CODE TIME TECHNOLOGIES

Abassi RTOS

QSPI Flash Memory Support

Disclaimer

Table of Contents

List of Figures

List of Tables

1 Introduction

This document describes the QSPI flash memory driver available for Abassi[1] [R1] (including mAbassi [R2] and µAbassi [R3]). The standalone use of the QSPI driver is also described here. The QSPI driver is an integral part of Code Time's System Calls layer [R4] as the driver provides easy access to these mass storage devices.

1.1 Distribution Contents

The set of files supplied with this distribution are listed in Table 1-1 below:

Table 1-1 Distribution

File Name	Description
`???_qspi.h`	Include file for the QSPI flash memory driver (??? is target dependent)
`???_qspi.c`	"C" file for the Abassi QSPI flash memory driver (??? is target dependent)
`Demo_30_`*`PROC`*`.c`	"C" file for testing. *<PROC>* is the target processor name.
`SAL.h`	Include file for the standalone abstraction layer (supplied with standalone package only)
`SAL.c`	"C" file for the standalone abstraction layer (supplied with standalone package only)
`ISRhandler_???.s`	"ASM" add-on file for the standalone version only. It contains support for both the driver and the demo application.

1.2 Limitations

- The QSPI indirect access mode is the only access mode supported; direct and XIP access modes are not supported.
- Write protection of blocks, or sectors, or memory regions are disabled, allowing programming of the whole flash memory.
- When a flash memory has a One Time Programmable (OTP) area, access by reading, or programming this memory area, is not supported by the driver.
- DMA transfers may not be supported for some target processors.
- Support for op-codes sent over 2 or 4 lanes is not supported.
- Only NOR serial flash are supported; serial NAND flash will be in a later release
- The commonly available QSPI flash memory pins `HOLD` and `WP` (Write Protect) are not controlled, nor toggled by the driver.
- Zynq's QSPI controller does not support multi-lane write operation. All multi-lane write operations are internally remapped to `QSPI_CMD_PAGE_PGM`. Refer to the file `xlx_lqspi.c` for more information.

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

1.3 Features

The QSPI driver API is kept the same across all target platforms. Target specific extra functionality is not described in this document; refer to the code itself and embedded comments.

The QSPI flash memory driver can operate using polling only, and/or interrupts with semaphores, and/or DMA transfers, and/or RTOS task sleeps.

As much as possible, the non-volatile bits in the status and configuration register(s) undergo as little programming as possible. This is done by not programming the register if it already holds the correct set-up. Because QSPI flash chips are typically hard-wired on the target platform, the operating conditions are expected to always be the same after the first access.

Each QSPI flash memory part has its own specific operating conditions and custom set-up procedures. For versatility, the QSPI driver individually identifies the parts attached to the controller using the JEDEC ID, and from the part identification it relies on an internal hard-coded definition table to determine the controller configuration to use for each part. The contents of this table is described in Section 3.1.

As much as possible, and depending on the target controller capabilities, the driver aborts transfers upon encountering an error and report the issue; e.g. TX FIFO under-run or RX FIFO overrun. Also monitored is the transfer time; if the transfer takes a time much longer than the expected exchange time then the driver also abort the transfer and report the issue.

1.4 Requirements

When the `WP` (Write protect) pin on the chip is not used as an I/O lane and is driven low (or left un-connected when the part has an internal pull-down), then it is possible some memory areas or the whole memory could be protected against programming depending on the non-volatile setting of the protection bits.

1.5 Nomenclature

All Code Time Technologies driver documentation uses “device” as the name for the hardware module or controller the driver interfaces with. The same use for “device” is maintained here and it does not mean the flash memory part connected to that hardware module, although it is common to use the expression *“memory device”*. Most hardware modules can access multiple flash memory chips. These individual chips are accessed through their “slave” number, which is the chip select line number connected to the QSPI flash memory part.

2 Target Set-up

All there is to do to configure and enable the use of the QSPI driver in an application based on Abassi is to include the following files in the build:

- `???_qspi.c` (For Abassi & Standalone)
- `SAL.c` (For Standalone only)
- `ISRhandler_???.s` (For Standalone only)

and to set-up the include search directory path making sure the file `???_qspi.h` is found (and `SAL.h` for the standalone) and set-up build options as required.

If interrupts are used, one or multiple QSPI interrupt handlers (`QSPIintHndl_n()`, Section 4.1.9) must be attached to the interrupt controller. In Abassi this is simply done using the `OSisrInstall()` component.

The QSPI driver may or may not, depending on the target platform, be independent from other include files.

Before using the driver, please refer to the Appendices (Sections 5, 6, 7, and 8)

2.1 Build Options

There are a few build options that allow the QSPI driver to be configured for the requirements of the target application. The following table lists all of them:

Table 2-1 Build Options

Token Name	Default	Description
`OS_PLATFORM`	`Target dependent`	Number indicating the target platform. Refer to `???_qspi.h` to see the list of supported platforms and the default one.
`QSPI_CLK`	`Target dependent`	Clock frequency of the QSPI controller.
`QSPI_MAX_DEVICES`	`Target dependent`	Number of QSPI controllers(s) supported by the target platform.
`QSPI_LIST_DEVICE`	`Target dependent`	Bit field selecting the QSPI controller(s) to use. The default value is dependent on the build option `OS_PLATFORM`.
`QSPI_MAX_SLAVES`	`Target dependent`	Number of slaves supported by the controller(s) on the target platform.
`QSPI_USE_MUTEX`	`1`	Boolean used to activate the QSPI driver internal protection for exclusive device access.
`QSPI_OPERATION`	`0x10101`	Bit field defining how the QSPI driver operates.
`QSPI_DMA_DEV`	`0`	DMA device # to use when DMA transfers are enable with `QSPI_OPERATION`.
`QSPI_ISR_RX_THRS`	`50`	Threshold in percentage of the RX FIFO size to trigger the RX interrupt.
`QSPI_ISR_TX_THRS`	`0`	Threshold in percentage of the TX FIFO size to trigger the TX interrupt.

QSPI_MIN_4_RX_DMA	64	Minimum number of bytes to read in order to use the DMA instead of polling or interrupts.
QSPI_MIN_4_TX_DMA	16	Minimum number of bytes to write in order to use the DMA instead of polling or interrupts.
QSPI_MIN_4_RX_ISR	64	Minimum number of bytes to read in order to use the interrupts instead of polling.
QSPI_MIN_4_TX_ISR	16	Minimum number of bytes to write in order to use the interrupts instead of polling.
QSPI_MAX_BUS_FREQ	0	Maximum bus clock frequency to use (in MHz)
QSPI_CMD_LOWEST_FRQ	-1	Boolean to select if and how the bus frequency is changed when issuing commands.
QSPI_READ_ONLY	0	Boolean to disable the data writing and erasing capabilities of the driver.
QSPI_KEEP_NONVOLATILE	0	Boolean to enable/disable the modification of non-volatile settings in the QSPI chip.
QSPI_REINIT	1	Boolean to enable/disable a re-init of the driver.
QSPI_MULTICORE_ISR	0	Boolean to enable/disable in the ISR handler when multiple cores can handle the same interrupt.
QSPI_TOUT_ISR_ENB	1	Boolean to enable/disable the interrupts during the timeout check for transfers done through polling
QSPI_REMAP_LOG_ADDR	1	Boolean to enable/disable the conversion from logical to physical address with DMA transfers
QSPI_RDDLY_#_#	-1	Overloads the value of `RdDly` in the part definition table. The first # is the device number and the second # is the slave number
QSPI_XTRA_PARTS	0	Boolean to add new part definitions, or overload existing part defintions by including a file with these definitions.
QSPI_ARG_CHECK	1	Boolean to enable/disable the check on the validity of the API function arguments
QSPI_DEBUG	0	Boolean controlling the sending of progress / debug messages to `stdout`.
QSPI_ID_ONLY	0	Exit from the initialization immediately after reading the JEDEC ID.

QSPI_MX25R_LOW_POW	0	Selects if the Macronix MX25R parts operate in low power mode or high performance.

2.1.1 OS_PLATFORM

The build option OS_PLATFORM informs the QSPI driver about the target platform it operates on. There are two benefits ensuing from the presence of this build option:

- The QSPI driver implicitly knows the total number of QSPI devices and number of slaves on the platform.
- The QSPI driver is able to configure and reset the QSPI devices without application intervention.

2.1.2 QSPI_CLK

The build option QSPI_CLK defines the clock frequency the QSPI module operates with. A default value is set according to the target platform specified by OS_PLATFORM. If the module clock frequency is different from the default value, all there is to do is defined the build option QSPI_CLK and set it to the clock frequency in Hz.

2.1.3 QSPI_MAX_DEVICES

The build option QSPI_MAX_DEVICES informs the QSPI driver of how many QSPI controllers (devices) are on the target platform. If this build option is not set, then the QSPI driver will rely on the build option QSPI_LIST_DEVICE (Section 2.1.4). If the build option QSPI_LIST_DEVICE is also not set, then the QSPI driver will rely on the OS_PLATFORM value (Section 2.1.1).

2.1.4 QSPI_LIST_DEVICE

The build option QSPI_LIST_DEVICE informs the QSPI driver about the individual QSPI controllers (devices) that are used by the application. When the target platform has multiple QSPI devices, enabling only the devices used by the application offers a main benefit:

- Minimize the data memory required by the driver, as there is no need to reserve memory for the queue descriptors / buffers / interrupt handlers and semaphores or optional mutexes of unused devices.

This build option is a bit field, where the bit position represents the QSPI device number. When the corresponding bit is cleared (reset to 0) it specifies the device is not used; when the corresponding bit is set to 1 then the device is used. The following table shows the valid combinations for a target platform with 2 QSPI devices:

Table 2-2 Build Options

QSPI_LIST_DEVICE	QSPI #0	QSPI #1
1	In use	Not used
2	Not used	In use
3	In use	In use

If the build option QSPI_LIST_DEVICE is not externally defined, the default value will be set according to the build option QSPI_MAX_DEVICES (Section 2.1.3). If the build option QSPI_MAX_DEVICES is also not set, then QSPI_LIST_DEVICE will be set according to the build option OS_PLATFORM (Section 2.1.1) and will make all the QSPI devices available on the target platform.

2.1.5 QSPI_MAX_SLAVES

The build option `QSPI_MAX_SLAVES` informs the QSPI driver of how many slaves can be attached to the controllers on the target platform. The default value is target platform dependent. If a single device is attached to the controller, it does not mean `QSPI_MAX_SLAVES` can be set to 1, as `QSPI_MAX_SLAVES` is used to dimension internal arrays that are indexed through the slave number. For example, if the only part attached is tied to the chip select #1 (chip select # starting at 0), then `QSPI_MAX_SLAVES` must be set to a value of at least 2.

2.1.6 QSPI_USE_MUTEX

In an RTOS environment, the driver can provide exclusive access protection to the QSPI device(s) through its internal mutex(es). By default, the build option `QSPI_USE_MUTEX` is set to a non-zero value, meaning the driver uses one mutex per device as the exclusive access protection mechanism. Defining and setting the build option `QSPI_USE_MUTEX` to a zero value will configure the driver to not use mutexes, therefore the application has to enforce that there be no concurrent accesses to the same device (that's the controller, not the memory device).

2.1.7 QSPI_OPERATION

The build option `QSPI_OPERATION` is used to configure how the QSPI flash memory driver operates. This build option is a bit field holding 5 bits. Each of the 5 bits, (bit position #0 is the LSBit), is described in the following table:

Table 2-3 QSPI_OPERATION bit definitions

Bit #	Description
0	Interrupts are disabled during a read burst.
1	Interrupts are used to empty the controller RX FIFO when performing a read.
2	DMA is used to empty the controller RX FIFO when performing a read.
8	Interrupts are disabled during a write burst.
9	Interrupts are used to send the data to the TX FIFO when performing a write.
10	DMA is used to send the data to the TX FIFO when performing a write.
16	Task sleep is used when performing an erase.

Detailed information on how the different settings of `QSPI_OPERATION` modify the operation of the driver is provided in Sections 3.4, 3.5, and 0.

2.1.8 QSPI_DMA_DEV

The build options `QSPI_DMA_DEV`, added in the mid-2020, specifies the DMA device used when DMA transfers are enabled with the build option `QSPI_OPERATION`. Code Time's DMA driver is used for the DMA transfers and by default the DMA device number selected is 0. To use a different device number, define and set the build option `QSPI_DEV_NMB` to the desired device number.

Take not the QSPI driver does not initialize the DMA (i.e. does not call `dma_init()`); it is the responsibility of the application to initialize the DMA before performing transfer operations with the QSPI driver

2.1.9 QSPI_ISR_RX_THRS

The build option `QSPI_ISR_RX_THRS` is used to set the threshold, or watermark, at which the data read interrupts are triggered. When interrupts are used to read from the QSPI flash memory (`QSPI_OPERATION` bit #1 set to 1), the RX interrupt is triggered when the read FIFO holds more than a number of bytes. The build option `QSPI_ISR_RX_THRS` specifies this threshold, in percentage of the FIFO size. Therefore only values between 0 and 100 are accepted for `QSPI_IS_RX_THRS`.

Each application has an optimal value for the RX threshold. To maximize the performance, the interrupt handler should ideally be entered exactly when the FIFO is full or very close to be full. As there is always a bit of latency between the time an interrupt is raised and when the interrupt handler is entered, the optimal threshold should be set to the number of bytes that are transferred on the QSPI bus in the latency duration. Assuming a FIFO of 512 bytes and assuming a latency of 2 *u*s with a QSPI bus of 50 MHz using 4 lanes for the data transfer, then 50 bytes are read in 2 *u*s. This optimal threshold is located at 512-50 bytes, 462 which is 90% of 512. In this example, the optimal value to set `QSPI_ISR_RX_THRS` is 90.

This build option is ignored if bit #1 in `QSPI_OPERATION` (Section 2.1.7) is reset to zero.

2.1.10 QSPI_ISR_TX_THRS

The build option `QSPI_ISR_TX_THRS` is used to set the threshold, or watermark, at which the data write interrupts are triggered. When interrupts are used to write to the QSPI flash memory (`QSPI_OPERATION` bit #9 set to 1), the TX interrupt is triggered when the write FIFO holds less than a number of bytes. The build option `QSPI_ISR_TX_THRS` specifies this threshold, in percentage of the FIFO size. Therefore only values between 0 and 100 are accepted for `QSPI_IS_TX_THRS`.

Each application has an optimal value for the TX threshold. To maximize the performance, the interrupt handler should in theory be entered exactly when the FIFO is empty or very close to be empty. As all QSPI flash memories have a bit of latency between the time the last byte has been sent and the write operation terminated, plus as there is a delay from unblocking a task in an interrupt and the task being un-blocked, these two delays should be taken into account when setting the threshold value.

This build option is ignored if bit #9 in `QSPI_OPERATION` (Section 2.1.7) is reset to zero.

2.1.11 QSPI_MIN_4_RX_DMA

The build option `QSPI_MIN_4_RX_DMA` is used to set the minimum number of byte to be read to use DMA transfers. The whole DMA handling always involves a certain amount of CPU overhead for the programming of the DMA and to handle the end of transfer interrupts (if interrupts are enabled). When a small number of bytes are to be read, it is highly probable the time required to perform the read is less than the overall DMA set-up and interrupt handling overhead. When the RX DMA transfers are enabled through the build option `QSPI_OPERATION`, if the number of bytes to read is less than the value specified by `QSPI_MIN_4_RX_DMA`, the read transfer is performed through polling or ISRs instead of using the DMA.

This build option is ignored if bit #2 in `QSPI_OPERATION` (Section 2.1.7) is reset to zero.

2.1.12 QSPI_MIN_4_TX_DMA

The build option `QSPI_MIN_4_TX_DMA` is used to set the minimum number of byte to be written to use DMA transfers. The whole DMA handling always involves a certain amount of CPU overhead for the programming of the DMA and to handle the end of transfer interrupts (if interrupts are enabled). When a small number of bytes are to be written, it is highly probable the time required to perform the write is less than the overall DMA set-up and interrupts overhead. When the TX DMA transfers are enabled through the build option `QSPI_OPERATION`, if the number of bytes to write is less than the value specified by `QSPI_MIN_4_TX_DMA`, the write transfer is performed through polling or interrupts instead of DMA.

This build option is ignored if bit #10 in `QSPI_OPERATION` (Section 2.1.7) is reset to zero.

2.1.13 QSPI_MIN_4_RX_ISR

The build option `QSPI_MIN_4_RX_ISR` is used to set the minimum number of byte to be read to use the interrupts. The whole interrupt handling always involves a certain amount of CPU overhead: a task becoming blocked, interrupt handler operating, and unblocking the task. When a small number of bytes are to be read, it is highly probable the time required to perform the read is less than the overall interrupt overhead. When the RX interrupts are enabled through the build option `QSPI_OPERATION`, if the number of bytes to read is less than the value specified by `QSPI_MIN_4_RX_ISR`, the read transfer is performed through polling instead of using interrupts.

This build option is ignored if bit #1 in `QSPI_OPERATION` (Section 2.1.7) is reset to zero.

2.1.14 QSPI_MIN_4_TX_ISR

The build option `QSPI_MIN_4_TX_ISR` is used to set the minimum number of byte to be written to use the interrupts. The whole interrupt handling always involves a certain amount of CPU overhead: a task becoming blocked, interrupt handler operating, and unblocking the task. When a small number of bytes are to be written, it is highly probable the time required to perform the write is less than the overall interrupt overhead. When the TX interrupt are enabled through the build option `QSPI_OPERATION`, if the number of bytes to write is less than the value specified by `QSPI_MIN_4_TX_ISR`, the write transfer is performed through polling instead of using interrupts.

This build option is ignored if bit #9 in `QSPI_OPERATION` (Section 2.1.7) is reset to zero.

2.1.15 QSPI_MAX_BUS_FREQ

The build option `QSPI_MAX_BUS_FREQ`, when set to a non-zero value indicates to the driver the maximum QSPI bus clock to use; the value indicated the maximum frequency in MHz. The limitation of the bus clock frequency may be needed when the external hardware cannot reliably operate at the maximum frequency a memory chip supports. The maximum frequency setting is used on all devices and all slaves.

2.1.16 QSPI_CMD_LOWEST_FRQ

Some parts can handle commands to access the chip registers at full data rate when others cannot. The build option `QSPI_CMD_LOWEST_FRQ` when set to a non-zero, sets the SPI bus at it lowest possible frequency when issuing commands (these are not the data read or write commands but typically accesses to the chip registers). When the build option is set to a value of zero, the bus frequency remains the same as currently sets. When sets to non-zero, a positive value disables the controller and re-enable the controller when the bus frequency is changed, and when sets to a negative value, the controller is kept enable when changing the bus frequency. By default, the build option is set to a negative value, meaning the lowest bus frequency is used and the controller remains enable when changing the bus frequency.

2.1.17 QSPI_READ_ONLY

The build options `QSPI_READ_ONLY` controls if the driver can write or erase parts. This build option is a Boolean; when set to a non-zero value the driver will not write data nor perform erasure.

2.1.18 QSPI_KEEP_NONVOLATILE

The build options `QSPI_KEEP_NONVOLATILE`, added in the mid-2020, controls if the driver can change non-volatile settings in the part. This build option is a Boolean; when set to a non-zero value the driver will not write data nor perform erasure. By default it is set to zero therefore the driver modifies the non-volatile setting to achieve the fastest data transfers.

If the Abassi's driver is the only software accessing the QSPI chip, the setting can remain to allow the modification of non-volatile settings. If other software access the QSPI, e.g. boot loader, external programmer, other core running an independent S/W, then it is on a case by case and the build option should initially be set to a non-zero value to not touch the non-volatile settings.

One example where the non-volatile setting should no be changed is the Altera/Intel SocFPGA boot loader with some Spansion parts – the boot process fails because the boot loader expect the non-volatile setting of the number of delay cycles to be set at the factory default.

One exception is the Quad Enable bit in the non-volatile setting that will always be turned on if needed.

2.1.19 QSPI_REINIT

The build options `QSPI_REINIT`, added in the mid-2020, controls if the a device can be re-initialized or not. This build option is a Boolean; when set to a non-zero value the driver will go through the whole init sequence when `qspi_init()` is called; this is the default value. When cleared to zero only the first time `qspi_init()` is called will the whole initialization procedure occurs; further call to `qspi_init()` will immediately exit the initialization. The main purpose to only allow the initialization to occurs once is when a file system initializes the media multiple times; one such example is FatFS.

2.1.20 QSPI_MULTICORE_ISR

This build option is only used on a multi-core application. When the interrupt controller sends or is set-up to send the same interrupt on multiple cores, then it is necessary to perform special handling in the interrupt handlers to make sure the same interrupt is not processed twice. When the build option `QSPI_MULTICORE_ISR` is set to a non-zero value, the ISR handlers are configured to properly handle multiple occurrences of the same interrupt. Setting this build option to non-zero in a single application or in multi-core application where the interrupt is targeted to a single core will not cause problems other than adding code that slows down the operation of the ISR handler.

2.1.21 QSPI_TOUT_ISR_ENB

The build option `QSPI_TOUT_ISR_ENB` is a Boolean controlling if interrupts are re-enable when checking transfer timeouts during transfers through polling. When a transfer through polling is performed and the interrupts are disabled during the burst (controlled with `QPSI_OPERATION`), this could make the update of the RTOS timer tick impossible as the timer tick counter is updated through interrupts. This build option should be set to non-zero, unless timeouts are not required (this could end up in the application lock-up), or on a multi-core application where more than one core receives the RTOS timer tick interrupt.

The setting of `QSPI_TOUT_ISR_ENB` does not affect the timeout check when the transfer is interrupt or DMA based, i.e. `QSPI_OPERATION` bits #1, or #2, or #9, or #10 set to 1.

2.1.22 QSPI_REMAP_LOG_ADDR

When the MMU is set-up to remap memory areas at different addresses from the physical address, it is necessary to convert the logical address to their physical equivalents when using DMA transfers. The build option `QSPI_REMAP_LOG_ADDR` is a Boolean that selects if the addresses used by the DMA are converted from logical to physical. By default it is set to a non-zero value (enable). Although the remapping function is a low instruction count, one may want to not perform a redundant remapping when the logical addresses are the same as the physical. The remapping can be turned off setting the build option `QSPI_REMAP_LOG_ADDR` to zero.

2.1.23 QSPI_RDDLY_#_#

The `RdDly` entry in the part definition table (See section 3.1.8) is target board dependent and a default value of 10 *n*s is used for the `RdDly` of all parts. As this defines a target board specific delay, the build options `QSPI_RDDLY_#_#` allows the application to ignore the value of `RdDly` in the part definition table and use the one specified by `QSPI_RDDLY_#_#`. As for the table entry, the number specifies the number of nanoseconds (*n*s) to use as the delay. If the default value from the definition table is desired, either do not define the build option `QSPI_RDDLY_#_#` or define it and set it to a negative value. The devices for which the build option applies is the number (from 0 and up) specified by the first (leftmost) # in the build option name. The slave for which the build option applies is the number (from 0 and up) specified by the second (rightmost) # in the build option name.

Note: these build options are only available in releases on or after 2019.

2.1.24 QSPI_XTRA_PARTS

The build option `QSPI_XTRA_PARTS` is a Boolean that informs the QSPI driver to include the file `QSPIxtraParts.h` as an add-on to the internal part definition table. If the build option is not defined or if it is defined with a value of 0, the file `QSPIxtraParts.h` is not included in the part definition table. If `QSPI_XTRA_PARTS` is defined and set to a non-zero value then the file `QSPIxtraParts.h` is included. A template file QSPIxtraPart.s is provided in the "include" directory of the drivers.

It is preferable to add parts using the included file instead of modifying the driver internal part table because it leaves the QSPI driver intact and simplify updating the driver for a new release.

Note: this build option is only available in releases on or after 2019.

2.1.25 QSPI_ARG_CHECK

The build options `QSPI_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, when set to a zero value, it does not check the validity of the arguments.

2.1.26 QSPI_DEBUG

The build options `QSPI_DEBUG` controls the printout of progress and error messages to `stdout`. This build option is a Boolean; when set to a non-zero value the driver will send the messages to `stdout` and when set to a value of zero, it will not send messages to `stdout`. When `QSPI_DEBUG` is set to a value of one (1), only the initialization information is printed out. When `QSPI_DEBUG` is set to a value greater than one (1), all information is printed out.

2.1.27 QSPI_ID_ONLY

The build options `QSPI_ID_ONLY` modifies the driver to exit the initialization (when `qspi_init()` is called) after the part identification sequence. It provides a safe way to slowly bring up the access to a new part. By default, the build option `QSPI_ID_ONLY` is set to 0, letting the driver operate normally. When defined and set to a non-zero value, information will be printed on `stdout` and `qspi_init()` will exit with an error after the part discovery operation.

2.1.28 QSPI_MX25R_LOW_POW

The build options `QSPI_MX25R_LOW_POW` selects if the parts from Macronix's MX25R series operate in low power mode or in high performance mode. By default, they are set to operate in high performance mode. If the low power mode is desired, the build option `QSPI_MX25R_LOW_POW` must be defined and set it to a non-zero value.

3 Operation

This section describes how the QSPI flash memory driver operates and the internal resources it utilizes. One characteristic of the driver is that although many slaves can be attached to a device, the slaves can be different QSPI flash memory parts, as the controller is always configured specifically for the target slave it has to access when either performing a read, a write, or an erase. By using individual configurations for each slave, it maximizes the data transfer performances for all slaves.

Internally, the driver maintains individual descriptors for each device validated in the build option `QSPI_LIST_DEVICE` (See Section 2.1.4). In each device descriptor there is room to hold the controller configuration for `QSPI_MAX_SLAVES` (See Section 2.1.5) individual slaves attached to the device.

3.1 Part Table

Each QSPI flash part or parts family has many specific non-uniform features. To handle as many parts as possible a hard-coded definition table is used internally by the QSPI flash memory driver. This table holds information like the page and sector sizes, the maximum read and write clock frequency, etc. The driver is provided with the table already set-up for many parts. Refer to Appendix C (Section 7) for a list of the parts that are already supported.

The following table shows the "C" data structure used by the part definition table to hold all the required information for the handling of many types of parts by the QSPI driver:

Table 3-1 PartInfo_t

```
typedef struct {
   uint32_t   DevID;               /* Device ID: 0xXXSSPPMM                          */
   uint32_t   Size[6];             /* Page, sub-sect, sect, die, device              */
   uint8_t    EraseCap[5];         /* Erase capabilities (if != 0, opcode)           */
   uint32_t   EraseTime[5];        /* Erase time in ms                               */
   uint32_t   OpRd;                /* Read op-code to use                            */
   uint32_t   ModeRd;              /* Mode byte, when required for read operation    */
   uint32_t   OpWrt;               /* Write op-code to use                           */
   uint8_t    WrtMaxFrq;           /* Maximum SPI clock frequency for write          */
   uint8_t    RdDly;               /* Read logic delay (in ns)                       */
   int        Delay[4];            /* Timing needed to comply with the part timing   */
   uint8_t    DumClkR[16];         /* Max Frequency for #Index Rd dummy clock        */
} Pinfo_t;
```

3.1.1 DevID

The `uint32_t` entry `DevID` is the 3 bytes JEDEC ID of the QSPI flash memory part. The 3 byte positions in the 32 bit integer are as follows:

`0xXXSSPPMM`

where: `MM` is the manufacturer

`PP` is the part identifier

`SS` is the part size

`XX` is most of the time `0x00`, but may be different depending on the part. For example, Spansion's `S25FL256` is offered in two variants: one with 64KBytes sectors and the other with 256KBytes sectors. One extra byte (the fifth in the device ID and common interface register) must also be read to discriminate between the two parts, and it is that byte that is added in the MSByte of `DevID`. Another case when `XX` is not `0x00` is when there are conflicts due to the manufacturer re-using the same JEDEC ID for parts with different functionality; what is called here "part variants". One such case is Macronix's `MX25L` series (refer to Section 7.1 for more details).

3.1.2 Size

The field `Size` is an array of 5 `uint32_t` entries. It holds the size of the different blocks on which operations can be performed on the part. The values specify the page, sub-sector, sector, die and full part sizes.

`Size[0]:` Size in bytes of a page. When a write is performed on a QSPI flash memory, a page is the size area and boundaries limits that can be written in a single write burst. QSPI flash memory pages are typically 256 bytes, and a few rare ones have pages of 512 bytes.

`Size[1]:` Size in bytes of a sub-sector. When a part supports sub-sectors, it is the smallest block size that can be erased when a page (entry #0) cannot be erased. If a part does not support sub-sectors, then the smallest erase block size is a sector and the value in `Size[1]` is a *don't care*.

`Size[2]:` Size in bytes of a sector. All parts can erase sectors.

`Size[3]:` Size in bytes of a die. Some large size parts are composed of multiple dies, or are subdivided in blocks that a single read command cannot cross. A part with multiple dies has some restrictions on the reading and the erasing operations. When a part is single die, the die size `Size[3]` must be set to the same value as the part size (`Size[4]`).

`Size[4]:` Size in bytes of the part; this is the total size of the part.

3.1.3 EraseCap

The field `EraseCap` is an array of 5 `uint8_t` entries. It holds the op-code for erasing the associated block size set in the field `Size[]`. When the part does not support the erasure of the related block size, a value of zero must be used in associated `EraseCap[]` entry.

`EraseCap[0]:` Op-code to erase a page; most likely zero as very few parts support page erase.

`EraseCap[1]:` Op-code to erase a sub-sector. When erasure of sub-sector is not supported by the part, a value of zero must be specified.

`EraseCap[2]:` Op-code to erase a sector. As a sector is a standard block size that can be erased, a value of zero should never be used.

`EraseCap[3]:` Op-code to erase a die. When a part is a single die, a value of zero must be specified.

`EraseCap[4]:` Op-code for bulk erase of the part. This is to erase the whole memory in the part and should never be zero.

There are definitions (`QSPI_CMD_ERASE_4K`, `QSPI_CMD_ERASE_SECT`, `QSPI_CMD_ERASE_CHIP`) for the "standard" op-codes (supported by Cypress/Spansion, Micron and Winbond) in the file `???_qspi.c`.

3.1.4 EraseTime

The field `EraseTime` is an array of 5 `uint32_t` entries. It holds the erase time, in *m*s, of the related block size indicated in the field `Size[]`. The value set in the table is 90% of the typical erase time specified by the manufacturer. When the part does not support the erasure of the related block size (as indicated by `EraseCap[]`), the value is ignored. The erasure time is only used when the driver is used with Abassi and is set-up (see build option `QSPI_OPERATION`, Section 2.1.7) to put the task to sleep when waiting for the completion of the erasure.

`EraseTime[0]:` Erase time for a page. The value is most likely a *don't care* as very few parts support a page erase.

`EraseTime[1]:` Erase time for a sub-sector; value is a *don't care* if the part does not support sub-sectors.

`EraseTime[2]:` Erase time for a sector.

`EraseTime[3]:` Erase time for a die; value is a *don't care* if the part is a single die.

`EraseTime[4]:` Erase time for the bulk erase of the part (full memory erase).

Refer to Section 3.4 for more information on how the entries in `EraseTime[]` are used.

3.1.5 OpRd

The field `OpRd` is a `uint32_t` entry specifying the read op-code to use with the part. It holds more information than the read op-code itself. The LSByte of `OpRd` is the read op-code itself and the next 3 nibbles provide the information about the bus width used by the op-code, the address, and the data. The bus width is specified with 3 values because depending on the read op-code used, the op-code, the address, and the data can be transferred over 1 lane, 2 lanes, or 4 lanes, and the transfer width can be different between the op-code, the address, and the data:

0 : the exchange uses 1 lane (in reality 2 unidirectional lanes: SI / SO)

1 : the exchange uses 2 lanes

2 : the exchange uses 4 lanes

Considering the following 32 bit number:

`0x000DAIRR`

`RR`: Read op-code byte

`I`: Op-code bus width (0,1,2)

`A`: Address bus width (0,1,2)

`D`: Data bus width (0,1,2)

There are definitions, prefixed with `QSPI_CMD_READ`, for the "standard" read op-codes (supported by Cypress/Spansion, Micron and Winbond) in the file `???_qspi.c`.

IMPORTANT: Do not forget the 3 nibbles when hand coding the op-code in hex!!!

3.1.6 ModeRd

The field `ModeRd` is a `uint32_t` entry specifying if a mode byte is used when initiating a read operation and, if it is needed, what is the value of the mode byte to send to the part. When `ModeRd` is a non-zero value, the LSByte of `ModeRd` is sent out as the mode byte when performing a read operation. If the mode byte to transfer is zero, then at least one bit in the 32 bit `ModeRd` (other than the in the LSByte) must be set to a non-zero value. If `ModeRd` is 0, then no mode byte is sent out when performing a read transfer.

3.1.7 OpWrt

The field `OpWrt` is a `uint32_t` entry specifying the write op-code to use with the part. It holds more information than the op-code itself. The LSByte of `OpWrt` is the write op-code itself and the next 3 nibbles provide the information about the bus width used by the op-code, the address, and the data. The bus width is specified with 3 values:

0 : the exchange uses 1 lane (in reality 2 unidirectional lanes: SI / SO)

1 : the exchange uses 2 lanes

2 : the exchange uses 4 lanes

Considering the following:

`0x000DAIWW`

`WW`: Write op-code byte

`I`: Op-code bus width (0, 1, 2)

`A`: Address bus width (0, 1, 2)

`D`: Data bus width (0, 1, 2)

There are definitions (`QSPI_CMD_PAGE_PGM`, `QSPI_CMD_2PAGE_PGM`, `QSPI_CMD_4PAGE_CMD`) for the "standard" write (programming) op-codes (supported by Cypress/Spansion, Micron and Winbond) in the file `???_qspi.c`.

IMPORTANT: Do not forget the 3 nibbles when hand coding the op-code in hex!!!

WrtMaxFrq

The field `WrtMaxFrq` is a `uint8_t` entry specifying the maximum frequency that can be used when writing to the part. This maximum frequency is in MHz units, and applies to the write op-code selected in the field `OpWrt`.

3.1.8 RdDly

The field `RdDly` is only used with the Cadence QSPI controller. It specifies the delay in *n*s to apply in the controller internal read logic. Empirically, 10 *n*s is the best value to use found for all development boards Code Time used to test the QSPI driver.

IMPORTANT: The optimal delay to use is target board dependent. The default delay of 10 *n*s is likely either non-optimal or simply wrong for some target boards. The build option `QSPI_RDDLY_#_#` (See section 2.1.23) should be used to overload the value in the part definition table.

3.1.9 Delay

The field `Delay` is an array of 4 `uint32_t` entries, used to make the controller comply with the part timing characteristics when performing transactions with the attached part.

`Delay[0]:` Delay in *n*s from the validation of the chip select and the first bit exchanged. This first bit is always an op-code bit.

`Delay[1]:` Delay in *n*s from the last bit exchanged and the invalidation of the chip select.

`Delay[2]:` Delay in *n*s between a chip select invalid and the chip select being valid again.

`Delay[3]:` Delay in *n*s between two transactions

Most of the time, the values set-up are much larger than the chip minimal requirements. This was chosen to keep a large margin of safety.

3.1.10 DumClkR

The field `DumClkR` is an array of 16 `uint8_t` entries, holding the information about the required number of dummy clocks according to the bus frequency, when performing the type of read operation using the op-code specified in the field `OpRd`. The values in the array represent the maximum frequency, in MHz, for 0 to 15 dummy cycles. The first entry, index #0 is the maximum frequency up to which no dummy cycles are needed. The second entry, index #1, is the maximum frequency up to which 1 dummy cycle can be used. The third entry, index #2, is the maximum frequency up to which 2 dummy cycle can be used… etc. If a part requires to have a minimum of `X` dummy cycles, no matter the bus frequency, then all entries from index #0 to index #`X-1` must be set to a value of zero. The `DumClkR[]` array must be fully initialized: all entries past the maximum frequency must be filled with the maximum frequency value, and when a jump in the number of dummy cycle occurs, the lower value must be duplicated (see example below with 5 to 7 dummy cycles).

For example, let's assume a part that always requires 2 dummy cycles up to 50 MHz, then it requires 3 dummy cycles up to 60 MHz, 4 up to 70 MHz, 5 up to 80 MHz, 7 up to 90 MHz, and 8 at 100 MHz, the maximum of its operations. The `DumClkR[]` array is initialized with the following values:

```
{ 0, 0, 50, 60, 70, 80, 80, 90, 100, 100, 100, 100, 100, 100, 100, 100 }
```

The value 80 is duplicate for 5 and 6 dummy cycles as there is a jump from 5 dummy cycles (80 MHz) to 7 dummy cycles (90 MHz) and all values above 8 dummy cycles are set to the maximum value of 100 MHz.

It is important to remember the values that are set in the `DumClkR[]` fields are most of the time only valid for the read op-code in the field `OpRd`. If the read op-code is changed, it is highly probable the values in the array `DumClkR[]` will not be correct unless they are updated for the new read op-code.

TRICK: The last entry in `DumClkR`, index 15, is the entry used to set the bus frequency for read operations. Changing only the last entry makes it possible to lower the bus frequency without updating the whole array.

3.2 General Information

Depending on the driver, when a part requires more than 3 bytes for the addressing, the read / write, and erase operation (a flash memory with more than 128 Mbit or 16 Mbytes), the part will be used either by writing the 4th byte or MSByte in the appropriate register or by configuring the part to convert all 3 byte address instructions into 4 byte address instructions (what Micron calls the "Enhanced SPI protocol"). The use of 4 bytes *vs* programming the 4th byte depends on the controller capabilities and is purely target dependent.

The driver never configures, nor uses the part in the mode where the command byte is transferred on 2 or 4 lanes. This was chosen as most of the time when a part is configured in this mode the information is held in non-volatile configuration bits. This means upon power-up, the part would immediately operate in this mode and this would make the initialization more complex because the driver's first step is to identify the part it has to initialize. If the 2 or 4 lane op-codes were used, the identification would fail using a single lane. As the part is not yet identified, it would be difficult to know what bit to modify to set-up the part to use a single lane. So the controller would need to be set-up to use 2 lane op-codes and try again to identify the part. If that failed too, then the controller would need to be set-up to use 4 lane op-codes. After identification, then all set-up commands would need to be different from the ones used, as the op-code would require 2 or 4 lanes. As one can see, this adds a fair amount of complexity in the driver for a mere saving of 4 or 6 clock cycles.

When a part requires a mode byte on read operations, the read operations should never set for continuous read mode, as this feature cannot be used due to the way the driver is operating.

3.3 Initialization

The initialization is done through the component `qspi_init()` (See Section 4.1.5). As most QSPI controllers support multiple slaves, which may be different parts, it is necessary to call `qspi_init()` for each of the slaves attached to the device. The initialization is fairly straightforward. First the JEDEC ID of the part is retrieved using the lowest QSPI bus clock frequency. If the JEDEC ID is found in the part definition table, the per-device/slave descriptor is set-up according to the information in the part definition table, and the part is configured to operate according to the descriptor set-up. If the part is not found in the part definition table, `qspi_init()` exits and reports the error.

3.4 Erasing

QSPI flash memory erasing can only be performed over a minimum block size, being the page, sub-sector, or the sector size of the part. As this is a characteristic of the QSPI flash memories, the driver can only erase memory in block sizes that are multiple of the smallest erase block size and with the erasing starting at addresses aligned with that minimal block size. The component `qspi_blksize()` (See Section 4.1.1) can be used to retrieve the minimum erase block size of the part attached to a device and slave number. As long as the erase size is a multiple of the minimum block and the start address is also aligned, the driver will always erase the required memory area and will do so by minimizing the number of erasure requests to the part. This is to say the driver always use the largest block size available for erasing. For example, considering a part with 4K sub-sectors and 64K sectors, if an erase is requested for a memory area of 0x00020000 bytes starting at address `0x00008000`, then there will be 8 sub-sector erases (blocks of 4KB at addresses between `0x00008000` and `0x0000FFFF`), one erase of a sector (blocks of 64KB at addresses between `0x00010000` and `0x0001FFFFF`) and 8 sub-sector erases (blocks 4 KB at addresses from `0x00020000` to `0x00027FFF`).

When an erase request is sent to a QSPI flash memory, it takes some time for the erasure to be completed. Depending on the erasure size, the erasure time can take from a fraction of a second to a many minutes (some 1 Gb parts typically take 10 to 12 minutes for a full erasure). Instead of continuously poll until the erasure is completed, the driver can be configured to put the task to sleep for the time indicated in the field `EraseTime[]` in the part definition table. When bit #16 of the build option `QSPI_OPERATION` (See 2.1.7) is set to 1, the driver will go to sleep. If the bit is reset to 0, the driver will continuously poll until the part reports the completion of the erasure. When the driver puts the task to sleep, it will do so at first for the time from the part definition table. After this sleep time, if the erasure is not completed, then the task will continuously be put to sleep for 1/32 of the first sleep time until the erasure is completed.

3.5 Writing

The driver does not have any restriction when requested to write data to a QSPI flash memory. That is, although a write operation on QSPI flash memories is restricted to a page size and bounded by the page size, the driver, when needed, performs multiple writes requests. Another characteristics of the driver is that although some controllers can only perform a write in multiple of 2 or 4 bytes, when operating in an application with these controllers and a write request is not a multiple of 2 or 4 bytes, the driver fills the remainder with `0xFF` as writing `0xFF` is a do-nothing on NOR QSPI flash memories.

The build option `QSPI_OPERATION` (See Section 2.1.7) bit #8, bit #9, and bit #10 settings are used to control how the driver operates when performing a write. When bit #8 is set to 1, it configures the driver to disable the interrupts when performing a write burst. The interrupts are disabled as long as the data burst, which is the data to write within a page, hasn't been fully transferred to the TX FIFO of the controller. If bit #8 in `QSPI_OPERATION` is instead reset to zero, the interrupts are never disabled when performing a write.

Bit #9 in the build option `QSPI_OPERATION`, when set to 1, configures the driver to operate using interrupts when performing a write operation. The way the interrupts are handled is as follows: first the data for the burst is fully written into the TX FIFO of the controller. Once all the data burst has been transferred, the driver then blocks the task by waiting on a semaphore. The interrupt is triggered when the TX FIFO contains less data than a preset threshold (typically when empty or very close to). In the interrupt, the semaphore the task is blocked on is posted. The last write performed in the interrupt is always done with a number of bytes that matches the threshold. This guarantees all data is transferred in the interrupt and no residual polling is requires. The verification for the part having completed the write operation cannot be used as the source of interrupt as this information must be extracted from the QSPI flash memory part. When the semaphore is posted, the task gets unblocked and polling is done to make sure the write operation has been completed by the part. The next transfer is then started and the sequence repeated. If lucky, the time taken from when the interrupt is raised to when the code start the check if the write is done will be the same as required by the part to complete the write.

Bit #10 in the build option `QSPI_OPERATION`, when set to 1, configures the driver to operate using DMA transfers when performing a write operation. The DMA is set-up for the transfer using an optimal TX FIFO threshold, the transfer is launched, and the end of transfer is waited for. The verification for the part having completed the write operation cannot be used as the source of interrupt as this information must be extracted from the QSPI flash memory part. When the semaphore is posted, the task gets unblocked and polling is done to make sure the write operation has been completed by the part. The next transfer is then started and the sequence repeated. If lucky, the time taken from when the interrupt is raised to when the code start the check if the write is done will be the same as required by the part to complete the write.

One should be aware when using interrupts or DMA that the task blocking may not free much CPU. If the TX FIFO of the controller is much smaller than the size of the part's page, then the blocking, interrupt overhead, and unblocking of the task may take longer than waiting for the small TX FIFO content to be sent to the memory and the write to be completed by the part.

Depending on the setting of the QSPI_OPERATION bits #9 and #10, polling, ISR of DMA transfer will be selected according to:

- Assume doing polling
- If `QSPI_OPERATION` bit #9 is non-zero and # bytes to write > `QSPI_MIN_4_TX_ISR` then use ISRs
- If `QSPI_OPERATION` bit #10 is non-zero and # bytes to write > `QSPI_MIN_4_TX_DMA` then use DMA
- If `QSPI_OPERATION` bit #9 is non-zero, the end of transfer waiting is done blocking on a semaphore posted by the ISR handler.

3.6 Reading

The driver does not have any restrictions when requested to read data from a QSPI flash memory. Although some controllers can only read multiples of 2 or 4 bytes, when the read request is not a multiple of 2 or 4 the driver will drop the extra bytes. Also, some controllers (e.g. Xilinx Zynq's[2]) can only be configured to use dummy bytes and not dummy cycles. So depending on the number of lanes used for the reading, this type of controller limitation can only operate with dummy cycles in multiples of 2, 4 or 8. With this kind of controller, the QSPI driver does not rely on the controller's dummy byte capabilities as it sets-up the controller to never use dummy byte. Instead, the driver performs the reading of these dummy data bits and shifts the overall data read according to the required optimal number of dummy cycles.

Read operations on QSPI flash memories can be continuous, no matter the number of bytes to read, except when a part has multiple dies. When a part has multiple dies, and the read request crosses die boundaries, the driver performs individual read bursts for each dies.

[2] This limitation of the controller is such that on the Zedboard and its Spansion part, the maximum usable QSPI bus frequency for the read is 50 MHz. By making the driver handle the dummy cycles, the maximum bus frequency goes up to 100 MHz, close to the 104 MHz maximum of the part.

The build option QSPI_OPERATION (See Section 2.1.7) bit #0, bit #1, and bit #2 settings are used to control how the driver operates when performing a read. Bit #0 in QSPI_OPERATION, when set to 1, configures the driver to disable the interrupts when performing a read burst. The interrupt disabling lasts as long as the data burst, which is the data to read, hasn't been fully transferred from the RX FIFO of the controller.

Bit #1 in the build option QSPI_OPERATION, when set to 1, configures the driver to use interrupts. The way the interrupts are used is as follows: the driver makes the task block on a semaphore that will be posted in the interrupt when all the data of the burst has been read. When the RX FIFO contains more data than a preset threshold, which is typically around 75% the size of the RX FIFO, an interrupt is generated by the controller. In the interrupt handler, the RX FIFO is completely emptied and when all the bytes in the data burst have been read, the interrupt generation is turned off and the semaphore the task was blocked on is posted. There are no conflict with having both bit #0 and #1 being set. When the interrupts are used (bit #1 set to 1), the interrupts are not disabled even if bit #0 is set to 1.

Bit #2 in the build option QSPI_OPERATION, when set to 1, configures the driver to operate using DMA transfers when performing a read operation. The DMA is set-up for the transfer using an optimal RX FIFO threshold, the transfer is launched, and the end of transfer is waited for.

One should be aware that using interrupts or the DMA when reading data makes the task block and this might not free much CPU. This is because, depending on the controller's RX FIFO size, it is quite possible to fill the RX FIFO faster than the overall time required to enter the interrupt, copy the data, and exit the interrupt, or to set-up the DMA transfer. For example, a part capable of reading at 100 MHz will fill a 128 byte FIFO in 2.5µs when using 4 lanes to transfer the data. When the read request is less than 64 bytes, the interrupts are not used even if bit #1 is set. This overriding is done as most of the time the part is read in the neighborhood of 100 MHz and 64 bytes or less will be read around less than 1 µs.

- Assume doing polling
- If QSPI_OPERATION bit #1 is non-zero and # bytes to write > QSPI_MIN_4_RX_ISR then use ISRs
- If QSPI_OPERATION bit #2 is non-zero and # bytes to write > QSPI_MIN_4_RX_DMA then use DMA
- If QSPI_OPERATION bit #1 is non-zero, the end of transfer waiting is done blocking on a semaphore posted by the ISR handler.

3.7 Multiple Drivers

It is possible to use 2 or more drivers for different QSPI controllers. Example of the need for this is a processor with on-chip QSPI(s) on a board with different type of QSPI controller or a SocFPGA with added QSPI controller in the FPGA fabrics that are of different type than the processor system QSPI contoller. To use multiple drivers the build option QSPI_MULTIPLE_DRIVER must be defined and set to a non-zero value. This changes the API names of the driver by pre-pending the QSPI type to the function names. For example, if cd_qspi.c is used, the APIs are named as following:

Table 3-2 BAPI remapping

Original	Multiple
qspi_blksize()	cd_qspi_blksize()
qspi_cmd_read()	cd_qspi_cmd_read()
qspi_cmd_write()	cd_qspi_cmd_write()
qspi_erase()	cd_qspi_erase()
qspi_init()	cd_qspi_int()
qspi_read()	cd_qspi_read()
qspi_size()	cd_qspi_size()

qspi_write()	cd_qspi_write()
QSPIintHndl_#()	cd_QSPIintHndl_#()

The prefix is always the prefix in the file name; e.g. `dw_qspi.c` prefix is dw and `cd_qspi.c` is cd.

All build options if not prefixed apply to all the drivers. To set build options on a per-driver basic al there is to do is used the build option that has been pre-fixed with the same prefix used in the API but in uppercase. For example to set each of the multiple drivers to support 3 devices each, the build option `QSPI_MAX_DEVICES` should defined and set to 3. If the `cd_qspi` and the `dw_qspi` drivers are use together and, as for the example below, there are 1 devices for the `cd` and 2 for the `dw`, then `cd_QSPI_MAX_DEVICES` should defined and set to q and `DW_QSPI_MAX_DEVICES` should defined and set to 2.

A custom wrapper must be provided. The following code shows such a driver for the `cd_qspi` (1 controller accessed as device #0) and the `dw_qspi` (1 QSPI controllers accessed as device #1 and #2):

Table 3-3 Multiple QSPI wrapper example (`QSPI.c`)

```
#include "qspi.h"


/* ------------------------------------------------------------------------------- */

int qspi_init(int Dev, int Slv, int Mode)
{
int RetVal;

   RetVal = (Dev == 0)
          ? cd_qspi_init(Dev,   Slv, Mode)
          : dw_qspi_init(Dev-1, Slv, Mode);

   return(RetVal);
}

/* ------------------------------------------------------------------------------- */

int qspi_erase(int Dev, int Slv, unint32_t Addr, uint32_t Len)
{
int RetVal;

   RetVal = (Dev == 0)
          ? cd_qspi_erase(Dev,   Slv, Addr, Len)
          : dw_qspi_erase(Dev-1, Slv, Addr, Len);

   return(RetVal);
}


/* ------------------------------------------------------------------------------- */

int qspi_read(int Dev, int Slv, unint32_t Addr, void *Buf, uint32_t Len)
{
int RetVal;

   RetVal = (Dev == 0)
          ? cd_qspi_read(Dev,   Slv, Addr, Buf, Len)
          : dw_qspi_read(Dev-1, Slv, Addr, Buf, Len);

   return(RetVal);
}

/* ------------------------------------------------------------------------------- */

int qspi_write(int Dev, int Slv, unint32_t Addr, const void *Buf, uint32_t Len)
{
int RetVal;

   RetVal = (Dev == 0)
          ? cd_qspi_write(Dev,   Slv, Addr, Buf, Len)
          : dw_qspi_write(Dev-1, Slv, Addr, Buf, Len);

   return(RetVal);
}


... and on

/* ------------------------------------------------------------------------------- */

void QSPIintHndl_0(void) {      cd_QSPIintHndl_0(); }
void QSPIintHndl_1(void) {      dw_QSPIintHndl_1(); }
```

```
void QSPIintHndl_2(void) {      dw_QSPIintHndl_2(); }

/* EOF */
```

Table 3-4 Multiple QSPI wrapper example (`QSPI.h`)

```
#ifndef __QSPI_H__
#define __QSPI_H__       1

#include "cd_QSPI.h"                          /* CD QSPI driver                    */
#include "dw_QSPI.h"                          /* DW QSPI driver                    */

#ifndef QSPI_MULTI_DRIVER                     /* It must be defined and set to !=0 */
  #define QSPI_MULTI_DRIVER      0            /* Set 0 to trigger the error message*/
#endif

#if ((QSPI_MULTI_DRIVER) == 0)
  #error "QSPI_MULTI_DRIVER must be defined and set to a non-zero value"
#endif

#define QSPI_MAX_DEVICES ((CD_QSPI_MAX_DEVICES)+(DW_QSPI_MAX_DEVICES))
#define QSPI_LIST_DEVICE ((CD_QSPI_LIST_DEVICE)|((DW_QSPI_LIST_DEVICE)<<2))

/* ------------------------------------------------------------------------------- */

int32_t qspi_blksize  (int Dev, int Slv);
int     qspi_cmd_read (int Dev, int Slv, int Cmd, uint8_t *Buf, int Nbytes);
int     qspi_cmd_write(int Dev, int Slv, int Cmd, uint8_t *Buf, int Nbytes);
int     qspi_erase    (int Dev, int Slv, uint32_t Addr, uint32_t Len);
int     qspi_init     (int Dev, int Slv, uint32_t Mode);
int     qspi_read     (int Dev, int Slv, uint32_t Addr, void *Buf, uint32_t Len);
int64_t qspi_size     (int Dev, int Slv);
int     qspi_write    (int Dev, int Slv, uint32_t Addr, const void *Buf,            \
                                         uint32_t Len);


/* ------------------------------------------------------------------------------- */

extern void QSPIintHndl_0(void);
extern void QSPIintHndl_1(void);
extern void QSPIintHndl_2(void);

#endif

/* EOF */
```

4 API

In this section, the API of all common QSPI driver functions is provided. The next section gives examples on how to use the QSPI.

4.1.1 qspi_blksize

Synopsis

```
#include "???_qspi.h"

int32_t qspi_blksize(int Dev, int Slv);
```

Description

`qspi_blksize()` is a component that reports the minimum block size in bytes of the flash memory part attached to device `Dev` on the port `Slv`. The QSPI minimum block size is the smallest block size that can be erased.

Arguments

`Dev`	Device number (Number starting at 0)
`Slv`	Slave number (Number starting at 0)

Returns

`int32_t`	`> 0`	Block size in bytes of the flash memory part
	`<= 0`	Error

Component type

Function

Options

Notes

`qspi_blksize()` will return an error if the device / slave access hasn't been initialized through `qspi_init()`.

See Also

`qspi_init` (Section 4.1.5)
`qspi_`ection 4.1.7)

4.1.2 qspi_cmd_read

Synopsis

```
#include "???_qspi.h"

int qspi_cmd_read(int Dev, int Slv, int Cmd, uint8_t *Buf,
                  int Nbytes);
```

Description

`qspi_cmd_read()` is a component to send a command to read `Nbytes` from the chip located at the port `Slv` on device `Dev`. This is not the memory read command; the component to read the memory is `qspi_read()`. It is a portal to send commands directly to the chip, alike reading the status register.

Arguments

`Dev`	Device number (Number starting at 0)
`Slv`	Slave number (Number starting at 0)
`Cmd`	Command op-code to send to the device
`Buf`	Buffer that will receive the data sent out by the chip
`Nbytes`	Number of bytes to read from the chip

Returns

`int32`	`== 0` Success
	`<= 0` Error

Component type

Function

Options

Notes

`qspi_cmd_read()` will return an error if the device / slave access hasn't been initialized through `qspi_init()`.

See Also

`qspi_init` (Section 4.1.5)
`qspi_cmd_write` (Section 4.1.3)

4.1.3 qspi_cmd_write

Synopsis

```
#include "???_qspi.h"

int qspi_cmd_write(int Dev, int Slv, int Cmd, uint8_t *Buf,
                   int Nbytes);
```

Description

`qspi_cmd_write()` is a component to send a command to write `Nbytes` to the chip located at the port `Slv` on device `Dev`. This is not the memory write command; the component to write to the memory is `qspi_write()`. It is a portal to send commands directly to the chip, alike writing the configuration register.

Arguments

`Dev`	Device number (Number starting at 0)
`Slv`	Slave number (Number starting at 0)
`Cmd`	Command op-code to sent to the device
`Buf`	Buffer that holds the data to send to the chip
`Nbytes`	nNumber of bytes to write to the chip

Returns

`int`	`== 0` Success
	`<= 0` Error

Component type

Function

Options

Notes

`qspi_cmd_write()` will return an error if the device / slave access hasn't been initialized through `qspi_init()`.

See Also

`qspi_init` (Section 4.1.5)
`qspi_cmd_read` (Section 4.1.2)

4.1.4 qspi_erase

Synopsis

```
#include "???_qspi.h"

int qspi_erase(int Dev, int Slv, uint32_t Addr, uint32_t Len);
```

Description

`qspi_erase()` is the component used to erase an area, or the whole memory, of a QSPI flash memory. The part is accessed according to the device indicated by the argument `Dev`, and the slave number `Slv` on that device. The starting address of the memory area to erase is indicated with the argument `Addr`, and the number of bytes to erase is indicated by the argument `Len`.

Arguments

`Dev`	Device number (Number starting at 0)
`Slv`	Slave to erase (Number starting at 0)
`Addr`	Base address of the memory area to erase
`Len`	Number of bytes to erase

Returns

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

Component type

Function

Options

The build option `QSPI_OPERATION` (See Section 2.1.7 and 3.4) controls the behavior of the driver when a read operation is requested. When `QSPI_OPERATION` bit #16 is set to 1, the driver puts the task to sleep for an amount of time extracted from the part definition table.

Notes

When erasing a block of memory in a QSPI flash memory, the minimum block size that can be erased can be obtained through the component `qspi_blksize()` (See Section 4.1.1). The number of bytes to erase must be an exact multiple of the minimal block size, and the start address must be aligned on the block size. A shortcut is available to perform a bulk erase, instead of having to rely on `qspi_size()`; that is to erase the all the memory of the part by specifying a start address of 0xFFFFFFFF and a memory block of 0 bytes:

```
qspi_erase(Dev, Slv, 0xFFFFFFFF, 0);
```

See Also

`qspi_read` (Section 4.1.6)
`qspi_write` (Section 4.1.8)

4.1.5 qspi_init

Synopsis

```
#include "???_qspi.h"

int qspi_init(int Dev, int Slv);
```

Description

`qspi_init()` is the component used to initialize a QSPI device, specified with the device indicated by the argument `Dev`, and one slave on that device; the slave is identified with the argument `Slv`.

Arguments

`Dev`	Device to initialize (Number starting at 0)
`Slv`	Slave to initialize (Number starting at 0)

Returns

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

Component type

Function

Options

Notes

When multiple slaves are connected on a device, `qspi_init()` must be called using the same device number but setting the slave number to initialize according to the slave on that device.
When `qspi_init()` is used on a slave, the QSPI flash memory part ID is read from the part and from the unique ID, using the information held in the device table.

See Also

`qspi_erase` (Section 4.1.4)
`qspi_init` (Section 4.1.5)
`qspi_read` (Section 4.1.6)
`qspi_write` (Section 4.1.8)

4.1.6 qspi_read

Synopsis

```
#include "???_qspi.h"

int qspi_read(int Dev, int Slv, uint32_t Addr, void *Buf,
              uint32_t Len);
```

Description

`qspi_read()` is the component used to read data from a QSPI flash memory. The part is accessed according to the device indicated by the argument `Dev`, and the slave number on that device; the slave number is identified with the argument `Slv`. The starting address of the block of data to read is indicated with the argument `Addr`, and the number of bytes to read is indicated by the argument `Len`.

Arguments

`Dev`	Device number (Number starting at 0)
`Slv`	Slave to read from (Number starting at 0)
`Addr`	Base address of the block of data to read
`Buf`	Buffer that will receive the data
`Len`	Number of bytes to read

Returns

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

Component type

Function

Options

The build option `QSPI_OPERATION` (See Section 2.1.7 and 0) controls the behavior of the driver when a read operation is requested. When `QSPI_OPERATION` bit #0 is set to 1, interrupts are disabled during each read burst. When bit #1 is set to 1, interrupts are used to transfer the data from the application memory to the TX FIFO of the controller.

Notes

See Also

`qspi_erase` (Section 4.1.4)
`qspi_write` (Section 4.1.8)

4.1.7 qspi_size

Synopsis

```
#include "???_qspi.h"

int64_t qspi_size(int Dev, int Slv);
```

Description

`qspi_size()` is a component that reports the size in bytes of the flash memory part attached to device `Dev` on the port `Slv`; the return value is the total number of bytes of the part.

Arguments

`Dev`	Device number (Number starting at 0)
`Slv`	Slave number (Number starting at 0)

Returns

`Int64_t`	`> 0`	Block size in bytes of the flash memory part
	`<= 0`	Error

Component type

Function

Options

Notes

`qspi_blksize()` will return an error if the device / slave access hasn't been initialized through `qspi_init()`.

See Also

`qspi_blksize` (Section 4.1.1)
`qspi_init` (Section 4.1.5)

4.1.8 qspi_write

Synopsis

```
#include "???_qspi.h"

int qspi_write(int Dev, int Slv, uint32_t Addr, const void *Buf,
               uint32_t Len);
```

Description

`qspi_write()` is the component used to write data to a QSPI flash memory. The part is accessed according to the argument `Dev`, and the slave number on that device; the slave number is identified with the argument `Slv`. The starting address of the block of data to write is indicated with the argument `Addr`, and the number of bytes to write is indicated by the argument `Len`.

Arguments

`Dev`	Device number (Number starting at 0)
`Slv`	Slave to write to (Number starting at 0)
`Addr`	Base address of the block of data to write
`Len`	Number of bytes to write

Returns

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

Component type

Function

Options

The build option `QSPI_OPERATION` (See 2.1.7 and 3.5) controls the behavior of the driver when a write operation is requested. When `QSPI_OPERATION` bit #8 is set to 1, interrupts are disabled during each write bursts. When bit #9 is set to 1, interrupts are used upon completion of the data transfer to the part.

Notes

See Also

`qspi_erase` (Section 4.1.4)
`qspi_read` (Section 4.1.6)

4.1.9 QSPIintHndl_*n*

Synopsis

```
#include "???_qspi.h"

void QSPIintHndl_n(void);
```

Description

`QSPIintHndl_n()` is the interrupt handler for the QSPI driver. The ***n*** in the name is a numerical value that specifies the device number the interrupt handler is for.

Arguments

`void`

Returns

`void`

Component type

Function

Options

The QSPI interrupt handler is not used, therefore not needed, if bit #1 and bit #9 of the build option `QSPI_OPERATION` are both set to 0 (See Section 2.1.7). Interrupt handlers are only available for the QSPI devices validated by the build option `QSPI_LIST_DEVICE`.

Notes

The interrupt handler should always be attached to the targeted QSPI device interrupt and the number of the interrupt handler MUST match the device number. If there is a mismatch, then the application will most likely freeze or even crash. If the interrupt handler is not attached and the related interrupt enabled, the QSPI driver for this device will not operate.

See Also

`qspi_read` (Section 4.1.6)
`qspi_write` (Section 4.1.8)

5 Appendix A

5.1 Bringing a part up

Before using the QSPI driver, the following should be performed, assuming the set-up has an operational `stdout` device:

1) Set the build option `QSPI_ID_ONLY` to a non-zero value and rebuild the application.

2) Perform a call to `qspi_init()` with the appropriate device controller number and slave. Run the application, and if the `stdout` display reports the following:

```
This JEDEC ID is available in the part definition table
```

Then the part has already an entry in the part definition table, which means you can proceed to step 3)

If instead the `stdout` display reports the following:

```
This JEDEC ID is NOT in the part definition table
```

Then it is necessary to add a new entry in the part definition table, which means you must proceed according to the next section (Section 6).

3) As the part is already in the part definition table, it will most likely operate properly "out of the box". The supported parts are listed in Section 7: parts in bold have been physically verified and as such the table entry is correct. Parts not in bold haven't been physically verified meaning the table entry may have errors; please double check these.

Un-define, or set the option `QSPI_ID_ONLY` to zero. Then set to one (1) the build option `QSPI_QUICK_CHECK` for the test file `QSPI_<TOOL>_<HW>.c` and build a test application based on this file.

4) The 256-byte dump on `stdout`, when running the test application, should be exactly like Table 8-1. If it is, you should run the full regression test by setting the build option `QSPI_QUICK_CHECK` to zero and make sure the results are all OK. If the quick test result is not OK, there are a few possible reasons:

a) QSPI bus clock for write is too high

b) QSPI bus clock for read is too high

c) Number of lanes for write is wrong

d) Number of lanes for read is wrong

If the regression test is not OK, then only a) and b) may be at fault.

5) To reduce the QSPI bus clock, these are the 2 fields in the part definition table that should be modified:

```
WrtMaxFrq

DumClkRd[15]
```

Set both fields to 1 (meaning a bus of around 1 MHz) and re-test. If the test passes, then adjust these values for the maximum frequency your target hardware supports.

6) If reducing the bus clock frequency did not solve the problem, it is quite possible it is necessary to reduce the number of lanes used for the transfers. Keeping the maximum clock to 1 MHz set in 5), change the contents of these two fields to set them to the op-codes that use the most basic transfers:

`OpWrt` set to `QSPI_CMD_PAGE_PGM`

`OpRD` set to `QSPI_CMD_READ`

Changing `OpRd` most likely invalidates the `DumClkR` array. Replace the whole `DumClkR[]` array with an array of 16 ones:

```
{ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
```

With these modifications, the QSPI transfers are performed at around 1 MHz with the most basic operations. Unless there is a hardware issue, this should work.

7) Upon success, slowly go back increasing the # lanes and the maximum QSPI bus clock.

6 Appendix B

6.1 Adding a new part

When a new entry must be added in the part definition table, it is most likely that there is more to do than simply adding the part definition table entry. As the part is not in the table, this also means the part family is not in the part definition table. Because of that, initialization code must be added. Areas where custom code must be added for a new part family are always located between these two comments (there are more than one location for code to be added:

```
+++++ PART ADD START ++++++++++++++++++++++
```

and

```
+++++ PART ADD END ++++++++++++++++++++++++
```

Carefully looking at the existing code in-between these comment should provide enough information on how to add the required code.

Adding new code and filling the new table entry always require a lot of care and full understanding of the capabilities and limitation of the part to add.

In case of doubt, do not hesitate to contact Code Time.

7 Appendix C (Supported Parts)

This appendix lists all the parts described in the part definition table. The parts are grouped by manufacturer and then by part family. The parts in **bold** letters are the parts that have been verified to operate correctly with the driver. The table entries of the parts not in bold letters are derived from the parts that have been verified and the table filled with care. Most of the times the only changes from the tested part are the JEDEC ID, sizes, and erase time. But as they haven't been verified with a physical part, these entries are not guaranteed to be correct.

The information on the parts that are in the default table are always for 3V parts. Some parts can operate at much lower voltage, many with different characteristics than the ones when operated at 3V. There are also speed variants for the same part, therefore before using the driver a check on the maximum speed must be performed to make sure it matches the target device.

7.1 Macronix

Macronix offers a full rainbow of variants for their QSPI flash memory parts. The variations are specified with the last 2 digits in the part number, and from the driver perspective, affect the number of lanes (and read-write op-code), dummy cycle when reading, and the write protection.

One specific variant is the one with #73 for which the QUAD I/O (QPI) is permanently enabled. As the driver does not support the 4-4-4 mode during the part initialization, this variant is not supported by the driver.

In the following list, the first line for each series enumerates the parts that have been verified. The two dashes (`--`) after the part numbers are the variant numbers:

MX25L series:	**MX25L1006E**	**MX25L6406E**	**MX25L6445E**	
	MX25L512-	MX25L10--	MX25L20—	MX25L40—
	MX25L80--	MX25L16--	MX25L32--	MX25L64—
	MX25L128--	MX25L256--	MX25L512--	
MX25R series:	**MX25R512F**	**MX25R2035F**	**MX25R6435F**	
	MX25R512	MX25R10--	MX25R20--	MX25R40--
	MX25R80--	MX25R16--	MX25R32—	MX25R64--
MX25V series:	**MX25V8006E**	**MX25V1635F**		
	MX25V512	MX25V10--	MX25V20--	MX25V40--
	MX25V80--			

Note: The MX25L series is the family with the highest number of part variants, and with many parts sharing the same JEDEC ID, there are lots of incompatibilities. The main two incompatibilities are a non-standard support of reading op-codes. The only common read modes are the basic mode and the fast mode. As such, all the part definition table entries for Macronix's MX25L series are set to the fast read mode. The other major incompatibility is that some part have one or more configuration bits to program the part for the number of dummy cycles to use during a 2 I/O or 4 I/O read. Not only do many variants not have the configuration register to set this up, but the one with this set-up capabilities does not have the bit at the same position in the register, when they have a configuration register, depending on the memory size of the part, this set-up uses 1, 2 or 3 bits. The set-up for the dummy cycle control must be provided through the MSByte of the part identification filed DevID. This upper byte holds the information on the LSbit position of the bits to set-up in the configuration register and the bits themselves the set in the register. The upper nibble holds the LSBit position to align to, i.e. the position (from 0 to 7) of dummy cycle configuration LSBit. The lower nibble holds the value to write in the dummy cycle configuration bit(s). Take note that all parts of size 128Mb and larger must use a bit position of 4 due to the possible presence of the *"preamble bit enable"* in the configuration register. The internal operations performed to insert the bits in the configuration register are the following:

Table 7-1 MX25L Dummy Cycle set-up

```
Bits  = (Part->DevID >> 24) & 0xF;       /* Isolate the bits to insert        */
Shift = (Part->DevID >> 28) & 0xF;       /* Isolate the shift to apply        */
if (PartSize >= 0x01000000) {            /* When the part is 128Mb or more    */
    Mask = ~(0x0B << Shift);             /* Mask to zero the bits to insert   */
}
else {
    Mask = ~(0x01 << Shift);             /* Mask to zero the bit to insert    */
}
CfgReg &= Mask;                          /* Zero the all bits to set-up       */
CfgReg |= Bits << Shift;                 /* Insert the desired bits           */
```

Here are three examples:

- The MX25L6436 has 1 bit to set-up the dummy cycle count in the configuration register. This bit is located at position #6 in the configuration register. Therefore, to set the bit in the register, the DevID field for that part should be set to 0x611720C2; that is, the upper byte set to 0x61, indicating the bit is at position #6 and the value of the bit to put in the register is 1. To clear the bit, DevID should be set to 0x601720C2.
- The MX25L25645 has 3 bits to set-up the dummy cycle count and preamble bits in the configuration register. These bits are located at position #4, #6 and #7 in the configuration register. Therefore, to set bit #4 to 1 (enable the preamble) and set bits #6 and #7 respectively to 1 and 0, the DevID field for that part should be set to 0x491920C2; that is, the upper byte set to 0x49, indicating the LSBit is at position #4 and the value of the 3 bits to put in the register is 9 (10X1b). To clear all three bits, DevID should be set to 0x401920C2.
- The MX25L25735 has 2 bits to set-up the dummy cycle count in the configuration register (it does not have a preamble bit enable, instead it is a *don't care* bit). Notice the JEDEC ID of the MX25L25735 is the same as the MX25L25645 used above. These 2 bits are located at position #6 and # 7, but as the part size is larger or equal to 128Mbit, the bit position that must be used is 4. To set the two bits to a value of 2 (10b) in the register, the DevID field for that part should be set to 0x481920C2; that is, the upper byte set to 0x48, indicating the LSBit is at position #4 and the value of the 2 bits to put in the register is 8 (10XXb). To clear the two bits, DevID should be set to to 0x601920C2.

Note: The MX25R series parts can be set-up for an ultra low power mode or a high performance mode. The table entries for the parts of the MX25R series are by default set-up for the high performance mode. To set-up the table entries for the MX25R parts in the ultra low power mode, define the build option QSPI_MX25R_LOW_POW and set it to a non-zero value (Section 2.1.28).

7.2 Micron

Micron series N25Q uses a different JEDEC ID for some of their 1.8V vs. 3.0V devices. Both variants are recognized.

M25P series	M25P05A	M25P10A	M25P20	M25P40
	M25P80	M25P16	M25P32	M25P64
	M25P128			
M25PE series	MSP25PE10	MSP25PE20	MSP25PE40	MSP25PE80
	MSP25PE16			
M25PX series	M25PX80	**M25PX16**	M25PX32	
M45PE series	M45PE10	M45PE20	M45PE40	M45PE80
	M45PE16			
N25Q series:	N25Q032A	N25Q064A	N25Q128A	N25Q256A
	N25Q512A	**N25Q00AA**		

7.3 Cypress / Spansion (Cypress)

FL-P series	**S25FL128P**			
FL-S series:	S25FL127S	S25FL128S	**S25FL256S**	S25FL512S
FL-1K series:	S25FL116K	**S25FL132K**	S25FL164K	
FL-2K series:	**S25FL208K**			

Note: S25FL127S has the same JEDEC ID as the S25FL128S. There is a functional difference between the two: when using the dual I/O mode, the S25FL127S requires a mode byte when the S25FL128S does not need one. Although the resulting number of cycles between the last address bit and the first data bit read is the same for the 2 devices, the driver uses the information if a mode byte is sent out when in the dual I/O in order to select the correct number of dummy cycles to program the device. Therefore it is very important in the dual I/O mode to use a mode byte for the S25FL127S and to not use a mode byte for the S25FL128S.

Note: The S25FL-P series parts have the same JEDEC ID as the S25FL-S series. The key differences are the number of lanes supported, the register set, and the instruction set. As the 25FL-P series is now "Not For New Design", the default table definitions and set-up of the parts are for the S25FL-P. If a S25FL-P part is to be supported, the table entry DevID field must have its most significant nibble (bit 31:28) set to a non-zero value. This will inform the driver about the part being a S25FL-P part and it will perform the set-up accordingly. For example, if a S25FL128P (with 64 KBytes sectors) is to be accessed, then replace the part definition table entry field DevID from 0x01182001 to 0x11182001.

7.4 SST

S25FV series:	S25FV016K

7.5 Winbond

W25Q series:	W25Q20	W25Q40	W25Q80	**W25Q16**
	W25Q32	W25Q64	**W25Q128**	W25Q256
	W25Q257			
W25X series:	W25X05	W25X10	W25X20	**W25X40**

8 Appendix D

The test file, `Demo_30_????.c` is supplied in the distribution as it greatly helps bringing up a new flash memory part, and it also provides a full regression test of the driver and the part.

8.1 Quick Test

The quick test is enabled when the build option `QSPI_QUICK_CHECK` is defined and set to a non-zero value. If the build option is not defined, or if it is defined with a value of zero, the regression test is activated (refer to the next sub-section). The quick test involves writing 256 bytes to the target part, with incrementing values from `0x00` up to `0xFF`. This quick test verifies the capability of the driver to erase the smallest block size, a write of 256 bytes and a read of 256 bytes. Here's a capture of the output with the debug option of the driver turned on (Build option `QSPI_DEBUG` defined and set to a non-zero value):

Table 8-1 Quick Test Output

```
QSPI - Initializing   : Dev:0 - Slv:0
QSPI - part JEDEC ID  : 0x00164001
QSPI - # part checked : 83
QSPI - CTRL clock     : 370000000 Hz
QSPI - SPI clk (wrt)  :  61666666 Hz [/6]
QSPI - SPI clk (read) :  61666666 Hz [/6]
QSPI - Read dummy clk : 1
QSPI - Delay register : 0x0F0F0202
QSPI - Read delay     : 4
QSPI - Status reg #1  : ori=0x00  new=0x00  read=0x00
QSPI - Config reg #2  : ori=0x04  new=0x04  read=0x04
QSPI - Config reg #3  : ori=0x70  new=0x11  read=0x11

QSPI test started

QSPI flash chip size  : 0x00400000
QSPI flash erase size : 0x00001000

erase result 0
Erase successful
Write result 0
read result 0
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F
70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F
80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F
90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF
B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF
C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF
D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF
E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF
F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
```

This quick test is useful for two purposes. Assuming the exchanges are operating correctly, it allows one to immediately see if the number of dummy cycles during a read transfer is correct. If the number of dummy cycles is wrong, the whole data will be shifted by a multiple of bits matching the number of lanes used for the data transfer. Secondly, if the data is garbled there are a lot of chances the entry in the part definition table for the device tested is wrong. Refer to Appendices A and B (Sections 5 and 6) for more details on how to properly set-up a new entry in the part definition table. This is only a quick test and even though the results may be correct, there could still issue related to bus clock or others parameters. The regression test (next sub-section) should always be run to confirm the reliable operation of the QSPI driver with the target part.

8.2 Regression Test

The regression test is enabled by default. To activate the quick test, refer to the previous sub-section. When any error is detected, the test is aborted and an error message is shown to describe the error. The regression test is quite exhaustive and goes through the following steps:

8.2.1 Test 01

First, every smallest erase block out of 13 is zeroed to make sure the chip is not blank followed by a full chip erase is performed.

8.2.2 Test 02

Once the chip is completely erased, it is checked to be blank, meaning the test verifies all data in the chip to be `0xFF`. The reading of the flash is assumed to be operating correctly, although this will be further verified in Tests 04 and 05.

8.2.3 Test 03

Having a blank chip, the whole chip is written with random data. The size of buffer written to the chip are selected to be very large, and decreased in size at every write. The writing of the flash is assumed to be operating correctly, although this will be further verified in Tests 06 and 07.

8.2.4 Test 04

The chip being written with random data, it is all read and each data location verified to hold the data that was written to it. The size of buffer, which is the size of a read request, that is used to read from the chip are selected to be very large, and decreased in size at every read request.

8.2.5 Test 05

This test verifies the read capabilities of the driver. Re-using the random data held in the chip, reads are first performed doing read of 1 to 30 bytes. After these 30 read requests, read requests are performed by size of 257, 2*257, 3*257… bytes. Each read request uses a different base address to read from.

8.2.6 Test 06

This one verifies the write capabilities of the driver. Before every write request, the area to erase is determined and an erase is performed. Writes are first performed doing write of 1 to 30 bytes. After these 30 read requests, write requests are performed by size of 257, 2*257, 3*257… bytes. Each write request uses a different base address to write to.

8.2.7 Test 07

This test erases the chip one block (smallest erase block size) at a time, write a random number of bytes in that block, with random values, stating at a random offset in the block. The non-written area of the erase block is checked to have remained at 0xFF and the written data check to be correct.

8.2.8 Test 08

Test 08 re-verifies the data written and the un-touched areas in the erased block are OK. This is done to double check there are no address aliasing in the erase, write and read operations of the driver.

8.2.9 Test 09

This test is used to verify the erasing performed by the driver is correct. As explained in Section 3.4, the driver minimizes the number of erase commands sent to the chip by always figuring the largest erase block according to the start address and number of bytes left to erase. The test erases from 1 smallest erase block to possibly 512Kbytes of data to erase (it may be less if the part has less then 512Kbytes). The erase start address is always the middle of the memory minus the smallest erase block size.

Here's a capture of the final output with the debug option of the driver turned on (Build option `QSPI_DEBUG` defined and set to a non-zero value). Many test over-write the output line showing each time the different parameters being used:

Table 8-2 Regression Test Output

```
QSPI - Initializing   : Dev:0 - Slv:0
QSPI - part JEDEC ID  : 0x00164001
QSPI - # part checked : 83
QSPI - CTRL clock     : 370000000 Hz
QSPI - SPI clk (wrt)  :  61666666 Hz [/6]
QSPI - SPI clk (read) :  61666666 Hz [/6]
QSPI - Read dummy clk : 1
QSPI - Delay register : 0x0F0F0202
QSPI - Read delay     : 4
QSPI - Status reg #1  : ori=0x00  new=0x00  read=0x00
QSPI - Config reg #2  : ori=0x04  new=0x04  read=0x04
QSPI - Config reg #3  : ori=0x70  new=0x11  read=0x11

QSPI test started

QSPI flash chip size  : 0x00400000
QSPI flash erase size : 0x00001000


Test 01 - Some data zeroing before erasing the chip
          Erasing the chip (This may take a while)
          Chip erased
Test 02 - Checking if all blank
          Checking 4 byte starting from address 0x003FFFFC
          Chip is blank
Test 03 - Full chip writing
          Writing 22 bytes starting at address 0x003FFFEA
          Chip all written
Test 04 - Checking if write OK
          Reading 102 bytes starting from address 0x003FFF9A
          Chip write OK
Test 05 - Checking read of 1800 bytes starting from address 0x000260F8
          Different size read OK
Test 06 - Checking write of 1800 bytes starting at address 0x000260F8
          Different size writes OK
Test 07 - Erasing, writing & checking sector #1023
          Sector erase & writing passed
Test 08 - Checking sector integrity #1023
          Sector integrity is OK
Test 09 - Erase size check
          Checking erase size of 0x00020000 starting from address 0x001ff000
          Erase size is OK


*** Test done ***
```

9 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 RTOS – System Calls Layer, available at http://www.code-time.com