<LIB>.c`	"C" file implementing the System Calls layer for the "C" library *<LIB>*. In the file name, *<LIB>* is the short-name of the specific "C" library. NOTE: Except for Newlib (GCC) and ARM CC µLib, two files are always required: `SysCall_<LIB>.c` and `SysCall_<FS>.c`. Newlib (GCC) is an exception as it only requires `SysCall_<FS>.c`; ARM CC µLib only requires `SysCall_uLib.c`.
`Abassi_<FS>.c`	"C" file providing mutex, semaphore and other RTOS facility required by the file system stack *<FS>*. In the file name, *<FS>* is the short-name of the specific file system stack library. This file is independent from the target "C" or from the target hardware platform.

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

Media_<*FS*>_<*HW*>.c	"C" file for the mass storage access. In the file name, <*FS*> is the short-name of the specific file system stack and <*HW*> is the target platform. This file is independent from the target "C" library but is specific to the file system stack to use AND the target hardware platform.
SysCall_<*HW*>.c	"C" file for platform-specific System Calls layer. This is currently used to provide access to the on-board RTC for the local date/time.
<*FS*>-vvv	Directory holding all files for the file system stack with the short-name <*FS*>, with vvv being the version of the file system stack release.
<*CFG*>.h	Configuration file for the selected File System Stack. Each file system stack uses a different name that has no real relationship with the file system stack name. e.g FatFS uses ffconf.h and FullFAT uses ff_config.h. File System stacks specific definition file name can be found by looking into the directory of the File System stack itself and an include file whose name starts with "_". This is the standard way Code Time "removes" the standard include file without modifying the code.

NOTE: When using the System Calls layer with the IAR CLIB and the file SysCall_<HW>.c is required to access the on-board time of the day, it is necessary to define the following for the whole project: _NO_DEFINITIONS_IN_HEADER_FILES=0 otherwise the time() function used will be the one from the library and an infinite recursive call will occur.

At the time of writing this document, some of the naming for <FS> are as follows:

Table 1-2 File System short-names

Short-Name	Description
noFS	No file system, only stdio (stdin, stdout, stderr) is handled.
ctFAT	Code Time FAT12/ FAT16 / FAT32 file system
FatFS	FatFS - FAT12 / FAT16 / FAT32 and exFAT file system ELM by ChanN [R4]
FullFAT	FullFAT FAT12 / FAT16 / FAT32 file system [R5]
ueFAT	Ultra-Embedded FAT12 / FAT16 / FAT32 file system [R6]

For the libraries <LIB>:

Table 1-3 "C" libraries short-names

Short-Name	"C" Library
ARMCC	ARM CC (Keil) full library
uLib	ARM CC (Keil) µLib library
CCS	TI's Code Composer "C" library.
IAR	IAR CLIB library.

For the platform short-naming, as there are many of them, please look into the port specific `port/src/` directory. The following table shows where the specific files are located in the distribution:

Table 1-4 Directory structure

```
./ --- / --- Abassi / SysCall.h
       |            |
       |            / Abassi_<FS>.c
       |            |
       |            / SysCall_<FS>.c
       |            |
       |            / SysCall_<LIB>.c
       |
       / --- PORT / --- inc / <CFG>.h
                  |
                  / --- src / Media_<FS>_<HW>.c
                            |
                            / SysCall_<HW>.c
```

NOTE: All open-source code is provided as is, unmodified. The files may be located in a different place than where they were in the original ZIP/GIT/SVN, but their content is never modified. The only change Code Time may have performed is to prepend a "_" to the file name of one or more include files, as these are for the configuration, which is application dependent.

Also part of the distribution are the UART, I2C, SD/MMC, etc. drivers, which may be used, and are all supported by the System Calls layer. Refer to the specific driver documentation for further information.

1.2 Limitations

There is currently no support for the CMSIS RTOS API, nor is there support for a standalone version of the System Calls layer.

There are limitations, which are all related to the way the specific file system stacks operates. As such, instead of giving a long description of each one in this document, the limitations are described in the `SysCall_<FS>.c` files, at the beginning of the file. In addition, each function in the `SysCall_<FS>.c` files have further textual information on how they are implemented and what is non-standard.

1.3 Features

The System Calls layer is an add-on to the compiler "C" library. Abassi does not need to use it, it's optional. It provides all the hooks necessary to perform input and output (I/O) through the native API of the "C" library. As such, all access to `stdin`, `stdout`, `stderr` is performed through the configured UART(s) of the platform. When the System Calls layer file is selected to support a file system stack, then all mass storage I/O operations of the "C" library become fully supported. The System Calls layer is intrinsically built to provide the I/O UNIX system calls, so this ubiquitous API also becomes fully accessible.

In addition to the mass storage I/O handling, the System Calls layer supports virtual non-mass storage devices such as TTY, I2C, etc. These are accessible exactly the way UNIX devices are accessed, by specifying filenames like `/dev/tty1` or `/dev/i2c4`. They can be configured, opened, read from, written to, closed, stat'ed, etc.

Another feature is related to how the mass storage devices are accessed: there is no reference to the physical location of the device. All mass-storage devices are accessed through mount points, therefore only when a device is mounted is the physical information required. For example, consider a system with 2 physical drives, where drive #0 is a SD/MMC and drive #1 is a QSPI flash memory. The mounting operation could, for example, mount drive #0 on `/` (the root directory) or `/logging`, and drive #1 on `/apps`. The use of mount points eliminates any reference to the physical interfaces in the file naming. Only when performing the mounting operation does the physical interface matter. One could see the benefit of using mount points to remove the information about the physical interface from the file names as delivering truly portable code. Refer to Section 2.3 for more explanation of the benefits of mount points.

2 Examples and How-To

2.1 Use

There is nothing special to do when the System Calls layer is part of an application. The layer API is exactly the same as the standard UNIX system calls and almost all standard "C" library I/O functions become fully supported and can be used as such. There are build options to define to override defaults setting; these are described in Section 3.3.

2.2 Build example

The first example considers using mAbassi with FatFS on Xilinx's Zynq using SD/MMC as the mass storage access point. Xilinx's SDK is using GCC, therefore there is no need to add a library specific file (`SysCall_<LIB>.c`, indicated in Table 1-1). The following table shows all the files and their location that are required to build a basic application:

Table 2-1 Zynq file example

```
./ --- / --- Abassi / mAbassi.c
       |             |
       |             / mAbassi.h
       |             |
       |             / ARMv7_SMP_L1_L2_GCC.s
       |             |
       |             / SysCall.h
       |             |
       |             / Abassi_FatFS.c
       |             |
       |             / SysCall_FatFS.c
       |
       / --- mAbassi_SMP_CortexA9_XSDK / … / --- inc / ffconf.h
       |                                   |         |
       |                                   |         / xlx_sdmmc.h
       |                                   |         |
       |                                   |         / xlx_uart.h
       |                                   |
       |                                   / --- src / Media_FatFS_xlx.c
       |                                             |
       |                                             / SysCall_xlx.c
       |                                             |
       |                                             / mAbassi_SMP_CORTEXA9_GCC.s
       |                                             |
       |                                             / xlx_sdmmc.c
       |                                             |
       |                                             / xlx_uart.c
       |
       |
       / --- FatFS-0.12 / --- src / ff.c
                          |
                          / --- inc / ff.h
```

The file names in bold are the files required for the System Calls layer. All other files are the standard ones needed by mAbassi or the standard files related to FatFS and UART.

The second example considers using mAbassi with FatFS on Altera's Cyclone V using SD/MMC and QSPI flash memory as the mass storage access points. Altera's Cyclone V is normally developed on ARM/Keil DS-5, which offers its own compiler, ARM "CC". As the "C" library is not GCC, it is necessary to add the library file `SysCall_ARMCC.c` in the build. The following table shows all the files and their location that are required to build a basic application:

Table 2-2 Cyclone V file example

```
./ --- / --- Abassi / mAbassi.c
       |            |
       |            / mAbassi.h
       |            |
       |            / ARMv7_SMP_L1_L2_GCC.s
       |            |
       |            / SysCall.h
       |            |
       |            / SysCall_ARMCC.c
       |            |
       |            / SysCall_FatFS.c
       |
       / --- mAbassi_SMP_CortexA9_DS5 / --- inc / dw_qspi.c
       |                                |         |
       |                                |         / dw_sdmmc.h
       |                                |         |
       |                                |         / dw_uart.h
       |                                |         |
       |                                |         / ffconf.h
       |                                |
       |                                / --- src / dw_qspi.c
       |                                          |
       |                                          / dw_sdmmc.c
       |                                          |
       |                                          / dw_uart.c
       |                                          |
       |                                          / Media_FatFS_CY5.c
       |                                          |
       |                                          / SysCall_CY5.c
       |                                          |
       |                                          / mAbassi_SMP_CORTEXA9_GCC.s
       |
       / --- FatFS-0.12 / --- src / ff.c
                          |
                          / --- inc / ff.h
```

The file names in bold are the files required for the System Calls layer. All other files are the standard ones needed by mAbassi or the standard files related to FatFS and UART.

2.3 Mount Points

The use of mount points removes almost all references to the physical drive device. Only when mounting the file system on a drive is it necessary to use the physical drive number. Thus, mount points allow the creation of applications that are dealing with unified file systems. For example, let's assume an application that uses two SD/MMC devices. If one SD/MMC is used to hold a web server `.html` set of files on a FAT formatted card, and the other one is used to record video on a NTFS formatted card, then the mount points used can be alike `/html` and `/video`. Using physical drive numbers to differentiate both file systems would require either to force the insertion of these two card in specific slots or if the card can be inserted in any slot, then to propagate the drives numbers across the application. The mount points eliminate that, as the FAT card is always accessed at the root level through `/html` and the NTFS through `/video`.

2.4 /dev

Using the same philosophy as UNIX, the System Calls layer offers access to virtual devices, where the file name root is `/dev`. This means the application can use `open()` / `read()` / `write()` / `close()` to read from and write to non-file system devices. For example, a second UART (device #3), different from the one used for stdin, stdout, stderr can be use with `fprintf()` and `fscanf()`. All there is to do is to make sure the UART is initialized and then to use `fopen("/dev/tty3", "r+")`. One can see, as for the mount points, this decouples the application from the underlying device drivers. All virtual device naming are alike `/dev/<medium>#`, where *<medium>* is like `tty`, `i2c`, `spi`, etc. and # is one or two digits from 0 to 9 that specifies the physical device to access. When a device requires a slave number, e.g. SPI, then the number following <*medium*> must have 2 digits. The first digit is the device controller number and the second digit is the slave number.

2.4.1 Device initialization

Any device accessed through the virtual device `/dev` has to be initialized before being useable. There are two ways to perform the initialization. The obvious one is to directly use the device initialization API alike `i2c_init()`. The other one is to use the System Calls layer API to initialize devices. This is done through the function `devctl()`. The API of this function is:

```
int devctl(int fd, const int *Cfg);
```

The first argument is the file descriptor that was returned when opening the device, and the second is an array of `int` that must provide all the arguments of the device driver initialization function (excluding the device number which is derived from the virtual device name). Zero is returned when successful, and non-zero when an error occurred. Using the example of the UART driver:

Table 2-3 Direct initialization example

```
uart_init(3, Baudrate, Parity, StopBits, RxSize, TxSize, Filter);
```

The same initialization performed through the system call layer is as follows:

Table 2-4 devctl() initialization example

```
fd = open("/dev/tty3", "r+");

Cfg[0] = Baudrate;
Cfg[1] = Parity;
Cfg[2] = StopBits;
Cfg[3] = RxSize;
Cfg[4] = TxSize;
Cfg[5] = Filter;

Devctl(fd, &Cfg[0]);
```

2.4.2 /dev/tty

To enable the support of UART devices, the build option `SYS_CALL_DEV_TTY` must be defined and set to a non-zero value. Other than making sure the device to use has been initialized, there is nothing special for tty devices other than turning on the detection of the EOF character (see Section 3.3.7).

2.4.3 /dev/i2c

To enable the support of I2C devices, the build option `SYS_CALL_DEV_I2C` must be defined and set to a non-zero value. Device read and write are supported, but combined send and receive is not supported. As the use of an I2C bus involves specifying the address of the target device on the bus, it is necessary for the application to supply this information to the `read()` or `write()` system calls. The target I2C device address is passed to these 2 functions through the buffer used to collect the data read or the one that holds the data to send out. As the I2C bus can use either 7 or 10 bit addresses, the first two bytes of the buffer holds the target address. The byte located at index zero in a `char` buffer holds the MSByte of the address and the byte at index 1 holds the LSByte of the address. Both bytes must always be provided even if the number of address bits used on the I2C bus is 7. When writing to an I2C device, the argument to `write()` that indicates the number of bytes to write is the real number of data bytes sent on the I2C; the data transferred excludes the first 2 bytes. So a buffer used to write `#bytes` on an I2C bus must be dimension to at least `2+#bytes`. When reading an I2C bus, the two address bytes in the buffer are overwritten by the data read.

2.4.4 /dev/spi

To enable the support of SPI devices, the build option `SYS_CALL_DEV_SPI` must be defined and set to a non-zero value. Device read and write are supported, but combined send and receive is not supported. As the use of an SPI bus involves specifying the number of the target device on the bus, it is necessary for the device to be of the form `/dev/spiMN`, where `M` is the device controller number and `N` is the slave number of the bus.

2.5 stdio

"C" standard I/O, i.e. stdin, stdout and stderr, are natively supported by the System Calls layer. The physical UART devices to associate with each I/O are provided to the system call layer through the variables it imports (see Section 3.2). There is no standardization if the standard I/O is uni-directional or bi-directional. The system call layer supports them in a bi-directional way. The UART device(s) used for stdio is not initialized by the System Call Layer, this must be done before using any of the 3 stdio devices.

If an application only needs the standard I/O and no file system, the file `SysCall_noFS.c` should be used. Another case may be an application that has no need, nor support for UART accesses. The build for this type of application still requires to include the UART driver for the function called by the System Calls layer, but setting the UART driver build option `UART_LIST_DEVICE` to zero will provide "do-nothing" functions for the UART driver.

3 Target Set-up

To better understand the purpose and how to set the build options, a short description on how the System Calls layer is implemented will help. Basically, the System Calls layer uses 3 types of descriptors (or "C" data structures). One type is used to hold the information about the physical devices, another type is for each open file, and the third type is for each open directory. There is most likely the need for multiple instances of the same type of descriptor (maybe not for the physical device, if a single drive is used). The way the standard "C" library and how UNIX system calls work when accessing a resource is to basically perform an "open" operation to obtain a new resource, and then access it at will, and to perform a "close" when done. It is the responsibility of the underlying layer to handle all the management of these resources. This "open" / access / "close" management requires one descriptor for each open, and the minimum number of descriptor of one type corresponds to the maximum number of the same resource that can be open at the same time. The System Calls layer has a limit on how many drives can be mounted, how many files can be open at the same time, and how many directories can be open at the same time.

Another factor to consider is the fact the System Calls layer is targeted for multi-tasking applications, and as such, there can be simultaneous requests to the System Calls layer. This means exclusive access protection must be used, and that is provided with mutexes. Each of the System Call Descriptors needs protection under the circumstances described in the following 2 sub-sections.

3.1.1 Drive access protection

The drive accesses are limited to the `mount()` and `unmount()` function. These two functions are protected by a single mutex internal to the System Calls layer. It does not matter how many drives the System Calls layer is set to support, this internal mutex is always available, and how it is used does not need to be configured.

There is also a need to protect the File System stack's accesses to the drives themselves. It is quite standard for File System stacks to support this protection and it is done by File System stack configuration and not by the System Calsl layer configuration. This protection MUST be used, and if is not used an error message will be issued during the compilation. If a File System stack does not have the capability of using mutexes for protection, then the Code Time supplied Media interface driver will always incorporate mutex protection.

3.1.2 File & Directory access protection

There are two types of protection to consider in the case of the file and directory accesses. The first one is when an "open" is requested; this implies the System Calls layer has to look through the descriptors to find an unused one that will be provided to the application. The "open" operation is always protected by a single internal mutex, which is the same as the one used for the drive access protection. As explained in the previous sub-section, how this mutex is used does not require any configuration setting.

The second type of protection involves the resources that are already open. When a resource is accessed by a single task, there is no need for access protection. But if the resource can be accessed at the same time by multiple tasks, done when the file or directory descriptor is shared amongst them, then it becomes necessary to guarantee exclusive access. How this protection is done can be configured with the build option `SYS_CALL_MUTEX`.

3.2 Imported variables

There are three variables that must be provided to the System Calls layer, and they provide the information about the UART devices to use for stdin, stdout, stderr. The following table lists them:

Variable	Description
`int G_UartDevIn`	Specifies the UART device number for stdin.
`int G_UartDevOut`	Specifies the UART device number for stdout.
`int G_UartDevErr`	Specifies the UART device number for stderr.

The value each of these variables must be set to is the UART device numbers associated to the stdio. The UART device number is the first argument used in the function `uart_init()`, `uart_recv()`, and `uart_send()`.

3.3 Build Options

There are a few build options that can be used to change how the System Calls layer operates. The following table lists all of them:

Table 3-1 Build Options

File Name	Default	Description
`OS_SYS_CALL`	`Not defined`	When defined and non-zero, it informs Abassi to reserve room in the task descriptor to support the System Calls layer. If not defined, or defined as zero, a compilation error will be issued to report the problem. This option is not needed/used when the File System selected is `noFS`.
`SYS_CALL_MUTEX`	`0`	Indicates if mutexes are used for exclusive access protection and if so, how the mutexes are used. By default (0) the individual directory & file descriptors are protected through a single global and shared mutex.
`SYS_CALL_N_DRV`	`1`	Specifies the maximum number of physical devices handled by the System Calls layer. The default value can be derived from the File System stack configuration.
`SYS_CALL_N_FILE`	`5`	Specifies the maximum number of open files handled by the System Calls layer. The default value can be derived from the File System stack configuration.
`SYS_CALL_N_DIR`	`2`	Specifies the maximum number of open directories handled by the System Calls layer.
`SYS_CALL_DEV_@@@`	`0`	Specifies if the virtual `/dev/@@@` devices are handled by the System Calls layer.

SYS_CALL_TTY_EOF	0	When /dev/tty virtual devices are supported, this build option defines the end-of-file (EOF) character. It also specifies if this EOF character is used for stdio inputs, no matter if /dev/tty are supported or not.

3.3.1 OS_SYS_CALL

The build option OS_SYSTEM_CALL is used to inform Abassi to reserve room in the task descriptor. It must be defined and set to a non-zero value. If it is not defined, or defined and set to zero, an error will be issued during the compilation. This is an Abassi build option needed by the System Calls layer. For more information, refer to [R1]. It is not necessary to define this build option when the file SysCall_noFS.c is the file used for the System Call Layer.

3.3.2 SYS_CALL_MUTEX

The build option SYS_CALL_MUTEX sets-up the way the System Calls layer protects the files and directory accesses in a multi-task environment. The three possibilities are listed in the following table:

Table 3-2 SYS_CALL_MUTEX settings

SYS_CALL_MUTEX	Conditions
== 0	Use this setting when limited data memory is available or when the file and directory descriptors are not shared between tasks. The "C" library access protection is irrelevant in this case.
> 0	The file and/or directory descriptors are shared between tasks. The "C" library access protection is irrelevant in this case.
< 0	The File System stack protects the accesses to the individual physical drives through its own mutexes. It also requires the "C" library to protect the access to the individual files if the application shares open file / directory descriptors between tasks. Using this setting, the access protection for the virtual devices (/dev/%%%) is not provided by the System Calls layer, therefore it must be provided by the device drivers themselves.

When SYS_CALL_MUTEX is negative, a single internal mutex is used to protect the mounting and un-mounting operations, all "open", and also all directory accesses. When it is set to 0, an internal mutex is used to protect the mounting and un-mounting operation, all "open", and also all directory accesses, plus the files descriptor accesses are protected by a different internal mutex, and the directory accesses are protected by a third internal mutex. Finally, when SYS_CALL_MUTEX is positive, instead of the file descriptors sharing a mutex and the directory descriptor sharing another one, each file descriptor and directory descriptor has their own unique mutex.

A few conditions are required to be fulfilled when using the System Calls layer with the build option SYS_CALL_MUTEX set to a negative value, otherwise issues will arise. One must be certain about the overall environment. The choice between the 2 other set-ups of SYS_CALL_MUTEX depends on both the available memory and the way the application accesses the file and directory. If the application is on a limited data memory platform, then SYS_CALL_MUTEX should definitely be set to a value of zero. Also, if there are very little concurrent accesses to the mass storage device, then having individual mutex (SYS_CALL_MUTEX positive) may not provide much real time efficiency saving.

3.3.3 SYS_CALL_N_DRV

The build option `SYS_CALL_N_DRV` sets the maximum number of physical drives the System Calls layer can handle. It is set by default to 1, except for FatFS, which the default value is derived from FatFS's build option `_VOLUME`. To use a different value from the default one, define `SYS_CALL_N_DRV` and set its value to the application requirements.

3.3.4 SYS_CALL_N_DIR

The build option `SYS_CALL_N_DIR` sets the maximum number of directories that can be open at the same time. Although directories may not be accessed directly through the `opendir()`, `readdir()`, etc, using the file statistics API, i.e. `fstat()` and `stat()` do access (open) directories. The default value for `SYS_CALL_N_DIR` is 2. To use a different value from the default one, define `SYS_CALL_N_DIR` and set its value to the application requirements

3.3.5 SYS_CALL_N_FILE

The build option `SYS_CALL_N_FILE` sets the maximum number of files that can be open at the same time. This number does not include the 3 file descriptors for stdio (i.e. stdin, stdout and stderr). By default, the value for `SYS_CALL_N_FILE` is 5. To use a different value from the default one, define `SYS_CALL_N_FILE` and set its value to the application requirements.

3.3.6 SYS_CALL_DEV_@@@

The `@@@` used in the build option name here is a wildcard (it can be `tty`, `i2c`, `spi`, etc). There are multiple `SYS_CALL_DEV_@@@` build options, e.g. `SYS_CALL_DEV_TTY`, `SYS_CALL_DEV_I2C`, etc. By default, all these build options are set to a value of zero, meaning the related virtual devices, e.g. `/dev/tty`, `/dev/i2c`, `/dev/spi`, etc. are not recognized nor handled by the System Calls layer. To make the System Calls layer recognize a specific virtual device, the associated build option must be defined and set to a non-zero value. And, quite important, the associated device driver must be also added to the application build process.

3.3.7 SYS_CALL_TTY_EOF

When virtual TTY devices are recognized and supported by the System Calls layer, it is possible to define the character that reports the end-of-file (EOF). Defining the build option `SYS_CALL_TTY_EOF` and setting it to the desired EOF character will make the system call layer report the end-of-file upon receiving this character, and will drop all future characters received on that terminal device until closed. e.g. to use CTRL-D as the end of file, `SYS_CALL_TTY_EOF` must be set to 0x4, which is the numerical value of CTRL-D. The standard devices input are not monitored for this EOF character. But if this is desired, when `SYS_CALL_DEV_EOF` is defined, set bit #31 of the definition to a 1. In the case of CTRL-D for example, `SYS_CALL_DEV_EOF` would be set to `0x80000004`. Beware that once the "C" library is informed a stdio device has reached the end-of-file, further reading from that device will always report the end-of-file condition.

3.4 Media Access Options

The media accesses are always performed through the API provided by the file `Media_<FS>_<HW>.c`. This file is always file system and platform specific. Although each file system requires a unique media interface, the configuration of the media accesses remains the same across all platforms and file systems. The configuration uses build options to assign mass storage devices to drive numbers. All the build option tokens use the same nomenclature, i.e. `MEDIA_<DEV>#_IDX`, where *<DEV>* is the device type, for example USB or SDMMC, and *#* is the device controller number for that device. When a media access build option is set to a negative value, it means that device number is not used. When non-negative, it indicates the drive number assigned to that device. There is one restriction when assigning the device to the drive numbers, the drive numbers must be contiguous and start at 0. For example, setting `MEDIA_QSPI0_IDX` to 0 and `MEDIA_SDMMC1_IDX` to 2 is incorrect and will issue an error during compilation, as the drive numbers should have been 0 and 1. Please refer to the port specific media access file for the list of devices supported.

When enabling QSPI devices, the slave number is also required. This is defined by the build option `MEDIA_QSPI#_SLV`. If `MEDIA_QSPI#_SLV` is not defined when `MEDIA_QSPI#_IDX` is defined, a default slave index of 0 is used.

4 Multiple File System Stack

The System Calls layer is not limited to the use of a single File System stack: it is capable of supporting multiple File System stacks at the same time. This allows an application the capability to support different file system formats. For example, a FAT32, exFAT, and a NTFS file systems can be combined, delivering the same capabilities as a Window's PC. The use of mount points makes the use of multiple stacks almost transparent to the application.

4.1 Set-up

When building an application with multiple File System stacks, all there is to do is to add the `SysCall_MultiFS.c` to the build, and obviously all the required files for each File System stacks. Re-using the example shown in Table 2-2, and adding NTFS to it, the required files are:

Table 4-1 Cyclone V Multi-FS file example

```
./ --- / --- Abassi / mAbassi.c
       |            |
       |            / mAbassi.h
       |            |
       |            / ARMv7_SMP_L1_L2_GCC.s
       |            |
       |            / SysCall.h
       |            |
       |            / Abassi_FatFS.c
       |            |
       |            / Abassi_NTFS.c
       |            |
       |            / SysCall_ARMCC.c
       |            |
       |            / SysCall_FatFS.c
       |            |
       |            / SysCall_NTFS.c
       |            |
       |            / SysCall_MultiFS.c
       |
       / --- mAbassi_SMP_CortexA9_DS5 / --- inc / dw_qspi.c
       |                                |         |
       |                                |         / dw_sdmmc.h
       |                                |         |
       |                                |         / dw_uart.h
       |                                |         |
       |                                |         / ffconf.h
       |                                |
       |                                / --- src / dw_qspi.c
       |                                          |
       |                                          / dw_sdmmc.c
       |                                          |
       |                                          / dw_uart.c
       |                                          |
       |                                          / Media_FatFS_CY5.c
       |                                          |
       |                                          / Media_NTFS_CY5.c
       |                                          |
       |                                          / SysCall_CY5.c
       |                                          |
       |                                          / mAbassi_SMP_CORTEXA9_GCC.s
       |
       / --- FatFS-0.12 / --- src / ff.c
       |                |
       |                / --- inc / ff.h
       |
       / --- FullFAT-2.0.1 / --- src / …
```

4.1.1 Build options

All the build options listed and described in Section 3.3 are shared amongst all file systems. Therefore, all file systems will have access to the same number of drives, the same number of open files and directories, with the same type of mutex handling, etc. To make the individual System Calls layer files usable in a multiple file system application, the build option `SYS_CALL_MULTI_<FS>` must be defined and set to a non-zero value. Again `<FS>` is the short-name, all in uppercase, for the file system stack (see Table 1-2).

4.2 Implementations

The way the System Calls layer handles multiple file system stacks is to rename all of the System Call Layer APIs with unique names for each file system. For example, when FatFS is integrated in an application with a single file system, the function to open a file is named `open()`. When FatFS is integrated in an application with multiple file system, then this function is named `open_FatFS()`. The system call function `open()` is then located in the file `SysCall_MultiFS.c` and in there is located the logic to select which of the File System stack to use, according to the drive number, and to handle the remapping of all the descriptor numbers associated to the open files and directories. All the File System stacks have access to exactly the same drives. The descriptor remapping is needed because each individual System Calls layer file uses descriptor numbers starting at zero.

4.3 Mounting a device

In the System Calls layer, the mounting operation is one of the only two operations that deals with the physical drive number (the other one is the media formatting). As all drives are shared amongst all the file systems stacks, there is nothing special to do. The multiple file mount function uses the argument specifying the type of file system to mount to determine the proper file system mount function. It is also possible to over-ride this auto selection of the File System stack mount to specify the desired File System stack mount to use.

5 API

The API for the System Calls layer is not described in much detail in this document, simply because all user accessible functions are exactly the same as the UNIX system calls and they are exactly the same as the "C" library itself. There are only three non-standard functions. One is `devctl()`, the device initialization described in Section 2.4.1. Another is the System Calls layer initialization, that must always be performed before using resources from the it, and it is `void SysCallInit(void)`. The last one is `int GetKey(void)`. It simply reads stdin and returns 0 when no characters are available from stdin, else it returns the character that was read. The system calls `mount()`, and `mkfs()` are described because their arguments, although standard, are strings and specific keywords must be used.

5.1 Alike UNIX system Calls

The following functions are the same as the UNIX system calls, described in section #2 of the UNIX man pages:

```
int      chdir(const char *path);
int      chmod(const char *path, mode_t mode);
int      chown(const char *path, uid_t owner, gid_t group);
int      close(int fd);
int      dup(int fd);
int      fstat(int fd, struct stat *pstat);
int      fstatfs(int fs, struct statfs *buf);
off_t    lseek(int fd, off_t offset, int whence);
int      mkdir(const char *path, mode_t mode);
int      mount(const char *type, void *dir, int flags, void *data);
int      open(const char *path, int flags, int mode);
_ssize_t read(int fd, void *vbuf, size_t size);
int      rename(const char *old, const char *new);
int      stat(const char *path, struct stat *pstat);
int      statfs(const char *path, struct statfs *buf);
mode_t   umask(mode_t mask);
int      unlink(const char *path);
int      unmount(const char *dir, int flags);
_ssize_t write(int fd, const void *vbuf, size_t len);
```

5.2 Alike UNIX “C” Library systems calls

The following functions are the same as the UNIX “C” library system calls, described in section #3 of the UNIX man pages:

```
int      closedir(DIR_t *dirp);
char    *getcwd(char *buf, size_t size);
int      isatty(int fd);
int      mkfs(const char *type, void *data);
DIR_t   *opendir(const char *path);
struct dirent *readdir(DIR_t *dirp);
void     rewinddir(DIR_t *dirp);
void     seekdir(DIR_t *dirp, long loc);
long     telldir(DIR_t *dirp);
```

The File System stack FatFS defines and uses its own `dir_t` typedef, therefore it is not possible to use `dir_t` as the directory API of the UNIX “C” library uses. Instead, this typedef is declared as `DIR_t` to not be in conflict with FatFS.

5.2.1 Non-standard functions

```
int      devctl(int fd, const int *Cfg);
int      GetKey(void);
void     SysCallInit(void);
```

5.2.2 devctl

Synopsis

```
#include "SysCall.h"

int devctl(int fd, const int *Cfg);
```

Description

`devctl()` is a component that can be used to initialize any `/dev` device. Its use is straight forward as the first argument, `fd`, is the file descriptor returned by the function `open()` and the other argument, `Cfg`, is an array of `int` that holds all the arguments required by the `????_init()` function used to initialize the type of device specified by `fd`.

Arguments

`fd`	file descriptor returned by open for a "/dev/???"
`Cfg`	array of `int` holding each of the arguments used by the initialization function `????_init()`, excluding the device number, as it is part of the string that was use with `open()` to obtain the device file descriptor.

Returns

`int`	`== 0 :` the device initialization was successful `!= 0 :` the device initialization has failed

Component type

Function

Options

Notes

See example in Section 5.3.

See Also

`i2c_init` (Reference [R7])
`uart_init` (Reference [R8])

5.2.3 GetKey

Synopsis

```
#include "SysCall.h"

int GetKey(void);
```

Description

`GetKey()` is the ubiquitous component to report if a key has been pressed on stdin and return the character of the key pressed or the indication no key has been pressed.

Arguments

`void`

Returns

`int`	`== 0 :`	no key has been pressed
	`!= 0 :`	char of the key pressed

Component type

Function

Options

Notes

`GetKey()` always and only operates on `stdin`. The equivalent operation is not supported on `/dev/tty#` . If an equivalent operation is needed on a /dev/ty#, please use directly `uart_recv()` (See reference [R8]).

See Also

`uart_recv` (See reference [R8])

5.2.4 mount

Synopsis

```
#include "SysCall.h"

int mount(const char *type, void *dir, int flags, void *data);
```

Description

`mount()` is the component for mounting, or attaching, a file system device.

Arguments

`type`	type of file system to mount
`dir`	mount point (must always be at the root level)

`flags`	qualifier when mounting:
	`0`
	`MNT_RDONLY`
	`MNT_UPDATE`

`data`	media device to mount

Returns

`int`	`== 0 :` success `!= 0 :` error

Component type

Function

Options

The argument `type` accepts a limited number of strings (the "" are used to indicate a string, do not use them in the string passed in `type`) listed below and in brackets are the associated tokens defined in `SysCall.h`:

```
"AUTO"      (FS_TYPE_NAME_AUTO)
"exFAT"     (FS_TYPE_NAME_EXFAT)
"FAT12"     (FS_TYPE_NAME_FAT12)
"FAT16"     (FS_TYPE_NAME_FAT16)
"FAT32"     (FS_TYPE_NAME_FAT32)
```

The string comparison performed in `mount()` is case insensitive, therefore `"AUTO"`, or `"auto"`, or `"Auto"` are all considered the same. In almost all cases `type` should be set to `"AUTO"` (or its definition token `FS_TYPE_NAME_AUTO`) as it will try to mount the media by checking all supported file system formats.

The argument `flags` could be set to a value of zero, meaning to mount a media read-write, or set to the token `MNT_RDONLY` to mount the media read-only, or set to the token `MNT_UPDATE` to change the properties of an already mounted media.

The argument `data` is used to indicate the media device to mount. It is a string and should always be `"#:"` where `#` is a digit, from 0 to 9. If an application is built with support for multiple File System stacks, then it is possible to specify the file system stack to use for the mounting operation through the argument `data`. This is done by appending the file system stack name after #: in the string passed as the argument data. The argument data is then alike `#:<FS>`, where:

`#` is a digit from 0 to 9

`<FS>` is the file system stack short name, listed in Table 1-2; as for the argument type, the internal string comparison is case in-sensitive

Notes

See Also

See example in Section 5.3.

5.2.5 mkfs

Synopsis

```
#include "SysCall.h"

int mkfs(const char *type, void *data);
```

Description

`mkfs()` is the component for formatting a file system device.

Arguments

`type`	type of file system to format the target device
`data`	media device to format

Returns

`int`	`== 0 :` success
	`!= 0 :` error

Component type

Function

Options

The argument `type` accepts a limited number of strings (the "" are used to indicate a string, do not use them in the string passed in `type`) listed below and in brackets are the associated tokens defined in `SysCall.h`.

```
"AUTO"        (FS_TYPE_NAME_AUTO)
"exFAT"       (FS_TYPE_NAME_EXFAT)
"FAT12"       (FS_TYPE_NAME_FAT12)
"FAT16"       (FS_TYPE_NAME_FAT16)
"FAT32"       (FS_TYPE_NAME_FAT32)
```

The string comparison performed in `mount()` is case insensitive, therefore `"FAT32"`, or `"Fat32"`, or `"fat32"` are all considered the same.

The argument `data` is used to indicate the media device to format. It is a string and should always be `"#:"` where # is a digit, from 0 to 9. If an application is built with the support for multiple File System stacks, then it is possible to specify the file system stack to use for the formatting operation with the argument `data`. This is done by appending the file system stack name after #: in the string passed as the argument data. The argument data is then alike `#:<FS>`, where:

is a digit from 0 to 9

`<FS>` is the file system stack short name, listed in Table 1-2; as for the argument type, the internal string comparison is case in-sensitive

Notes

See Also

See example in Section 5.3.

5.2.6 SysCallInit

Synopsis

```
#include "SysCall.h"

void SysCallInit(void);
```

Description

`SysCallInit()` is the component used to initialize the System Call.

Arguments

`void`

Returns

`void`

Component type

Function

Options

Notes

`SysCallInit()` must be called once Abassi has been started, i.e. after `OSstart()` has been called, and before using any functions from the System Calls layer. `SysCallInit()` does not initialize the non-File System devices, e.g. I2C or the UART. Non-File System devices can be initialized through the device type specific initialization functions `???_init()`, before or after calling `SysCallInit()`, or they can be initialized through the System Calls layer `devinit()` function. Obviously, using a `/dev` requires the initialization of the device.

See Also

`i2c_init` (Reference [R7])
`uart_init` (Reference [R8])

5.3 Examples

This section shows a few examples for the non-standard functions of the System Calls layer.

5.3.1 devctl() usage

This section provides an example on how to use the function `devctl()`. It shows how to initialize UART #2 on the platform. The configuration shown is for a baud rate of 115200, 8 data bits, no parity, 1 stop bit, a RX queue size of 128, a TX queue size of 128, and some filtering options. If the initialization is performed with `uart_init()`, this would be done as follows:

Table 5-1 UART #2 initialization example

```
uart_init(2, 115200, 8, 0, 10, 256, 128, UART_FILT_OUT_LF_CRLF
                                       | UART_FILT_IN_CR_LF
                                       | UART_ECHO
                                       | UART_ECHO_BS_EXPAND);
```

Using devctl() to perform the initialization involves first to call `open()` to obtain a file descriptor for `/dev/tty2`. Then calling `devctl()` with this file descriptor and an `int` array holding all `uart_init()` arguments, in order, except the first one which is the device number. Error trapping (`open()` error or `devctl()` error) is not shown.

Table 5-2 /dev/tty2 initialization example

```
int Cfg[8];
int Fd;

   Fd = open("/dev/tty2", O_RDWR, 0777);

   Cfg[0] = 115200;                        /* Baud rate                          */
   Cfg[1] = 8;                             /* 8 data bits                        */
   Cfg[2] = 0;                             /* No parity                          */
   Cfg[3] = 10;                            /* 1 stop bit                         */
   Cfg[4] = 256;                           /* RX queue size of 256 elements      */
   Cfg[5] = 128;                           /* TX queue size of 128 elements      */
   Cfg[6] = UART_FILT_OUT_CR_LF            /* UART filtering options             */
          | UART_FILT_IN_CR_LF
          | UART_ECHO
          | UART_ECHO_BS_EXPAND;

   devctl(fd, &Cfg[0]);                    /* Initialze UART #2                  */
```

5.3.2 Mounting

The following table shows how to mount device #1 on the mount point `/dsk1`:

Table 5-3 mount example

```
          mount("Auto", "/dsk1", 0, "1:");
```

The following table shows the same example as the previous one, but it mounts the media in a read-only way:

Table 5-4 mount example

```
mount("Auto", "/dsk1", MNT_RDONLY, "1:");
```

The following table shows how to change an already mounted media from being read-only to be readable and writable:

Table 5-5 mount example

```
mount("Auto", "/dsk1", MNT_UPDATE, "1:");
```

The following table shows the same operation as in the first example, but in an application using the multiple File System stack feature, with the requirement to use the FatFS File System stack when performing the mounting:

Table 5-6 Multi-FS mount example

```
fd = mount("Auto", "/dsk1", 0, "1:FatFS");
```

5.3.3 Formatting

The following table shows how to create a FAT32 media on device #1

Table 5-7 format example

```
mkfs("FAT32", "1:");
```

The following table shows the same operation, but in application using the multiple File System Stack feature with the requirement to use the FatFS File System stack when performing the mounting and allowing FatFS to determine itself which, amongst FAT12, FAT16 or FAT32, is the best format according to the size of the media device.:

Table 5-8 Multi-FS format example

```
fd = mkfs("Auto", "1:FatFS");
```

6 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] FatFS – FAT & exFAT file system, available at http://elm-chan.org/fsw/ff/00index_e.html
[R5] FullFAT – FAT file system, available at https://github.com/jameswalmsley/FullFAT
[R6] Ultra-Embedded FAT file system, available at http:// http://ultra-embedded.com/fat_filelib
[R7] Abassi RTOS – I2C Support, available at http://www.code-time.com
[R8] Abassi RTOS – UART Support, available at http://www.code-time.com