Contents

[2 Document History 1](#_Toc378777518)

[3 Preface 1](#_Toc378777519)

[4 Main design decisions 2](#_Toc378777520)

[5 The API 2](#_Toc378777521)

[5.1 Data structures/types 2](#_Toc378777522)

[5.1.1 Instance handle structure 2](#_Toc378777523)

[5.1.2 Instance initialization structure 2](#_Toc378777524)

[5.1.3 Transfer completion callback 3](#_Toc378777525)

[5.2 API common for both master and slave 3](#_Toc378777526)

[5.3 API for master transfers 4](#_Toc378777527)

[5.4 API for slave transfers 5](#_Toc378777528)

# Document History

|  |  |  |
| --- | --- | --- |
| Jan 10 2014 | Kjell Eirik Andersen | Initial revision. |
| Jan 20 2014 | Kjell Eirik Andersen | Updates after spec. review.   * Removed txValue from receive functions. The value to transmit in receive only functions will be specified when initializing the driver. * Added GetBaudrate() function. * Claryfied slave transfer function functionality if called in the middle of a master transfer. * New TransferStatus() function reports both items transmitted and items remaining. |
| Jan 23 2014 | Kjell Eirik Andersen | Changes baudrate to bitrate. |
| Jan 29 2014 | Kjell Eirik Andersen | Added Set/GetFramelength() functions. |

# Preface

This document summarize the discussion over a new SPI (**S**erial **P**eripheral **I**nterface) API which has been ongoing within the 32bit Embedded MCU software group in the second half of December 2013.

# Main design decisions

The API is supposed to be usable over several chip families/types. As different chips will have different capabilities it is acceptable that the init function can have different signatures in different implementations of the API.

* The API supports both slave and master mode.
* Only one SPI transfer can exist at a time.
* The transfer functions does not buffer transferred data. The application will provide transfer buffers. These buffers must stay valid for the duration of the transfer.
* Transfers are made using DMA.
* Transfer functions come in two flavor’s; blocking (B at end of function name) and non-blocking with optional callback.
* Several driver instances can coexist.

Note: All API functions returns Ecode\_t. Refer to the coding standard for general information about e-codes.

The API will also handle chip selects (CS) when the underlying SPI hardware permits, i.e. on an EFM32 the SPI master has one chip select which may be automatically driven by the SPI peripheral. The auto function can be initialized to on or off when initializing a driver instance. When the auto function is turned off and in cases when the SPI peripheral does not support chip selects, the application itself can provide the necessary chip select functionality using gpio’s.

# The API

## Data structures/types

### Instance handle structure

As a chip may have several physical SPI ports, it is possible to instantiate several SPI drivers. Each instance is identified by its ***handle***. The handle is a pointer to a data structure containing all necessary data and/or “states” needed for one instance.

The handle is allocated by the application and passed to the SPI driver when calling the initialization function. Even though the application code has full access to the handle data, the application shall not read or write any of its content. All SPI API functions take a handle pointer as its first argument.

### Instance initialization structure

When initializing an instance of the SPI driver, an initialization structure is passed to the init function. This structure contains all initialization data needed such as bus bitrate, framelength/itemsize, clocking phase and polarity etc. Default initializers for the most common use cases will be predefined in the API header file.

The layout of the initialization structure may vary depending on the chip used.

### Transfer completion callback

typedef void (\*SPIDRV\_Callback\_t)( SPIDRV\_Handle\_t \*handle,

Ecode\_t transferStatus,

int itemsTransferred );

## API common for both master and slave

The SPI init function initializes everything necessary to make the SPI port functional. Driver instance handle and initialization data structures are discussed in the previous section.

// Initialize SPI, setup clocks, SPI hw and init internal book-keeping data

Ecode\_t SPIDRV\_Init( SPIDRV\_Handle\_t \*handle, SPIDRV\_Init\_t initData );

The deinit function will deinitialize the SPI hardware, turn off clocks etc.

// Disable SPI peripheral, clocks etc.

Ecode\_t SPIDRV\_DeInit( SPIDRV\_Handle\_t \*handle );

The status function is used to check number of items transferred/remaining so far.

// Check progress of ongoing transfer

Ecode\_t SPIDRV\_GetTransferStatus( SPIDRV\_Handle\_t \*handle,

int \*itemsTransferred,

int \*itemsRemaining );

// Abort ongoing transfer

Ecode\_t SPIDRV\_AbortTransfer( SPIDRV\_Handle\_t \*handle );

// Set new SPI framelength

Ecode\_t SPIDRV\_SetFramelength( SPIDRV\_Handle\_t \*handle, int frameLength );

// Get SPI framelength

Ecode\_t SPIDRV\_GetFramelength( SPIDRV\_Handle\_t \*handle, int \*frameLength );

## API for master transfers

Transfer functions are either blocking (trailing B in function name) or non-blocking with completion callback. Transfer data buffers are void pointers to accommodate for implementations with bus framelength (item size) larger than a byte (an EFM32 supports 4-16 bit framelengths).

// Set new SPI bitrate

Ecode\_t SPIDRV\_SetBitrate( SPIDRV\_Handle\_t \*handle, int bitRate );

// Get SPI bitrate

// (can be used to check if requested bitrate is possible)

Ecode\_t SPIDRV\_GetBitrate( SPIDRV\_Handle\_t \*handle, int \*bitRate );

// Transmit data (while ignoring received data)

Ecode\_t SPIDRV\_MTransmit( SPIDRV\_Handle\_t \*handle,

const void \*buffer,

int count,

SPIDRV\_Callback\_t \*callback );

Ecode\_t SPIDRV\_MTransmitB( SPIDRV\_Handle\_t \*handle,

const void \*buffer,

int count );

// Receive data (while sending a fixed txValue (usually 0 or FF))

// The txValue is specified when initializing the driver.

Ecode\_t SPIDRV\_MReceive( SPIDRV\_Handle\_t \*handle,

void \*buffer,

int count,

SPIDRV\_Callback\_t \*callback );

Ecode\_t SPIDRV\_MReceiveB( SPIDRV\_Handle\_t \*handle,

void \*buffer,

int count );

// Transmit and receive data

Ecode\_t SPIDRV\_MTransfer( SPIDRV\_Handle\_t \*handle,

const void \*txBuffer,

void \*rxBuffer,

int count,

SPIDRV\_Callback\_t \*callback );

Ecode\_t SPIDRV\_MTransferB( SPIDRV\_Handle\_t \*handle,

const void \*txBuffer,

void \*rxBuffer,

int count );

// Do one transfer

Ecode\_t SPIDRV\_MTransferSingleItemB( SPIDRV\_Handle\_t \*handle,

uint32\_t txValue,

uint32\_t \*rxValue );

## API for slave transfers

The slave transfer API is similar to the master API with the exception of the timeout parameter. A timeout of zero is treated as no timeout.

NOTE: Slave transfers will not start if master is in the middle of a transfer. This situation is detected by checking the level on the chipselect line. When the ongoing transfer has completed the SPI hw will be armed.

// Transmit data (while ignoring received data)

Ecode\_t SPIDRV\_STransmit( SPIDRV\_Handle\_t \*handle,

const void \*buffer,

int count,

SPIDRV\_Callback\_t \*callback,

int timeoutMs );

Ecode\_t SPIDRV\_STransmitB( SPIDRV\_Handle\_t \*handle,

const void \*buffer,

int count,

int timeoutMs );

// Receive data (while sending a fixed txValue (usually 0 or FF))

// The txValue is specified when initializing the driver.

Ecode\_t SPIDRV\_SReceive( SPIDRV\_Handle\_t \*handle,

void \*buffer,

int count,

SPIDRV\_Callback\_t \*callback,

int timeoutMs );

Ecode\_t SPIDRV\_SReceiveB( SPIDRV\_Handle\_t \*handle,

void \*buffer,

int count,

int timeoutMs );

// Transmit/Receive data

Ecode\_t SPIDRV\_STransfer( SPIDRV\_Handle\_t \*handle,

const void \*txBuffer,

void \*rxBuffer,

int count,

SPIDRV\_Callback\_t \*callback,

int timeoutMs );

Ecode\_t SPIDRV\_STransferB( SPIDRV\_Handle\_t \*handle,

const void \*txBuffer,

void \*rxBuffer,

int count,

int timeoutMs );