CODE TIME TECHNOLOGIES

Abassi RTOS

SPI Support

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 SPI driver available for Abassi[1] [R1] (including mAbassi [R2] and µAbassi [R3]). The standalone use of the SPI driver is also described here. This is not the document describing the Quad SPI (QSPI) flash memory driver; for information on the QSPI driver refer to [R4].

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
`???_spi.h`	Include file for the SPI driver (??? is target dependent)
`???_spi.c`	"C" file for the Abassi SPI driver (??? is target dependent)
`Demo_40_<PROC>_<TOOL>.c`	"C" file for testing. *<TOOL>* is the name of the build environment tool-set, *<PROC>* is the processor/target.
`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 Features

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

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

The SPI driver can operate using polling, and/or interrupts with semaphores and/or DMA transfers. Depending on the target SPI controller the followings features could be supported:

- Protocols:
  - Motorola SPI protocol
  - Texas Instrument SSP protocol
  - National Semiconductor MicroWire protocol
- Selectable number of bits per frames
- Memory data alignment
  - Left aligned frames
  - Right aligned frames
  - Packed left aligned frames
  - Packed right aligned frames
- Data exchanges:
  - Send only
  - Receive only
  - Send and receive (Full-duplex)
  - Send followed by receive (Half-duplex, or EEPROM type of transfer)
- CPHA and CPOL polarity selection
- Transfers:
  - Using polling
  - Using polling with interrupt at the end of the transfer
  - Done in interrupts
  - Done with DMA
- Error trapping:
  - Transfer abort upon TX FIFO under-run
  - Transfer abort upon RX FIFO overflow
  - Transfer abort when the transfer time is much longer than it should be

1.3 Limitations

Some controllers cannot support some of the features listed in the previous section. Please refer to the specific driver code for a description / list of these limitations; this is described near the top of the files `??_spi.h` and `??_spi.c`.

2 Target Set-up

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

- `???_spi.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 `???_spi.h` is found (and `SAL.h` for the standalone) and set-up build options as required.

If interrupts are used, one or multiple SPI interrupt handlers (`SPIintHndl_n()`, Section 4.1.5) must be attached to the interrupt controller. In Abassi this is simply done using the `OSisrInstall()` component.

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

2.1 Build Options

There are build options that allow the SPI driver to be configured for the requirements of the target application. The following table lists all of them (there is an alternate token naming, refer to section 3.5) :

Table 2-1 Build Options

Token Name	Default	Description
`OS_PLATFORM`	`Target dependent`	Number indicating the target platform. Refer to `???_spi.h` and `Platform.h` to see the list of supported platforms and the default one.
`SPI_CLK`	`Target dependent`	Clock frequency of the SPI module.
`SPI_MAX_DEVICES`	`Target dependent`	Number of SPI controllers(s) supported by the target platform.
`SPI_LIST_DEVICE`	`Target dependent`	Bit field selecting the SPI controller(s) to use. The default value is dependent on the build option `OS_PLATFORM`.
`SPI_MAX_SLAVES`	`Target dependent`	Number of slaves supported by the controller(s) on the target platform. The default value is dependent on the build option `OS_PLATFORM`.
`SPI_USE_MUTEX`	`1`	Boolean used to activate the SPI driver internal protection for exclusive device access.
`SPI_OPERATION`	`0x10303`	Bit field defining how the SPI driver operates.
`SPI_ISR_RX_THRS`	`50`	Threshold in percentage of the RX FIFO size to trigger the RX interrupt.
`SPI_ISR_TX_THRS`	`50`	Threshold in percentage of the TX FIFO size to trigger the TX interrupt.
`SPI_MIN_4_RX_DMA`	`32`	Minimum number of bytes to read in order to use the DMA instead of polling or

		interrupts.
SPI_MIN_4_TX_DMA	32	Minimum number of bytes to write in order to use the DMA instead of polling or interrupts.
SPI_MIN_4_RX_ISR	16	Minimum number of frames to read in order to use the interrupts instead of polling.
SPI_MIN_4_TX_ISR	16	Minimum number of frames to write in order to use the interrupts instead of polling.
SPI_MULTICORE_ISR	1	Boolean to enable/disable in the ISR handler when multiple core can handle the same interrupt.
SPI_TOUT_ISR_ENB	1	Boolean to enable/disable the interrupts during the timeout check for transfers done through polling
SPI_REMAP_LOG_ADDR	1	Boolean to enable/disable the conversion from logical to physical address with DMA transfers
SPI_ARG_CHECK	1	Boolean to enable/disable the check on the validity of the API function arguments
SPI_DEBUG	0	Boolean controlling the sending of progress / debug messages to `stdout`.
SPI_MULTIPLE_DRIVER	0	To use drivers for different SPI controllers at the same time.

2.1.1 OS_PLATFORM

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

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

The information on the numbering used for `OS_PLATFORM` is available in the `Platform.txt` and `Platform.h` files also supplied as part of the distribution.

2.1.2 SPI_CLK

The build option `SPI_CLK` defines the clock frequency the SPI 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 `SPI_CLK` and set it to the clock frequency in Hz.

2.1.3 SPI_MAX_DEVICES

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

2.1.4 SPI_LIST_DEVICE

The build option `SPI_LIST_DEVICE` informs the SPI driver about the individual SPI controllers (devices) that are used by the application. When the target platform has multiple SPI 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 SPI 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 SPI devices:

Table 2-2 Build Options

SPI_LIST_DEVICE	SPI #0	SPI #1
1	In use	Not used
2	Not used	In use
3	In use	In use

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

2.1.5 SPI_MAX_SLAVES

The build option `SPI_MAX_SLAVES` informs the SPI 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 `SPI_MAX_SLAVES` can be set to 1, as `SPI_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 `SPI_MAX_SLAVES` must be set to a value of 2 or more.

2.1.6 SPI_USE_MUTEX

In an RTOS environment, the driver can provide exclusive access protection to the SPI device(s) through its internal mutex(es). By default, the build option `SPI_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 `SPI_USE_MUTEX` to a zero value will configure the driver to not use mutexes, therefore the application has to enforce there be no concurrent accesses to the same device.

2.1.7 SPI_OPERATION

The build option `SPI_OPERATION` is used to configure how the SPI 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 SPI_OPERATION bit definitions

Bit #	Description
0	Interrupts are disabled during a read (receive) burst. To not disable the interrupts during a read burst, bit #0 must be reset to zero. To disable the interrupts during a read burst, bit #0 must be set to 1.
1	Interrupts are used to empty the controller RX FIFO when performing a read (receive) operation and/or interrupts are used to report the end of transmission. To not allow the use of interrupts when reading the RX FIFO, bit #1 must be reset to zero. To allow use of interrupts when reading RX FIFO, bit #1 must be set to 1.
2	DMA is used to empty the controller RX FIFO when performing a read (receive) operation. To not allow the use of the DMA to read the RX FIFO, bit #2 must be reset to zero. To allow the use of the DMA to read the RX FIFO, bit #2 must be set to 1. The end of the transfer can be polled or blocked on an interrupt depending on the setting of bit #1.
8	Interrupts are disabled during a write (send) burst. To not disable the interrupts during a write burst, bit #8 must be reset to zero. To disable the interrupts during a write burst, bit #8 must be set to 1.
9	Interrupts are used to fill the controller TX FIFO when performing a write (send) operation and/or interrupts are used to report the end of transmission. To not allow the use of interrupts when filling the TX FIFO, bit #9 must be reset to zero. To allow the use of interrupts when filling TX FIFO, bit #9 must be set to 1.
10	DMA is used to fill the controller TX FIFO when performing a write (send) operation. To not allow the use of the DMA to fill the TX FIFO, bit #9 must be reset to zero. To allow the use of the DMA to fill the TX FIFO, bit #9 must be set to 1. The end of transfer can be polled or blocked on an interrupt depending on the setting of bit #9.
16	When bit #16 is reset to zero, it removes the code performing the packing and unpacking of the frames. If packed data is not used in the application, removing the packing code reduce the code size and slightly improve the real-time performance of the SPI driver. To keep the packing code, set bit #16 to 1.

Detailed information on how the different settings of SPI_OPERATION modify the operation of the driver is provided in Sections 3.2, 3.3, and 3.4.

The use of interrupts is controlled by two methods: the bits in the build option SPI_OPERATION and when a slave is initialized, the use of SPI_XFER_POLLING / SPI_XFER_ISR and SPI_EOT_POLLING / SPI_EOT_ISR in the Mode argument of spi_init(). This may seems a redundant way to configure the SPI driver but it allows optimal transfer when multiple and different slaves can be attached to the controller(s). Take the example of an application that needs to communicate with a DAC and an EEPROM. The DAC may only need 4 bytes, therefore using ISRs for the transfer is overkill and CPU inefficient, but the EEPROM may be read by blocks of 512 or 1024 bytes, for which ISRs are the most CPU efficient transfer method. The configuration in SPI_OPERATION is used to either include or exclude the code for the ISR and the spi_init() configuration is used to select if ISRs are the transfer method with a slave or not, when the code is included through SPI_OPERATION.

As for the interrupts, the use of DMA transfers is controlled by two methods: the bits in the build option SPI_OPERATION and when a slave is initialized, the use of SPI_XFER_POLLING / SPI_XFER_ISR / SPI_XFER_DMA.

2.1.8 SPI_ISR_RX_THRS

The build option SPI_ISR_RX_THRS is used to set the threshold, or watermark, at which the data receive interrupts are triggered. When interrupts are used to read from the SPI (SPI_OPERATION bit #1 set to 1), the RX interrupt is triggered when the RX FIFO holds more than a preset number of frames. The build option SPI_ISR_RX_THRS specifies this threshold, in percentage of the FIFO size. Therefore only values between 0 and 100 are accepted for SPI_ISR_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 frames that are transferred on the SPI bus in the latency duration. Assuming a FIFO of 512 bytes with 8-bit frames and assuming an interrupt latency of 2 *u*s with a SPI bus of 80 MHz, then 20 frames are read in 2 *u*s. This optimal threshold is located at 512-20 frames, which is 492 or 96% of 512. In this example, the optimal value to set SPI_ISR_RX_THRS is 96.

Setting this threshold too low has the effect of increasing the number of interrupts and setting it too high will or could provoke overflows of RX FIFO, therefore loss of received frames. Most controller don't offer back-pressure, therefore if the RX FIFO overflows, frames will most likely be lost and the driver will abort transfer.

This build option is ignored if bit #1 in SPI_OPERATION (Section 0) is reset to zero.

2.1.9 SPI_ISR_TX_THRS

The build option SPI_ISR_TX_THRS is used to set the threshold, or watermark, at which the data write interrupt is triggered. When interrupts are used to write to the SPI (SPI_OPERATION bit #9 set to 1), the TX interrupt is triggered when the write FIFO holds less than a preset number of frames. The build option SPI_ISR_TX_THRS specifies this threshold, in percentage of the FIFO size. Therefore only values between 0 and 100 are accepted for SPI_ISR_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 SPI flash memories have a bit of latency between the time the last frame 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.

Setting this threshold too high has the effect of increasing the number of interrupts and setting it too low will or could provoke under-runs of TX FIFO, therefore loss of transmitted frames. Most controller don't offer back-pressure, therefore if the TX FIFO under-runs, the bus transaction will most likely show the end of transfer conditions; due to that, the driver will abort transfer.

This build option is ignored if bit #9 in SPI_OPERATION (Section 0) is reset to zero.

2.1.10 SPI_MIN_4_RX_DMA

The build option SPI_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 frames 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 SPI_OPERATION, if the number of bytes to read is less than the value specified by SPI_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 SPI_OPERATION (Section 0) is reset to zero.

NOTE: DMAs can only perform 8 bit of 16 bit transfers, therefore packed data cannot be used with DMA transfers. The driver ignores the setting for packing when DMA transfers are used

2.1.11 SPI_MIN_4_TX_DMA

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 SPI_OPERATION, if the number of bytes to write is less than the value specified by SPI_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 SPI_OPERATION (Section 0) is reset to zero.

NOTE: DMAs can only perform 8 bit of 16 bit transfers, therefore packed data cannot be used with DMA transfers. The driver ignores the setting for packing when DMA transfers are used

2.1.12 SPI_MIN_4_RX_ISR

The build option SPI_MIN_4_RX_ISR is used to set the minimum number of frames to be read for using 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 frames are to be read, it is highly probable the time required to perform the read itself is less than the overall interrupt overhead. When the RX interrupts are enabled through the build option SPI_OPERATION, if the number of frames to read is less than the value specified by SPI_MIN_4_RX_ISR, the read transfer is performed through polling instead of using interrupts.

This build option is ignored if bit #1 in SPI_OPERATION (Section 0) is reset to zero.

2.1.13 SPI_MIN_4_TX_ISR

The build option SPI_MIN_4_TX_ISR is used to set the minimum number of frames to be written for using 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 frames are to be written, it is highly probable the time required to perform the write itself is less than the overall interrupt overhead. When the TX interrupt are enabled through the build option SPI_OPERATION, if the number of frames to write is less than the value specified by SPI_MIN_4_TX_ISR, the write transfer is performed through polling instead of using interrupts.

This build option is ignored if bit #9 in SPI_OPERATION (Section 0) is reset to zero.

2.1.14 SPI_TOUT_ISR_ENB

The build option `SPI_TOUT_ISR_ENB` is a Boolean controlling if interrupts are shortly re-enabled when checking for timeouts when performing a transfer through polling. When a transfer through polling is performed and the interrupts are disabled during the burst (controlled with `SPI_OPERATION`), this could makes the update of the RTOS timer tick impossible as the timer tick counter is updated through interrupts. The RTOS timer tick won't get updated is the interrupts are disabled on a single core target or when on a multicore target if the core where the driver is operating is the only one handling the RTOS timer tick interrupts. This is one case on a multi-core where it would be advisable to interrupt all core for the RTOS timer tick update.

The setting of `SPI_TOUT_ISR_ENB` does not affect the timeout check when the transfer is interrupt based or DMA based.

2.1.15 SPI_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 `SPI_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 `SPI_REMAP_LOG_ADDR` to zero.

2.1.16 SPI_ARG_CHECK

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

2.1.17 SPI_DEBUG

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

2.1.18 SPI_MULTIPLE_DRIVER

See section 3.5.

3 Operation

This section describes how the SPI 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 SPI parts. The controller is always configured specifically for the target slave it has to access when performing either a receive, a send, or a combined send and receive. By using individual configurations for the slaves, it provides a way to maximize the data transfer performances for all slaves.

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

3.1 Initialization

The initialization is done through the component `spi_init()` (See Section 4.1.1). As most SPI controllers support multiple slaves, which may be different parts, it is necessary to call `spi_init()` for each of the slaves attached to the device. The initialization is fairly straightforward as it simply sets-up the internal device/slave descriptor according to the functionality specified by the arguments to the initialization API. There is a lot of information to specify through the arguments, alike the SPI bus frequency, or the protocol SPI, SSP, µWire, or CPHA and CPOL, etc. Refer to section 4.1.1 for a detailed description on how to use the SPI initialization API.

3.2 Sending

The driver itself does not have any restriction when requested to send frames to a slave on the SPI bus. The send operation is performed using the `spi_send()` component (Section 4.1.3). The port / target controller may have its own limitation, therefore refer to the code for this information. The SPI driver accepts buffers containing the frames to transmit with either one frame per byte/16 bit word, or fully packed data, and the frames can be left or right aligned in the buffer; all this frame formatting is selectable upon initialization for each ones of the controller / slave pairs.

The build option `SPI_OPERATION` (See Section 0) bit #8, bit #9, and bit #10 settings are used to control how the driver operates when sending frames. When bit #8 is set to 1, it configures the driver to disable the interrupts when sending frames. The interrupts are disabled as long as the frame burst hasn't been fully transferred to the TX FIFO of the controller. If bit #8 in `SPI_OPERATION` is instead reset to zero, the interrupts are never disabled when sending the frames. The disabling of the interrupts may be necessary when a controller does not support backpressure, as interrupts and task switching could starve the feeding of the TX FIFO, triggering an abort by the SPI driver of the transfer.

Bit #9 in the build option `SPI_OPERATION`, when set to 1, configures the driver to operate using interrupts when performing a transmit 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.

Bit #10 in the build option `SPI_OPERATION`, when set to 1, configures the driver to operate using DMA transfers when performing a transmit 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.

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 SPI_OPERATION bits #9 and #10, polling, ISR of DMA transfer will be selected according to:

- Assume doing polling
- If `SPI_OPERATION` bit #9 is non-zero and # bytes to write > `SPI_MIN_4_TX_ISR` then use ISRs
- If `SPI_OPERATION` bit #10 is non-zero and # bytes to write > `SPI_MIN_4_TX_DMA` then use DMA
- If `SPI_OPERATION` bit #9 is non-zero, the end of transfer waiting is done blocking on a semaphore posted by the ISR handler.

When the slave to send frames to is a µWire device, the control word must be sent before sending the frames. The information about the control word is located in the buffer holding the frames to send. Refer to Appendix B.

3.3 Receiving

The driver itself does not have any restriction when requested to receive frames from a slave on the SPI bus. The receive operation is performed using the `spi_recv()` component (Section 4.1.2). The port / target controller may have its own limitation, therefore refer to the code for this information. The SPI driver puts the received data in buffers formatted either as one frame per byte/16 bit word, or fully packed data, and the frames can be left or right aligned in the buffer; all this frame formatting is selectable upon initialization for each ones of the controller / slave pairs.

The build option `SPI_OPERATION` (See Section 0) bit #0, bit #1, and bit #2 settings are used to control how the driver operates when performing a read. Bit #0 in `SPI_OPERATION`, when set to 1, configures the driver to disable the interrupts when performing a read (receive) burst. The interrupt disabling lasts as long as the frame 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 `SPI_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 the preset threshold indicated by build option `SPI_ISR_RX_THRS` (Section 2.1.8), an interrupt is generated by the controller. In the interrupt handler, the RX FIFO is completely emptied and when all the frames in the data burst have been read, the interrupt generation is turned off and the semaphore the task was blocked on is posted. There is 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. If the receive watermark (preset threshold) cannot be modified once the transfer has been initiated, then the interrupt posts the semaphore when there is less than the preset threshold of frames left to be received. This is necessary, as the SPI controller will not interrupt the processor due to its contents that will never exceed the threshold.

Bit #2 in the build option `SPI_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 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. For example, a part capable of reading at 100 MHz will fill a 128 frame FIFO in 2.5µs when using 4 lanes to transfer the data. When the read request is less than 64 frames, 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 frames or less will be read around less than 1 µs.

Depending on the setting of the SPI_OPERATION bits #1 and #2, polling, ISR of DMA transfer will be selected according to:

- Assume doing polling
- If `SPI_OPERATION` bit #1 is non-zero and # bytes to read > `SPI_MIN_4_RX_ISR` then use ISRs
- If `SPI_OPERATION` bit #2 is non-zero and # bytes to read > `SPI_MIN_4_RX_DMA` then use DMA
- If `SPI_OPERATION` bit #1 is non-zero, the end of transfer waiting is done blocking on a semaphore posted by the ISR handler.

3.4 Sending and Receiving

The SPI driver supports two types of combined sending and receiving transfers, through the `spi_send_recv()` (Section 4.1.4) component. One transfer is a full-duplex one where frames are sent out at the same time as frames are captured. It is not necessary to have the same number of frames to transmit as the number of frames to receive; either the controller itself deals with this condition or the SPI driver send dummies frames or ignores the extra received frames. The other transfer is a half-duplex one, where frames are sent out first, then frames are captured; this is the way EEPROM / flash memory operates when reading from the memory. If the controller does not support half-duplex (or EEPROM) transfers, the SPI drivers send the necessary dummy frames and ignores the received frames to properly implement the protocol. The selection of the type of transfer is done when initializing the controller / slave pair with `spi_init()` (See 4.1.1). To use half-duplex, the argument Mode should be set to `SPI_TX_RX` and to use the half-duplex mode, it should be set to `SPI_TX_RX_EEPROM`.

The way `spi_send_recv()` operates is a combination of the sending and receiving operation described in the two previous sections. The only difference, depending on the controller, may be the sending of dummy frames and / or the ignoring of received frames.

3.5 Multiple Drivers

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

Table 3-1 BAPI remapping

Original	Multiple
`spi_init()`	`cd_spi_init()`
`spi_recv()`	`cd_spi_recv()`
`spi_send()`	`cd_spi_send()`
`spi_set()`	`cd_spi_set()`
`spi_send_recv()`	`cd_spi_send_recv()`
`SPIintHndl_#()`	`cd_SPIintHndl_#()`

The prefix is always the prefix in the file name; e.g. `cd_spi.c` prefix is “cd” and `dw_spi.c` is “dw”.

All build options if not prefixed apply to all the drivers. To set build options on a per-driver basic all 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 `SPI_MAX_DEVICES` should defined and set to 3. If the `cd_spi` and the `dw_spi` drivers are use together and, as for the example below, there are 2 devices used for the `cd` and 3 used for the `dw`, then `CD_SPI_MAX_DEVICES` should defined and set to 2 and `DW_SPI_MAX_DEVICES` should defined and set to 3. When a driver specific build option is defined then the driver for which the build option applies ignores the equivalent global build option.

A custom wrapper must be provided. The following code shows such a driver for the `cd_spi` (2 SPIs accessed as device #0 and #1, mapped to cd’s #01 and #1) and the `dw_spi` (3 SPIs accessed as device #2, #3, and #4, mapped as dw’s #0, #1, and #2). The example uses contiguous device number for cd (#0 and #1) and for dw (#0, #1, and #2) as it’s easier to show the “how to”. Non-contiguous and non starting from #0 devices can also be used by implementing the appropriate input–output device remapping.

Table 3-2 Multiple SPI wrapper example (`spi.c`)

```
#include "spi.h"

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

int spi_init(int Dev, int Slv, int Freq, int Bits, uint32_t Mode)
{
int RetVal;

   RetVal = (Dev < 2)
          ? cd_spi_init(Dev,   Slv, Freq, Bits, Mode)
          : dw_spi_init(Dev-2, Slv, Freq, Bits, Mode);

   return(RetVal);
}

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

int spi_recv(int Dev, int Slv, char *Buf, uint32_t Len)
{
int RetVal;

   RetVal = (Dev < 2)
          ? cd_spi_recv(Dev,   Slv, Buf, Len)
          : dw_spi_recv(Dev-2, Slv, Buf, Len);

   return(RetVal);
}


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

int spi_send(int Dev, int Slv, char *Buf, uint32_t Len)
{
int RetVal;

   RetVal = (Dev < 2)
          ? cd_spi_send(Dev,   Slv, Buf, Len)
          : dw_spi_send(Dev-2, Slv, Buf, Len);

   return(RetVal);
}

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

int spi_set(int Dev, int Slv, int Type, int Val)
{
int RetVal;

   RetVal = (Dev < 2)
          ? cd_spi_set(Dev,   Slv, Type, Len)
          : dw_spi_set(Dev-2, Slv, Type, Val);

   return(RetVal);
}

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

int spi_send_recv(int Dev, int Slv, const char *BufTX, uint32_t LenTX, char *BufRX,
                  uint32_t LenRX)
{
int RetVal;

   RetVal = (Dev < 2)
```

```
          ? cd_spi_send_recv(Dev,   Slv, BufTx, LenTx, BufTX, LenRX)
          : dw_spi_send_recv(Dev-2, Slv, BufTx, LenTx, BufTX, LenRX);

   return(RetVal);
}

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

void SPIintHndl_0(void) {cd_SPIintHndl_0(); }
void SPIintHndl_1(void) {cd_SPIintHndl_1(); }
void SPIintHndl_2(void) {dw_SPIintHndl_0(); }
void SPIintHndl_3(void) {dw_SPIintHndl_1(); }
void SPIintHndl_4(void) {dw_SPIintHndl_2(); }

/* EOF */
```

Table 3-3 Multiple SPI wrapper example (`spi.h`)

```
#ifndef __SPI_H__
#define __SPI_H__     1

#include "cd_spi.h"                             /* cd SPI driver                    */
#include "dw_spi.h"                             /* dw SPI driver                    */

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

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

#define SPI_MAX_DEVICES  ((CD_SPI_MAX_DEVICES)+(DW_SPI_MAX_DEVICES))
#define SPI_LIST_DEVICE  ((CD_SPI_LIST_DEVICE)|((DW_SPI_LIST_DEVICE)<<2))

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

int spi_init     (int Dev, int Slv, int Freq, int Bits, uint32_t Mode);
int spi_recv     (int Dev, int Slv, void *Buf, uint32_t Len);
int spi_send     (int Dev, int Slv, const void *Buf, uint32_t Len);
int spi_set      (int Dev, int Slv, int Type, int Val);
int spi_send_recv(int Dev, int Slv, const void *BufTX, uint32_t LenTX, void *BufRX,
                  uint32_t LenRX);

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

extern void SPIintHndl_0(void);
extern void SPIintHndl_1(void);
extern void SPIintHndl_2(void);
extern void SPIintHndl_3(void);
extern void SPIintHndl_4(void);

#endif

/* EOF */
```

4 API

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

4.1.1 spi_init

Synopsis

```
#include "???_spi.h"

int spi_init(int Dev, int Slv, int Freq, int Nbits, int Mode);
```

Description

`spi_init()` is the component used to initialize a SPI device, specified with the device indicated by the argument `Dev`, and one slave attached to that device; the slave is identified with the argument `Slv`. The maximum SPI bus frequency is specified by the argument `Freq`, the number of bits per frames is specified by the argument `Nbits` and the mode of operation trough the argument `Mode`. The argument `Mode` is used to hold a lot of information, refer to the **Options** sub-section for more information on `Mode`. An already initialized slave can be re-initialized at any time.

Arguments

`Dev`	Device to initialize (Number starting at 0)
`Slv`	Slave to initialize (Number starting at 0)
`Freq`	Desired SPI bus frequency
`Nbits`	Number of bites per frame
`Mode`	Mode of operation, refer to the **Options** section

Returns

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

Component type

Function

Options

The argument `Mode` is used to set-up almost everything on how the controller, specified by the argument `Dev`, communicates with the Slave indicated by the argument `Slv`. The following table shows all the supported settings, which must be binary ORed together for the value set for `Mode`. It is important to always construct the value use for `Mode` by using one and only one token from **all** groups; if all tokens from a group are missing, the result is target specific. The groups are shown in the table boxed inside a wide border and only one of the token in the group can be used.

Table 4-1 Mode OR'ed values

Defined token	Description
SPI_PROTO_SPI	The transfer protocol is the original SPI (Motorola)
SPI_PROTO_SSP	The transfer protocol is Texas Instruments' Serial Protocol (SSP)
SPI_PROTO_UWIRE	National Semiconductors's µ-Wire ptotocol
SPI_CLK_CPHA0	Data is captured (read) on the inactive to active clock transition
SPI_CLK_CPHA1	Data is captured (read) on the active to inactive clock transition
SPI_CLK_CPOL0	Idle state (inactive) of the clock is 0
SPI_CLK_CPOL1	Idle state (inactive) of the clock is 1
SPI_ALIGN_LEFT	The frames in the buffers passed to spi_recv(), spi_send(), and spi_send_recv() are left aligned in the buffer.
SPI_ALIGN_RIGHT	The frames in the buffers passed to spi_recv(), spi_send(), and spi_send_recv() are right aligned in the buffer.
SPI_DATA_NONPACK	The frames in the buffers passed to spi_recv(), spi_send(), and spi_send_recv() are one frame per byte/16 bit words.
SPI_DATA_PACK	The frames in the buffers passed to spi_recv(), spi_send(), and spi_send_recv() are packed. Ignored, and SPI_DATA_NONPACK is used instead, if the build option SPI_OPERATION bit #16 is reset to 0.
SPI_TX_RX	When spi_send_recv() is used, both transmission and reception are going on at the same time. It is a full-duplex transfer.
SPI_TX_RX_EEPROM	When spi_send_recv() is used, frames are transmitted first and frames are received. It is a half-duplex transfer.
SPI_MASTER	The controller is the bus master.
SPI_SLAVE	The controller is a slave on the bus.
SPI_XFER_POLLING	No transfers use interrupts, polling is used.
SPI_XFER_ISR	All transfers use interrupts. When receiving frames, this is ignored and SPI_XFER_POLLING is used instead if the build option SPI_OPERATION bit #1 is reset. When transmitting frames, this is ignored and SPI_XFER_POLLING is used instead if the build option SPI_OPERATION bit #9 is reset.
SPI_XFER_DMA	All transfers use DMA. When receiving frames, this is ignored and SPI_XFER_POLLING is used instead if the build option SPI_OPERATION bit #2 is reset. When transmitting frames, this is ignored and SPI_XFER_POLLING is used instead if the build option SPI_OPERATION bit #10 is reset.
SPI_EOT_POLLING	When transmitting frames, uses polling to detect when all frames have been transferred.

SPI_EOT_ISR	When transmitting frames, uses an interrupt to detect when all frames have been transmitted. This is ignored and SPI_EOT_POLLING is used instead if the build option SPI_OPERATION bit #9 is reset.
SPI_UWIRE_HS	When SPI_PROTO_UWIRE is used in Mode, makes the transfer use the µWire handshake. It is a "don't care" if SPI_PROTO_UWIRE is not selected in Mode.
SPI_UWIRE_NO_HS	When SPI_PROTO_UWIRE is used in Mode, makes the transfer to not use the µWire handshake. It is a "don't care" if SPI_PROTO_UWIRE is not selected in Mode.

Notes

When multiple slaves are connected on a device, `spi_init()` must be called using the same device number but with a different slave number to initialize according to the slave on that device.

See Also

`spi_recv` (Section 4.1.2)
`spi_send` (Section 4.1.3)
`spi_send_recv` (Section 4.1.3)

4.1.2 spi_recv

Synopsis

```
#include "???_spi.h"

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

Description

`spi_recv()` is the component used to receive frames from a SPI device. The part is accessed using the controller specified by the argument `Dev`, and the slave number on that device; the slave number is identified with the argument `Slv`. The number of frames to receive is indicated by the argument `Len`, and the received frames are deposited in the buffer indicated by the argument `Buf`. The frames are deposited in the receive buffer `Buf` according to the setting of `Mode` in `spi_init()`. The tokens `SPI_ALIGN_LEFT`, `SPI_ALIGN_RIGHT`, `SPI_DATA_PACK` and `SPI_DATA_NONPACK` control the way the frames lands in `Buf`. Refer to Appendix A (Section 5) for more information.

Arguments

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

Returns

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

Component type

Function

Options

The build option `SPI_OPERATION` (Section 0) controls the behavior of the driver when a read operation is requested. When `SPI_OPERATION` bit #0 is set to 1, interrupts are disabled during each read burst. When bit #1 is set to 1, and `SPI_XFER_ISR` is set in Mode during initialization, interrupts can be used to transfer the data from the RX FIFO of the controller to the application memory.

Notes

When the transfer is set-up to use the µWire protocol, the first 4 bytes of `Buf` must be properly set-up to provide the driver with the control word used by µWire. Refer to Appendix B (Section 6) for the required values of these 4 bytes.

See Also

`spi_init` (Section 4.1.1)
`spi_send` (Section 4.1.3)
`spi_send_recv` (Section 4.1.4)

4.1.3 spi_send

Synopsis

```
#include "???_spi.h"

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

Description

`spi_send()` is the component used to send frames to a SPI device. The part is accessed using the controller specified by the argument `Dev`, and the slave number on that device; the slave number is identified with the argument `Slv`. The number of frames to send is indicated by the argument `Len`, and the frames sent are retrieved from the buffer indicated by the argument `Buf`. The frames in the transmit buffer `Buf` must be formatted according to the setting of `Mode` in `spi_init()`. The tokens `SPI_ALIGN_LEFT`, `SPI_ALIGN_RIGHT`, `SPI_DATA_PACK` and `SPI_DATA_NONPACK` control the way the frames land in `Buf`. Refer to Appendix A (Section 5) for more information.

Arguments

`Dev`	Device number (Number starting at 0)
`Slv`	Slave to write to (Number starting at 0)
`Buf`	Frames to send
`Len`	Number of frames to send

Returns

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

Component type

Function

Options

The build option `SPI_OPERATION` (Section 0) controls the behavior of the driver when a send operation is requested. When `SPI_OPERATION` bit #8 is set to 1, interrupts are disabled during each write burst. When bit #9 is set to 1, and `SPI_XFER_ISR` is set in Mode during initialization, interrupts can be used to transfer the data from the application memory to the TX FIFO of the controller.

Notes

When the transfer is set-up to use the µWire protocol, the first 4 bytes of `Buf` must be properly set-up to provide the driver with the control word used by µWire. Refer to Appendix B (Section 6) for the required values of these 4 bytes.

See Also

`spi_init` (Section 4.1.1)
`spi_recv` (Section 4.1.2)
`spi_send_recv` (Section 4.1.4)

4.1.4 spi_send_recv

Synopsis

```
#include "???_spi.h"

int spi_send_recv(int Dev, int Slv, const void *BufTX,
                  uint32_t LenTX, void *BufRX, uint32_t LenRX);
```

Description

`spi_send_recv()` is the component used to read and write data to a SPI slave. The part is accessed using the controller specified by the argument `Dev`, and the slave number on that device; the slave number is identified with the argument `Slv`. The number of frames to send is indicated by the argument `LenTX`, and the frames sent are retrieved from the buffer indicated by the argument `BufTX`. The number of frames to receive is indicated by the argument `LenRX`, and the frames recived are deposited in the buffer indicated by the argument `BufRX`. The frames in the buffers `Buf` and the frames deposited in the buffer `BufRX` are formatted according to the setting of `Mode` in `spi_init()`. The tokens `SPI_ALIGN_LEFT`, `SPI_ALIGN_RIGHT`, `SPI_DATA_PACK` and `SPI_DATA_NONPACK` control the buffer formatting. Refer to Appendix A (Section 5) for more information.

The transfer performed by `spi_send_recv()` can be either half-duplex or full-duplex. A half-duplex transfer send `LenTX` frames on the SPI bus and the receive `LenRX` frames from the SPI bus. A full-duplex transfer sends and receives frames at the same time, and the duration of the transfer is the largest value between `LenTX` and `LenRX`. The full-duplex is used when `spi_init()` Mode is set with `SPI_TX_RX` and the half-duplex is used when Mode is set with `SPI_TX_RX_EEPROM`.

Arguments

`Dev`	Device number (Number starting at 0)
`Slv`	Slave to write to (Number starting at 0)
`BufTX`	Base address of the buffer holding the frames to transmit
`LenTX`	Number of frames to send on the bus
`BufRX`	Base address of the buffer where the received frames are deposited
`LenRX`	Number of frames to receive from the bus

Returns

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

Component type

Function

Options

The build option `SPI_OPERATION` (Section 0) controls the behavior of the driver when a read operation is requested. When `SPI_OPERATION` bit #0 is set to 1, or bit #8 is set to 1 interrupts are disabled during the frame exchange burst. When bit #9 is set to 1, and `SPI_XFER_ISR` is set in Mode during initialization, interrupts can be used to transfer the data from the application memory to the TX FIFO of the controller. When bit #1 is set to 1, and `SPI_XFER_ISR` is set in Mode during initialization, interrupts can be used to transfer the data from the RX FIFO of the controller to the application memory.

Notes

`spi_send_recv()` cannot be used when the transfer protocol is set to µWire, i.e. when `Mode` in `spi_init()` is set to `SPI_PROTO_UWIRE`.

See Also

`spi_init` (Section 4.1.1)
`spi_recv` (Section 4.1.2)
`spi_send` (Section 4.1.3)

4.1.5 SPIintHndl_*n*

Synopsis

```
#include "???_spi.h"

void SPIintHndl_n(void);
```

Description

`SPIintHndl_n()` is the interrupt handler for the SPI 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 SPI interrupt handler is not used, therefore not needed, if bit #1 and bit #9 of the build option `SPI_OPERATION` are both set to 0 (Section 0). Interrupt handlers are only available for the SPI devices validated by the build option `SPI_LIST_DEVICE`.

Notes

The interrupt handler should always be attached to the targeted SPI 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 SPI driver for this device will not operate.

See Also

`spi_recv` (Section 4.1.2)
`spi_send` (Section 4.1.3)
`spi_send_recv` (Section 4.1.4)

5 Appendix A

This appendix shows graphically all the types of frame buffering supported by the SPI driver. The first and foremost important characteristic is the SPI driver supports from 1 to 16 bit frames. When the number of bits per frame specified during initialization is less or equal to 8 bits, the buffered frames are held in byte buffers. If the number of bits per frames is greater than 8, then the buffered frames are held in 16 bits buffers. It was necessary to proceed with 8 bit and 16 buffers as processor memory is organized in either little or big endian.

All examples show a byte buffering using 5 bit frames and a 16 bit buffering using 11 bit frames. The numbers in the table are the frame number and `x`'s are don't care bits. All frames have their MSBit on the left (higher bit index in the byte or word) and the MSBits are always the first bit sent/received on the SPI bus. The MSBit of the frame, which is the first bit of the frame transferred on the bus, is always on the left no matter the data alignment chosen.

5.1 5 bit frame: left aligned non-packed

`spi_init()` arguments:

```
Nbits : 5
Mode  : SPI_ALIGN_LEFT | SPI_DATA_NONPACK | …
```

Table 5-1 Left-aligned non-packed 5 bit frame

Byte\Bit	7	6	5	4	3	2	1	0
0	0	0	0	0	0	x	x	x
1	1	1	1	1	1	x	x	x
2	2	2	2	2	2	x	x	x

5.2 11 bit frame: left aligned non-packed

`spi_init()` arguments:

```
Nbits : 11
Mode  : SPI_ALIGN_LEFT | SPI_DATA_NONPACK | …
```

Table 5-2 Left-aligned non-packed 11 bit frame

Word\Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	x	x	x	x	x
1	1	1	1	1	1	1	1	1	1	1	1	x	x	x	x	x
2	2	2	2	2	2	2	2	2	2	2	2	x	x	x	x	x

5.3 5 bit frame: right aligned non-packed

`spi_init()` arguments:

```
Nbits : 5
Mode  : SPI_ALIGN_RIGHT | SPI_DATA_NONPACK | …
```

Table 5-3 Right-aligned non-packed 5 bit frame

Byte\Bit	7	6	5	4	3	2	1	0
0	x	x	x	0	0	0	0	0
1	x	x	x	1	1	1	1	1
2	x	x	x	2	2	2	2	2

5.4 11 bit frame: right aligned non-packed

`spi_init()` arguments:

```
Nbits : 11
Mode  : SPI_ALIGN_RIGHT | SPI_DATA_NONPACK | …
```

Table 5-4 Right-aligned non-packed 11 bit frame

Word\Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	x	x	x	x	x	0	0	0	0	0	0	0	0	0	0	0
1	x	x	x	x	x	1	1	1	1	1	1	1	1	1	1	1
2	x	x	x	x	x	2	2	2	2	2	2	2	2	2	2	2

5.5 5 bit frame: left aligned packed

`spi_init()` arguments:

```
Nbits : 5
Mode  : SPI_ALIGN_LEFT | SPI_DATA_PACK | …
```

Table 5-5 Left-aligned packed 5 bit frame

Byte\Bit	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	1
1	1	1	2	2	2	2	2	3
2	3	3	3	3	4	4	4	4

5.6 11 bit frame: left aligned packed

`spi_init()` arguments:

```
Nbits : 11
Mode  : SPI_ALIGN_LEFT | SPI_DATA_PACK | …
```

Table 5-6 Left-aligned packed 11 bit frame

Word\Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2
2	2	3	3	3	3	3	3	3	3	3	3	3	4	4	4	4

5.7 5 bit frame: right aligned packed

`spi_init()` arguments:

```
Nbits : 5
Mode  : SPI_ALIGN_REIGH | SPI_DATA_PACK | …
```

Table 5-7 Right-aligned packed 5 bit frame

Byte\Bit	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0	0
1	3	2	2	2	2	2	1	1
2	4	4	4	4	3	3	3	3

5.8 11 bit frame: right aligned packed

`spi_init()` arguments:

```
Nbits : 11
Mode  : SPI_ALIGN_REIGH | SPI_DATA_PACK | …
```

Table 5-8 Right-aligned packed 11 bit frame

Word\Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
1	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1
2	4	4	4	4	3	3	3	3	3	3	3	3	3	3	3	2

6 Appendix B

The µWire protocol from National Semiconductor uses a control word at the beginning of all bus transactions. A key thing is this control word does not necessary have the same number of bits as the frames have. As only µWire requires a control word, it was decided to provide the control word and its number of bits through the read or write buffers passed to `spi_recv()` and `spi_send()`. The µWire control word information is always held in the first 4 bytes of the buffer, providing the number of bits information and the control word itself. When receiving frames from the SPI bus, the µWire 4 bytes are overwritten with the received frames. When sending frames on the SPI bus, the frames to send must be located immediately after these four bytes in the transmit buffer.

The µWire control word information is indicated in the following table:

Table 6-1 µWire Control Word

Byte index	Description
`#0`	Number of bits in the control word
`#1`	Ignored
`#2`	MSBits of the control word, if more than 8 bits
`#3`	LSBits of the control word

Although the SPI driver supports left or right aligned frames, the µWire control word (bytes #2 and #3) is always right aligned in the bytes. The reason there is a “don’t care” byte is due the possibility the buffer holds 16 bits words when the number of bits per frame is greater than 8. As many processors require data alignment, adding a “don’t care” byte was necessary to retain the buffer alignment.

7 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 – QSPI Flash Memory Support, available at http://www.code-time.com
[R5] Abassi RTOS – System Calls Layer, available at http://www.code-time.com

8 Revision History

Date	Version	Author/Editor	Description
2016.09.03	1.1	EV	First Draft
2016.09.19	1.2	AP	Review changes
2016.11.21	1.3	EV	Added `SPI_POLLING_TOUT`
2016.11.23	1.4	AP	Review changes
2017.03.22	1.5	EV	Small changes
2017.05.30	1.6	EV	Full clean-up
2017.08.31	1.7	EV	Update
2018.08.17	1.8	EV	Added multiple drivers
2018.08.28	1.9	EV	Small changes
2018.11.21	1.10	EV	Alternate Token naming