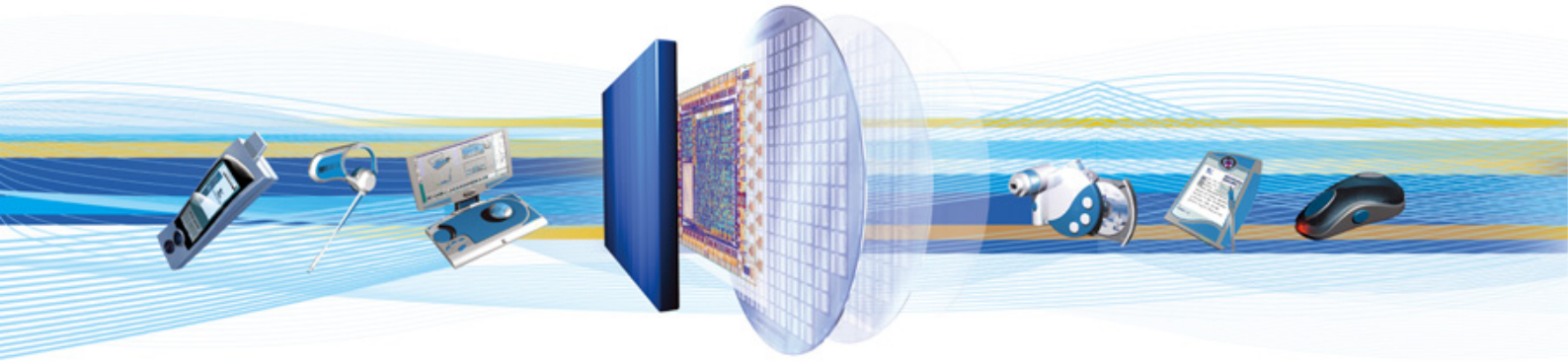
CSR Synergy Framework 3.1.0

SDIO

API Description

August 2011



Cambridge Silicon Radio Limited

Churchill House
Cambridge Business Park
Cowley Road
Cambridge CB4 0WZ
United Kingdom

Registered in England and Wales 3665875

Tel: +44 (0)1223 692000
Fax: +44 (0)1223 692001
www.csr.com

Contents

List of tables:

List of figures:

1 Introduction

1.1 Introduction and Scope

This document describes the requirements and functionality for implementations of the CSR Synergy Framework SDIO interface. It is the intention that this document can serve as the foundation for a complete and compliant implementation that will enable generic Function Drivers such as the SDIO Type-A Bluetooth driver provided with the framework.

The function prototypes, which are required for complying with the requirements and functionality, described in this document can be found in `csr_sdio.h`.

1.2 Prerequisites

This document assumes that the reader has a fundamental and thorough understanding of the SDIO Specification [SDIO-SPEC].

2 CSR SDIO Interface

2.1 Introduction

Figure 1 illustrates the SDIO API/interface (denoted by the line containing csr_sdio.h) as it is implemented in the BDB2 BSP. It is up to the readers to implement this interface with respect to their own platform. The BDB2 BSP provides two different implementations for the SDIO API to showcase how different transports (SDIO and CSPI) implement the same interface. These two transports share some common code and in order to avoid duplication, the SDIO API has been implemented using a combination of platform independent code (CSR SDIO) and platform dependant code (SDIO/SPI Master Driver). This is an implementation choice and the reader may choose to do the same.

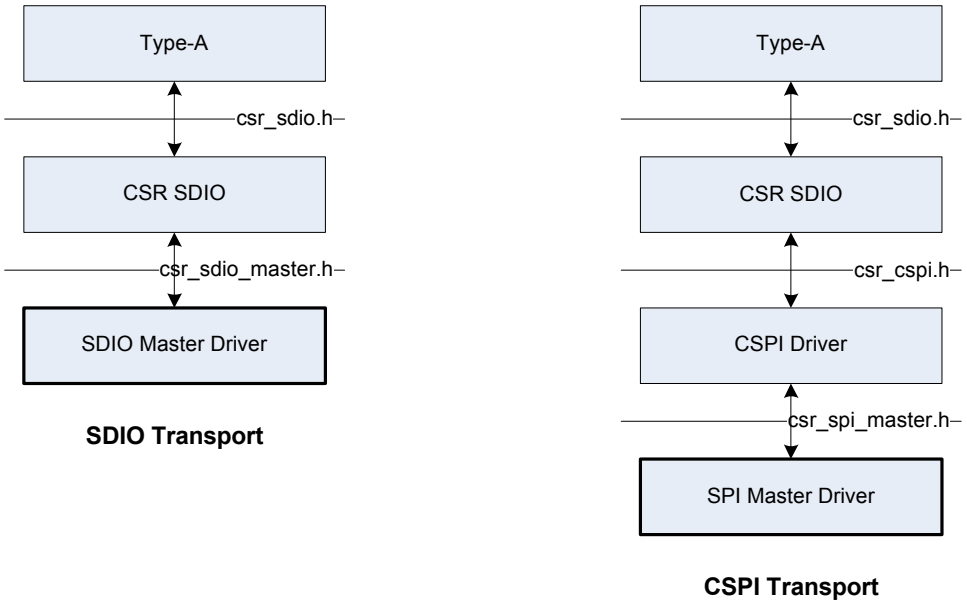


Figure 1: Illustration of SDIO interface and the BDB2 BSP implementation

This document will only describe the SDIO API. For more implementation specific details regarding csr_sdio_master.h, csr_cspi.h and csr_spi_master.h, see the header files in the BDB2 BSP.

2.2 Definitions

The following definitions are used in this document:

Definition	Meaning
Device	A Device is a physical device connected over either SDIO or CSPI.
Device Function	A Device consists of a number of logical units, each of which is a Device Function. Device Functions are numbered from 0 to 7. Device Function 0 is a special control unit that provides access to the Device, whereas Device Functions 1-7 are normal units that may expose interfaces such as Type-A.

Definition	Meaning
Function Driver	A driver for a Device Function. A Function Driver may support multiple different Device Functions, and it may be controlling multiple Device Functions at the same time. When a Function Driver is controlling a Device Function, it is said to be ``attached'' to the Device Function. When a Function Driver is attached to a Device Function, it has exclusive access to it.
Function Handle	A Function Handle is a data structure that is used by Function Drivers to identify a particular Device Function.
Device Specification	A set of parameters a Device Function (and its corresponding Device) must match in order for a given Function Driver to attempt to attach to it. The possible parameters are vendor ID, Device ID, function number, and SDIO Standard Function interface code. A Function Driver may specify a wildcard value for parameters for which it has no preference or requirements.
Device Event	Either Device Insertion or Device Removal.
Device Insertion	A Device Insertion is an event that is signalled to a Function Driver when a Device Function that fulfils the Device Specification of the Function Driver has become available.
Device Removal	A Device Removal is an event that is signalled to a Function Driver when the Device Function to which it is attached has become unavailable.
Device Interrupt	A Device Interrupt is an event that is signalled to Function Drivers when the interrupt line for the Device for the Device Function is asserted.

2.3 General overview

This section provides a general overview of how a Function Driver (e.g. Type-A) interacts with the SDIO implementation. This section may omit many details to keep this explanation simple as possible. All the missing details can be found in the later sections where each function is described.

From the point of view of the Function Driver, the following steps are required to establish communication with a specific Device Function of an SDIO/CSPI Device.

1. Define a Function Driver supporting one or more Device Functions of an SDIO/CSPI Device.
2. Register the Function Driver with CSR SDIO using the CsrSdioFunctionDriverRegister call.
3. When one or more supported Device Functions become available, CSR SDIO signals the Function Driver. The Device Functions are then said to be attached to the Function Driver. A particular Device Function can only be attached to one Function Driver, while one Function Driver can have several attached Device Functions.

4. When one or more attached Device Functions become unavailable, CSR SDIO signals the Function Driver.

5. If needed, a Function Driver can be unregistered at any time, regardless of whether any Device Functions are currently attached, using the CsrSdioFunctionDriverUnregister call.

2.3.1 Defining a Function Driver

A Function Driver consists of a CsrSdioFunctionDriver struct instance, a specification of supported devices, and some associated callback functions for which function pointers are stored in the CsrSdioFunctionDriver struct instance:

```
typedef struct
{
    CsrSdioInsertedCallback     inserted;
    CsrSdioRemovedCallback      removed;
    CsrSdioInterruptCallback    intr;
    CsrSdioSuspendCallback      suspend;
    CsrSdioResumeCallback       resume;
    CsrSdioFunctionId           *ids;
    CsrUint8                    idsCount;
CsrSdioFunctionDriver;

}
```

Function Driver callbacks

The callback functions provide the entry points to a Function Driver for CSR SDIO. They are called by CSR SDIO whenever a Device Event occurs, when a Device Interrupt is detected, or when CSR SDIO needs to inform the Function Drivers of system power events. These events are asynchronous and must be acknowledged by the Function Driver as explained later.

Two types of Device Events can occur, Device Insertion and Device Removal, and only in this order for a given Device Function. The insertion callback is called to notify a Function Driver that it has exclusive access to a supported Device Function that has become available. This event does not solely happen on Device Insertion and thus does not necessarily mean that the Device was inserted at this point but simply that a given Device Function is now available to a given Function Driver. For instance, this may be caused by a permanently inserted Device that has now become available to a Function Driver simply on account of the Function Driver being registered. Similarly, the removal callback does not necessarily indicate that the Device has been removed but merely that the Device Function is no longer available to the Function Driver. This could be caused by both Device Removal and unregistration of the Function Driver.

If a Function Driver supports several different Device Functions, it will receive one callback for each Device Function that becomes available. The remove callback will also be signalled for each attached Device Function that becomes unavailable.

Whenever a Device performs a Device Interrupt, this is signalled the Function Drivers via their interrupt callback function as soon as possible. When the interrupt callback is called for a Function Driver, the interrupt is masked and must explicitly be acknowledged by the Function Driver before another Device Interrupt can be received.

A system power event signals to the Function Driver that the system is either about to suspend to a state where the Function Driver will cease to run or that the system is restoring from such a state. The intention is that a Function Driver may prepare the Device and itself for losing communication such that communication can be reestablished when power is restored. On some platforms, suspend and resume may be unsupported or not require any handling at all. However, usually this involves reading hardware registers off the Device and into main memory prior to suspending and loading these values back onto the Device during restore.

Function Driver callback acknowledgement

The Function Driver callbacks must be acknowledged using the following functions:

```
void CsrSdioInsertedAcknowledge(CsrSdioFunction *function, CsrResult res);
void CsrSdioRemovedAcknowledge(CsrSdioFunction *function);
void CsrSdioInterruptAcknowledge(CsrSdioFunction *function);
void CsrSdioSuspendAcknowledge(CsrSdioFunction *function, CsrResult res);
```

```
void CsrSdioResumeAcknowledge(CsrSdioFunction *function, CsrResult res);
```

Insertion callbacks must be acknowledged to inform CSR SDIO whether or not the Function Driver attached to the Device Function. From the insertion callback has been called until the insertion is acknowledged, the Function Driver has exclusive access to the Device Function and may perform read or write operations on it if needed to determine whether to attach to the Device Function or not. If attachment fails for some reason (e.g. a lack of memory or other resources), the insertion must be acknowledged with `CSR_RESULT_FAILURE` as the value for `res`. `CSR_RESULT_FAILURE` must be passed if the Function Driver claims the Device Function and attaches to it. If `CSR_RESULT_FAILURE` is passed, the Function Driver loses its access rights to the Device Function from this point on. Otherwise, the Function Driver may use the handle until it is lost through its remove callback as described below.

Removal callbacks indicate that the Device has been removed or otherwise become unavailable. This event must be acknowledged by calling `CsrSdioRemovedAcknowledge` to inform CSR SDIO that the Function Driver no longer uses the Function Handle. Once the removal has been acknowledged, the Function Driver has lost access rights to the Device Function.

Suspend callbacks indicate that the system wants to enter a power state where the Function Driver will not be running. This event must be acknowledged, and the Function Driver can inform CSR SDIO whether it is able to suspend or not by passing for `res` either `CSR_RESULT_SUCCESS` or `CSR_RESULT_FAILURE` if able or unable, respectively.

Resume callbacks indicate that the system is returning to a power state where the Function Driver will be running. This event must be acknowledged, and the Function Driver can inform CSR SDIO whether it is able to resume operation or not by passing for `res` either `CSR_RESULT_SUCCESS` or `CSR_RESULT_FAILURE` if able or unable, respectively.

Interrupt callbacks are explained in greater detail later on but must be acknowledged using `CsrSdioInterruptAcknowledge`.

Device Specification

Along with the callback functions that provide the entry points to a Function Driver, there is also an array of descriptors that inform CSR SDIO of which Devices are supported. The length of this array is the final member of the CsrSdioFunctionDriver struct, and the array is of type CsrSdioFunctionId:

```
typedef struct
{
    CsrUint16 manfId;
    CsrUint16 cardId;
    CsrUint8 sdioFunction;
    CsrUint8 sdioInterface;
} CsrSdioFunctionId;
#define CSR_SDIO_ANY_MANF_ID        0xFFFF
#define CSR_SDIO_ANY_CARD_ID        0xFFFF
#define CSR_SDIO_ANY_SDIO_FUNCTION  0xFF
#define CSR_SDIO_ANY_SDIO_INTERFACE 0xFF
```

CsrSdioFunctionId contains values for vendor ID, Device ID, Device Function number, and Device Function interface class ID. These can be set to a particular value or the corresponding CSR_SDIO_ANY_* wildcard value if a Function Driver has no particular preference on a given property.

This section provides several examples of CsrSdioFunctionId lists.

A) Support CSR (0x032A) UniFi-1 (0x0001) Device Function 1 with Standard Interface Code 0:

```
CsrSdioFunctionId descriptors[] =
{
    {
        0x032A,
        0x0001,
        1,
        0
    },
};
```

B) Support all CSR Bluetooth Type-A (2) Devices:

```
CsrSdioFunctionId csrTypeA[] = {
   { 0x032a, CSR_SDIO_ANY_CARD_ID, CSR_SDIO_ANY_SDIO_FUNCTION, 2 },
};
```

C) Support all Device Functions of CSR UniFi-1 with any SDIO Standard Interface Code:

```
CsrSdioFunctionId descriptors[] =
{
    {
        0x032A,
        0x0001,
        CSR_SDIO_ANY_SDIO_FUNCTION,
        CSR_SDIO_ANY_SDIO_INTERFACE
    },
};
```

D) Support Device Function 1 and 3 of an arbitrary given vendor/Device ID:

```
CsrSdioFunctionId descriptors[] =
{
    {
        0x1234,
        0xABCD,
        1,
        CSR_SDIO_ANY_SDIO_INTERFACE
    },
    {
        0x1234,
        0xABCD,
        3,
        CSR_SDIO_ANY_SDIO_INTERFACE
    },
};
```

E) Support Device Function 1 of two different vendor/Device ID:

```
CsrSdioFunctionId descriptors[] =
{
    {
        0x1234,
        0xABCD,
        1,
        CSR_SDIO_ANY_SDIO_INTERFACE
    },
    {
        0x5678,
        0xEEEE,
        1,
        CSR_SDIO_ANY_SDIO_INTERFACE
    },
};
```

Note that in example C to E several different Device Functions are supported, which means that the single Function Driver must be able to handle these simultaneously using only a single set of callback functions that will be called independently for each Device Function attached. Depending on the number of slots supported by the implementation, it is possible that the Function Driver must handle several identical Device Functions - even for example A, if identical Devices are inserted in several slots. The Function Driver can distinguish two identical attached Device Functions by looking at their function identifiers that are passed as a parameter to all Function Driver callback functions.

Function identifiers

The callbacks are passed a pointer to a CsrSdioFunction struct that uniquely identifies the Device Function on which the event happened:

```
typedef struct
{
    CsrSdioFunctionId sdioId;
    CsrUint16 blockSize;
    CsrUint32 features;
    void *driverData;   /* For use by the Function Driver */
    void *priv;         /* For use by the SDIO Driver */
} CsrSdioFunction;
```

The field features provides a bitmap that describes the features supported by the SDIO driver. In particular, CSR_SDIO_FEATURE_BYTE_MODE indicates support for CMD53 in byte mode. The field blockSize is the block size selected by the SDIO driver and written to the block size register of the function, or zero. The field priv is meant to be used only by the SDIO driver itself and may not be changed by the Function Driver.

The driverData field of this struct can be used by the Function Driver to store internal state data about the Device. The sdioId field provides a Device Specification that a Function Driver can use for instance to implement Device-specific quirks if necessary.

2.4 Callbacks and contexts

Three categories are defined for the contexts in which callback functions supported by the SDIO API are called. It is important to note the difference since these contexts define what the user of the SDIO API may and may not do. The following contexts are defined:

1. Thread context: all operating system features and services are available including e.g. the ability to suspend on blocking operations and allocate memory

2. Low level context: very few, if any, operating system features are available

3. High level context: can set scheduler background interrupt

Low level context may, depending on implementation, mean low level interrupt context. The following criteria shall be met for the low level callback execution context:

1. It is possible to start another SDIO data transfer operation by calling one of the asynchronous SDIO data transfer functions directly from the callback.

2. It is possible to acknowledge Device Interrupts by calling the appropriate function directly from the callback.

Specifically, item 1 explicitly allows calling the CsrSdio*Async() functions described later on. Item 2 explicitly allows calling the CsrSdioInterruptAcknowledge() function described in section 0. Alternatively to item 1 and 2, it is possible to schedule a CsrSdioDsrCallback, which provides a higher level interrupt service context, by returning a CsrSdioDsrCallback from this callback. Note: execution may or may not be interruptible in this execution context, and as a consequence suspending on any thread locking mechanisms may deadlock.

In high level context it is as a minimum possible to set a background interrupt in the scheduler.

2.4.1 CsrSdioInsertedCallback

CsrSdioInsertedCallback is the type of function called when a Device Function becomes available to a registered Function Driver that supports the function:

```
typedef void (*CsrSdioInsertedCallback)(CsrSdioFunction *function);
```

This type of function is called in thread context.

2.4.2 CsrSdioRemovedCallback

CsrSdioRemovedCallback is the type of function called when a Device Function is no longer available to a Function Driver, either because the Device has been removed, or the Function Driver has been unregistered:

```
typedef void (*CsrSdioRemovedCallback)(CsrSdioFunction *function);
```

This type of function is called in thread context.

2.4.3 CsrSdioInterruptCallback

CsrSdioInterruptCallback is the type of function called when a low-level interrupt is received. This function may return a pointer to a function of type CsrSdioInterruptDsrCallback that is scheduled at a higher, more permissive execution context.

```
typedef CsrSdioInterruptDsrCallback(*CsrSdioInterruptCallback)
(CsrSdioFunction *function);
```

This type of function is called in low level context.

2.4.4 CsrSdioInterruptDsrCallback

CsrSdioInterruptDsrCallback is the type of function that may be scheduled from a low-level interrupt callback function:

```
typedef void (*CsrSdioInterruptDsrCallback)(CsrSdioFunction *function);
```

This type of function is called in high level context.

2.4.5 CsrSdioSuspendCallback

CsrSdioSuspendCallback is the type of function called when CSR SDIO needs to inform a Function Driver that the system is about to be put in a state where the Function Driver will not run:

```
typedef void (*CsrSdioSuspendCallback)(CsrSdioFunction *function);
```

This type of function is called in thread context.

2.4.6 CsrSdioResumeCallback

CsrSdioSuspendCallback is the type of function called when CSR SDIO needs to inform a Function Driver that the system is back in a state where the Function Driver will run:

```
typedef void (*CsrSdioResumeCallback)(CsrSdioFunction *function);
```

This type of function is called in thread context.

2.4.7 CsrSdioCallback

This callback type is used when an asynchronous data transfer operation completes.

```
typedef CsrSdioDsrCallback (*CsrSdioCallback)(CsrSdioFunction *function, CsrResult
result, void *userData);
```

This type of function is called in low level context.

2.4.8 CsrSdioDsrCallback

This callback type is used when a valid CsrSdioDsrCallback is returned from a CsrSdioCallback callback:

```
typedef void (*CsrSdioDsrCallback)(CsrSdioFunction *function, CsrResult result);
```

This type of function is called in high level context.

2.5 SDIO API description

This section describes all the SDIO API functions in details. Each function description contains:

- Prototype of the function
- Number and type of each parameter
- Possible return values

The functions may return an error code as the result if the requested operation did not complete successfully. The result may either be:

- The CSR Synergy success result code, `CSR_RESULT_SUCCESS`
- The CSR Synergy general failure result code, `CSR_RESULT_FAILURE`
- Any of the following CSR SDIO result code:
  - `CSR_SDIO_RESULT_INVALID_VALUE`
  - `CSR_SDIO_RESULT_NO_DEVICE`
  - `CSR_SDIO_RESULT_CRC_ERROR`
  - `CSR_SDIO_RESULT_TIMEOUT`
  - `CSR_SDIO_RESULT_NOT_RESET`

Note: All functions may return either of `CSR_RESULT_SUCCESS` and `CSR_RESULT_FAILURE`, so these return values are implicit and therefore not mentioned in the following descriptions except when there are special circumstances that require returning them.

2.5.1 CsrSdioFunctionDriverRegister

Prototype

```
#include "csr_sdio.h"

CsrResult CsrSdioFunctionDriverRegister(CsrSdioFunctionDriver *driver);
```

Description

Registering a Function Driver will make it eligible for receiving the insert callback when a supported Device Function becomes available. CSR SDIO will never try to deallocate the memory area associated with the CsrSdioFunctionDriver instance. It is the responsibility of the Function Driver to allocate and deallocate this memory. The instance must be retained for as long as the Function Driver remains registered as CSR SDIO will continue to reference the given pointer.

Important: This function must never be called from a callback, or in any other context that is not interruptible, and it must be allowed to suspend on thread locking mechanisms.

Parameters

Type	Argument	Description
`CsrSdioFunctionDriver*`	`driver`	Pointer to Function Driver object, which describes the callbacks and the function descriptors.

Table 1: Arguments to CsrSdioFunctionDriverRegister

Returns

Value	Reason

CSR_RESULT_FAILURE	It was not possible to register the Function Driver.

2.5.2 CsrSdioFunctionDriverUnregister

Prototype

```
#include "csr_sdio.h"

void CsrSdioFunctionDriverUnregister (CsrSdioFunctionDriver *driver);
```

Description

Before unregistering a Function Driver, all data transfer activity must have ceased, and interrupts from the attached Device Functions must be disabled. Calling CsrSdioFunctionDriverUnregister causes the remove callback to trigger for any attached functions.

Important: This function must never be called from a callback, or in any other context that is not interruptible, and it must be allowed to suspend on thread locking mechanisms.

Parameters

Type	Argument	Description
CsrSdioFunctionDriver*	driver	Pointer to the same Function Driver object that was passed to CsrSdioFunctionDriverRegister() and which should now be unregistered.

Table 2: Arguments to CsrSdioFunctionDriverUnregister

Returns

This function does not return a value.

2.5.3 CsrSdioFunctionIdle

Prototype

```
#include "csr_sdio.h"

void CsrSdioFunctionIdle(CsrSdioFunction *function);
```

Description

This can be used by a Function Driver to inform CSR SDIO that the driver expects the Device Function to be idle. When all Device Functions on a single Device are idle, the SDIO implementation may perform certain operations to lower the power consumption. When a Function Driver has marked itself as being idle and it needs to perform a bus transaction, it must first call `CsrSdioFunctionActive()` before it can issue commands.

Important: This function must never be called from a callback, or in any other context that is not interruptible, and it must be allowed to suspend on thread locking mechanisms.

Parameters

Type	Argument	Description
CsrSdioFunction*	func	A Function Handle that uniquely represents a given Device Function on the Device.

Table 3: Arguments to CsrSdioFunctionIdle

Returns

This function does not return a value.

2.5.4 CsrSdioFunctionActive

Prototype

```
#include ”csr_sdio.h”

void CsrSdioFunctionActive(CsrSdioFunction *function);
```

Description

This function is used by Function Drivers to inform CSR SDIO that the function is no longer idle and needs to make a bus transaction. A Function Driver that has marked itself as being idle with `CsrSdioFunctionIdle()` must call this function before performing any other SDIO operation.

Important: This function must never be called from a callback, or in any other context that is not interruptible, and it must be allowed to suspend on thread locking mechanisms.

Parameters

Type	Argument	Description
`CsrSdioFunction*`	`func`	A Function Handle that uniquely represents a given Device Function on the Device.

Table 4: Arguments to CsrSdioFunctionActive

Returns

This function does not return a value.

2.5.5 CsrSdioFunctionEnable

Prototype

```
#include ”csr_sdio.h”

CsrResult CsrSdioFunctionEnable(CsrSdioFunction *function);
```

Description

After a Device Function has been attached to a Function Driver it is initially disabled. Before transferring data the Device Function must be enabled.

The CsrSdioFunctionEnable function will enable the specified Device Function and poll the ready register immediately after. If the function does not become ready within a reasonable amount of time or an error occurs, this is indicated in the function return value.

Important: This function must never be called in any context that is not interruptible, and it must be allowed to suspend on thread locking mechanisms.

Parameters

Type	Argument	Description
`CsrSdioFunction*`	`func`	A Function Handle that uniquely represents a given Device Function on the Device.

Table 5: Arguments to CsrSdioFunctionEnable

Returns

Value	Reason
CSR_SDIO_RESULT_NO_DEVICE	The Device does not exist anymore.
CSR_SDIO_RESULT_CRC_ERROR	A CRC error occurred. The state of the related bit in the I/O Enable register is undefined.
CSR_SDIO_RESULT_TIMEOUT	No response from the Device, or the related bit in the I/O ready register was not set/cleared within the timeout period.
CSR_SDIO_RESULT_INVALID_VALUE	The specified Device Function cannot be enabled because it does not exist or it is not possible to individually enable/disable Device Functions.
CSR_SDIO_RESULT_INVALID_VALUE	The SDIO R5 response is available, and either of the FUNCTION_NUMBER or OUT_OF_RANGE bits are set.
CSR_RESULT_FAILURE	The ERROR bit is set (but none of FUNCTION_NUMBER or OUT_OF_RANGE). The ILLEGAL_COMMAND and COM_CRC_ERROR bits shall be ignored.

2.5.6 CsrSdioFunctionDisable

Prototype

```
#include "csr_sdio.h"

CsrResult CsrSdioFunctionDisable(CsrSdioFunction *function);
```

Description

To disable a previously enabled function, CsrSdioFunctionDisable() must be called.

The CsrSdioFunctionDisable function will disable the specified Device Function and poll the ready register immediately after. If the function stays ready for an implementation defined amount of time, this is indicated in the function return value.

Important: This function must never be called from a callback or in any other context that is not interruptible, and it must be allowed to suspend on thread locking mechanisms.

Parameters

Type	Argument	Description
CsrSdioFunction*	func	A Function Handle that uniquely represents a given Device Function on the Device.

Table 6: Arguments to CsrSdioFunctionDisable

Returns

Value	Reason
CSR_SDIO_RESULT_NO_DEVICE	The Device does not exist anymore.
CSR_SDIO_RESULT_CRC_ERROR	A CRC error occurred. The state of the related bit in the I/O Enable register is undefined.
CSR_SDIO_RESULT_TIMEOUT	No response from the Device, or the related bit in the I/O ready register was not set/cleared within the timeout period.
CSR_SDIO_RESULT_INVALID_VALUE	The specified Device Function cannot be disabled because it does not exist or it is not possible to individually enable/disable Device Functions.

Value	Reason
CSR_SDIO_RESULT_INVALID_VALUE	The SDIO R5 response is available, and either of the FUNCTION_NUMBER or OUT_OF_RANGE bits are set.
CSR_SDIO_RESULT_INVALID_VALUE	The CSPI response is available, and any of the FUNCTION_DISABLED or CLOCK_DISABLED bits are set, CSR_SDIO_RESULT_INVALID_VALUE will be returned.
CSR_RESULT_FAILURE	The ERROR bit is set (but none of FUNCTION_NUMBER or OUT_OF_RANGE). The ILLEGAL_COMMAND and COM_CRC_ERROR bits shall be ignored.

2.5.7 CsrSdioInterruptEnable

Prototype

```
#include "csr_sdio.h"

CsrResult CsrSdioInterruptEnable(CsrSdioFunction *function);
```

Description

Initially all Device Interrupts are disabled on the device. By calling the `CsrSdioInterruptEnable()` function Device Interrupts are enabled on the device for this specified Device Function, and the reception of Device Interrupts is unmasked for the device associated with the given function. If a Device Interrupt occurs, a CSR SDIO implementation may inform all Device Functions with interrupts enabled of the interrupt via their Function Driver interrupt callback function and let the individual Function Drivers determine if they have a pending interrupt, or the CSR SDIO implementation may choose to check the interrupt pending register and only inform the relevant functions. A generic Function Driver must be able to handle spurious interrupts.

Important: This function must never be called from a context that is not interruptible, and it must be allowed to suspend on thread locking mechanisms.

Parameters

Type	Argument	Description
CsrSdioFunction*	func	A Function Handle that uniquely represents a given Device Function on the Device.

Table 7: Arguments to CsrSdioInterruptEnable

Returns

Value	Reason
CSR_SDIO_RESULT_NO_DEVICE	The Device does not exist anymore.
CSR_SDIO_RESULT_CRC_ERROR	A CRC error occurred. The state of the related bit in the I/O Enable register is undefined.
CSR_SDIO_RESULT_TIMEOUT	No response from the Device, or the related bit in the I/O ready register was not set/cleared within the timeout period.
CSR_SDIO_RESULT_INVALID_VALUE	The SDIO R5 response is available, and either of the FUNCTION_NUMBER or OUT_OF_RANGE bits are set.
CSR_SDIO_RESULT_INVALID_VALUE	The CSPI response is available, and any of the FUNCTION_DISABLED or CLOCK_DISABLED bits are set, CSR_SDIO_RESULT_INVALID_VALUE will be returned.
CSR_RESULT_FAILURE	The ERROR bit is set (but none of FUNCTION_NUMBER or OUT_OF_RANGE). The ILLEGAL_COMMAND and COM_CRC_ERROR bits shall be ignored.

2.5.8 CsrSdioInterruptDisable

Prototype

```
#include ”csr_sdio.h”

CsrResult CsrSdioInterruptDisable(CsrSdioFunction *function);
```

Description

The function CsrSdioInterruptDisable can be used for disabling interrupts for a specific attached function.

Important: This function must never be called from a callback, or in any other context that is not interruptible, and it must be allowed to suspend on thread locking mechanisms.

Parameters

Type	Argument	Description
CsrSdioFunction*	func	A Function Handle that uniquely represents a given Device Function on the Device.

Table 8: Arguments to CsrSdioInterruptDisable

Returns

Value	Reason
CSR_SDIO_RESULT_NO_DEVICE	The Device does not exist anymore.
CSR_SDIO_RESULT_CRC_ERROR	A CRC error occurred. The state of the related bit in the I/O Enable register is undefined.
CSR_SDIO_RESULT_TIMEOUT	No response from the Device, or the related bit in the I/O ready register was not set/cleared within the timeout period.
CSR_SDIO_RESULT_INVALID_VALUE	The SDIO R5 response is available, and either of the FUNCTION_NUMBER or OUT_OF_RANGE bits are set.
CSR_SDIO_RESULT_INVALID_VALUE	The CSPI response is available, and any of the FUNCTION_DISABLED or CLOCK_DISABLED bits are set, CSR_SDIO_RESULT_INVALID_VALUE will be returned.
CSR_RESULT_FAILURE	The ERROR bit is set (but none of FUNCTION_NUMBER or OUT_OF_RANGE). The ILLEGAL_COMMAND and COM_CRC_ERROR bits shall be ignored.

2.5.9 CsrSdioInterruptAcknowledge

Prototype

```
#include ”csr_sdio.h”

void CsrSdioInterruptAcknowledge(CsrSdioFunction *function);
```

Description

When a Device Interrupt is detected, Device Interrupts are masked and the Function Driver interrupt callback function is called. It is the responsibility of a Function Driver to clear the pending Device Interrupt in a Device Function specific manner. This is usally done by writing to a specific register within the Device Function. When the Device Interrupt has been handled, the Function Driver must call the `CsrSdioInterruptAcknowledge()`

function. Until this function is called, all Device Interrupts from the associated device will remain masked and CSR SDIO will not signal any further Device Interrupts for any Device Functions on this device.

Important: This function must not be called in any context that is not interruptible, and it must be allowed to suspend on thread locking mechanisms. However, as an exception, it can be called safely from callbacks that are triggered in response to a data transfer operation or the interrupt callback itself.

Parameters

Type	Argument	Description
`CsrSdioFunction*`	`func`	A Function Handle that uniquely represents a given Device Function on the Device.

Table 9: Arguments to CsrSdioInterruptAcknowledge

Returns

This function does not return a value.

2.5.10 CsrSdioPowerOff

Prototype

```
#include "csr_sdio.h"

void CsrSdioPowerOff(CsrSdioFunction *function);
```

Description

The CsrSdioPowerOff function can be used by a Function Driver to power off an Device. Once this has been done, no more operations may be issued to the device before it has been powered back on using `CsrSdioPowerOn`.

Note that a Device can only be powered off when all attached Function Drivers have requested power off Function Drivers are isolated and that multi-function Devices with e.g. both a Bluetooth and a wifi function do not need to coordinate when setting Device power state.

Important: This function must never be called from a callback or in any other context that is not interruptible, and it must be allowed to suspend on thread locking mechanisms.

Parameters

Type	Argument	Description
`CsrSdioFunction*`	`func`	A Function Handle that uniquely represents a given Device Function on the Device.

Table 10: Arguments to CsrSdioPowerOff

Returns

This function does not return a value.

2.5.11 CsrSdioPowerOn

Prototype

```
#include "csr_sdio.h"
```

```
CsrResult CsrSdioPowerOn(CsrSdioFunction *function);
```

Description

The CsrSdioPowerOn function is used by a Function Driver to power an Device back on when it has previously been powered off using `CsrSdioPowerOff`.

The return value of this function can be used for determining if power has actually been restored or if the Device was never powered down. The latter can happen e.g. on multi-function Devices where only one attached Function Driver has requested power off.

Important: This function must never be called from a callback or in any other context that is not interruptible, and it must be allowed to suspend on thread locking mechanisms.

Parameters

Type	Argument	Description
`CsrSdioFunction*`	`func`	A Function Handle that uniquely represents a given Device Function on the Device.

Table 11: Arguments to CsrSdioPowerOn

Returns

Value	Reason
`CSR_SDIO_RESULT_NO_DEVICE`	The Device does not exist anymore.
`CSR_SDIO_RESULT_CRC_ERROR`	A CRC error occurred during reinitialisation
`CSR_SDIO_RESULT_TIMEOUT`	No response from the Device during reinitialisation.
`CSR_SDIO_RESULT_NOT_RESET`	The reset was not applied because it is not supported. The state of the Device is unchanged.

2.5.12 CsrSdioHardReset

Prototype

```
#include "csr_sdio.h"

CsrResult CsrSdioHardReset(CsrSdioFunction *function);
```

Description

A Function Driver may request the CSR SDIO layer to perform a hardware reset of the Device. This can be done by asserting a dedicated reset line or by power-cycling the Device.

`CsrSdioHardReset()` must not cause any Function Driver's inserted() or removed() callbacks to be called.

Important: This function must never be called from a callback or in any other context that is not interruptible, and it must be allowed to suspend on thread locking mechanisms.

Parameters

Type	Argument	Description
`CsrSdioFunction*`	`func`	A Function Handle that uniquely represents a given Device Function on the Device.

Table 12: Arguments to CsrSdioHardReset

Returns

Value	Reason
CSR_SDIO_RESULT_NO_DEVICE	The Device does not exist anymore.
CSR_SDIO_RESULT_CRC_ERROR	A CRC error occurred during reinitialisation
CSR_SDIO_RESULT_TIMEOUT	No response from the Device during reinitialisation.
CSR_SDIO_RESULT_NOT_RESET	The reset was not applied because it is not supported. The state of the Device is unchanged.

2.5.13 CsrSdioBlockSizeSet

Prototype

```
#include "csr_sdio.h"

CsrResult CsrSdioBlockSizeSet(CsrSdioFunction *function, CsrUint16 blockSize);
```

Description

The CsrSdioBlockSizeSet function sets the operational block size for data transfers, and the currently configured value is available from the `blockSize` member in the Function Handle. CSR SDIO does not necessarily set `blockSize` by default, and in this case it shall be set to 0. Function Drivers that wish to configure a particular block size may call this function, but Function Drivers are not required to do so if they have no preference for block size value.

When this function returns, the value of the block size register on the Device is available in the `blockSize` member of the Function Handle. If the implementation is running on CSPI which has no concept of a transfer block size, `blockSize` is set to zero, and calls to this function will have no effect and `blockSize` shall remain 0.

Important: This function must never be called from a callback, or in any other context that is not interruptible, and it must be allowed to suspend on thread locking mechanisms.

Note: Setting the block size requires two individual operations. The implementation shall ignore the OUT_OF_RANGE bit of the SDIO R5 response for the first operation, as the partially configured block size may be out of range, even if the final block size (after the second operation) is in the valid range.

Parameters

Type	Argument	Description
CsrSdioFunction*	func	A Function Handle that uniquely represents a given Function on the Device.
CsrUint16	blockSize	The desired block size. The actual frequency set by the SDIO implementation is also returned in this parameter.

Table 13: Arguments to CsrSdioBlockSizeSet

Returns

Value	Reason
CSR_SDIO_RESULT_INVALID_VALUE	The specified block size is invalid.

CSR_SDIO_RESULT_INVALID_VALUE	The SDIO R5 response is available, and either of the FUNCTION_NUMBER or OUT_OF_RANGE bits are set.
CSR_SDIO_RESULT_INVALID_VALUE	The CSPI response is available, and any of the FUNCTION_DISABLED or CLOCK_DISABLED bits are set, CSR_SDIO_RESULT_INVALID_VALUE will be returned.
CSR_SDIO_RESULT_NO_DEVICE	The Device does not exist anymore.
CSR_SDIO_RESULT_CRC_ERROR	CRC error occurred. The configured block size is undefined.
CSR_SDIO_RESULT_TIMEOUT	No response from the Device.

2.5.14 CsrSdioMaxBusClockFrequencySet

Prototype

```
#include ”csr_sdio.h”

CsrResult CsrSdioMaxBusClockFrequencySet(CsrSdioFunction *function, CsrUint32
maxFrequency);
```

Description

Set the maximum clock frequency to use for the device associated with the specified function. The actual configured clock frequency for the device shall be the minimum of:

1) Maximum clock frequency supported by the device

2) Maximum clock frequency supported by the host controller

3) Maximum clock frequency specified for any function on the same device

If the clock frequency exceeds 25MHz, it is the responsibility of the SDIO driver to enable high speed mode on the device, using the standard defined procedure, before increasing the frequency beyond the limit.

Note that the clock frequency configured affects all functions on the same device.

Important: This function must never be called from a callback, or in any other context that is not interruptible, and it must be allowed to suspend on thread locking mechanisms.

Parameters

Type	Argument	Description
CsrSdioFunction*	func	A Function Handle that uniquely represents a given Function on the Device.
CsrUint32	maxFrequency	The desired bus frequency.

Table 14: Arguments to CsrSdioMaxBusClockFrequencySet

Returns

Value	Reason
CSR_SDIO_RESULT_INVALID_VALUE	The specified clock frequency is invalid.
CSR_SDIO_RESULT_INVALID_VALUE	The SDIO R5 response is available, and either of the FUNCTION_NUMBER or OUT_OF_RANGE bits are set.
CSR_SDIO_RESULT_INVALID_VALUE	The CSPI response is available, and any of the FUNCTION_DISABLED or CLOCK_DISABLED bits are set, CSR_SDIO_RESULT_INVALID_VALUE will be returned.

CSR_SDIO_RESULT_NO_DEVICE	The Device does not exist anymore.
CSR_SDIO_RESULT_CRC_ERROR	CRC error occurred. The configured block size is undefined.
CSR_SDIO_RESULT_TIMEOUT	No response from the Device.

2.5.15 CsrSdioRead8

Prototype

```
#include "csr_sdio.h"

CsrResult CsrSdioRead8(CsrSdioFunction *function, CsrUint32 address, CsrUint8
*data);
```

Description

This function provides a blocking byte oriented data transfer using SDIO CMD52 or register operation for CSPI.

Important: This function must not be called in any context that is not interruptible, and it must be allowed to suspend on thread locking mechanisms.

Parameters

Type	Argument	Description
CsrSdioFunction	func	A Function Handle that uniquely represents a given Device Function on the Device.
CsrUint32	address	Address where the data is read from.
CsrUint8*	data	Pointer to an address where to store the data from the Device.

Table 15: Arguments to CsrSdioRead8

Returns

Value	Reason
CSR_SDIO_RESULT_INVALID_VALUE	One or more arguments were invalid.
CSR_SDIO_RESULT_INVALID_VALUE	The SDIO R5 response is available, and either of the FUNCTION_NUMBER or OUT_OF_RANGE bits are set.
CSR_SDIO_RESULT_INVALID_VALUE	The CSPI response is available, and any of the FUNCTION_DISABLED or CLOCK_DISABLED bits are set, CSR_SDIO_RESULT_INVALID_VALUE will be returned.
CSR_SDIO_RESULT_NO_DEVICE	The Device does not exist anymore.
CSR_SDIO_RESULT_CRC_ERROR	A CRC error occurred. No data read/written.
CSR_SDIO_RESULT_TIMEOUT	No response from the Device.
CSR_RESULT_FAILURE	The ERROR bit is set (but none of FUNCTION_NUMBER or OUT_OF_RANGE), shall be returned. The ILLEGAL_COMMAND and COM_CRC_ERROR bits shall be ignored.

2.5.16 CsrSdioRead8Async

Prototype

```
#include "csr_sdio.h"
```

```
void CsrSdioRead8Async(CsrSdioFunction *function, CsrUint32 address, CsrUint8
*data, CsrSdioCallback cb);
```

Description

This function provides a non-blocking byte oriented data transfer using SDIO CMD52 or register operation for CSPI. When the operation completes, the provided callback is called and passed a result code that indicates the status of the operation. Note that the function does not return a value. All operation status is provided solely to the callback.

If a Function Driver is already performing a data transfer operation, the command may be queued or an error code will be returned if the implementation does not support queuing of commands or if the queue is full.

Parameters

Type	Argument	Description
CsrSdioFunction*	func	A Function Handle that uniquely represents a given Device Function on the Device.
CsrUint32	address	Address where the data is read from.
CsrUint8*	data	Pointer to an address where to store the data from the Device.
CsrSdioCallback	cb	A function pointer to a callback that is called upon operation completion.

Table 16: Arguments to CsrSdioRead8Async

Returns

This function does not return a value but instead passes error codes to the supplied callback:

Value	Reason
CSR_SDIO_RESULT_INVALID_VALUE	One or more arguments were invalid.
CSR_SDIO_RESULT_INVALID_VALUE	The SDIO R5 response is available, and either of the FUNCTION_NUMBER or OUT_OF_RANGE bits are set.
CSR_SDIO_RESULT_INVALID_VALUE	The CSPI response is available, and any of the FUNCTION_DISABLED or CLOCK_DISABLED bits are set, CSR_SDIO_RESULT_INVALID_VALUE will be returned.
CSR_SDIO_RESULT_NO_DEVICE	The Device does not exist anymore.
CSR_SDIO_RESULT_CRC_ERROR	A CRC error occurred. No data read/written.
CSR_SDIO_RESULT_TIMEOUT	No response from the Device.
CSR_RESULT_FAILURE	The ERROR bit is set (but none of FUNCTION_NUMBER or OUT_OF_RANGE), shall be returned. The ILLEGAL_COMMAND and COM_CRC_ERROR bits shall be ignored.

2.5.17 CsrSdioWrite8

Prototype

```
#include "csr_sdio.h"

CsrResult CsrSdioWrite8(CsrSdioFunction *function, CsrUint32 address, CsrUint8
data);
```

Description

This function provides a blocking byte oriented data transfer using SDIO CMD52 or register operation for CSPI.

Important: This function must not be called in any context that is not interruptible, and it must be allowed to suspend on thread locking mechanisms.

Parameters

Type	Argument	Description
CsrSdioFunction*	func	A Function Handle that uniquely represents a given Device Function on the Device.
CsrUint32	address	Address on the Device where the data is written to.
CsrUint8	data	An 8 bit value to write to the Device.

Table 17: Arguments to CsrSdioWrite8

Returns

Value	Reason
CSR_SDIO_RESULT_INVALID_VALUE	One or more arguments were invalid.
CSR_SDIO_RESULT_INVALID_VALUE	The SDIO R5 response is available, and either of the FUNCTION_NUMBER or OUT_OF_RANGE bits are set.
CSR_SDIO_RESULT_INVALID_VALUE	The CSPI response is available, and any of the FUNCTION_DISABLED or CLOCK_DISABLED bits are set, CSR_SDIO_RESULT_INVALID_VALUE will be returned.
CSR_SDIO_RESULT_NO_DEVICE	The Device does not exist anymore.
CSR_SDIO_RESULT_CRC_ERROR	A CRC error occurred. No data read/written.
CSR_SDIO_RESULT_TIMEOUT	No response from the Device.
CSR_RESULT_FAILURE	The ERROR bit is set (but none of FUNCTION_NUMBER or OUT_OF_RANGE), shall be returned. The ILLEGAL_COMMAND and COM_CRC_ERROR bits shall be ignored.

2.5.18 CsrSdioWrite8Async

Prototype

```
#include ”csr_sdio.h”

CsrResult CsrSdioWrite8Async(CsrSdioFunction *function, CsrUint32 address, CsrUint8
data, CsrSdioCallback cb);
```

Description

This function provides a non-blocking byte oriented data transfer using SDIO CMD52 or register operation for CSPI. When the operation completes, the provided callback is called and passed a result code that indicates the status of the operation. Note that the function does not return a value. All operation status is provided solely to the callback.

If a Function Driver is already performing a data transfer operation, the command may be queued or an error code will be returned if the implementation does not support queuing of commands or if the queue is full.

Parameters

Type	Argument	Description
CsrSdioFunction*	func	A Function Handle that uniquely represents a given Device Function on the Device.
CsrUint32	address	Address on the Device where the data is written to.
CsrUint8	data	An 8 bit value to write to the Device.
CsrSdioCallback	cb	A function pointer to a callback that is called upon operation completion.

Table 18: Arguments to CsrSdioWrite8Async

Returns

This function does not return a value but instead passes error codes to the supplied callback:

Value	Reason
CSR_SDIO_RESULT_INVALID_VALUE	One or more arguments were invalid.
CSR_SDIO_RESULT_INVALID_VALUE	The SDIO R5 response is available, and either of the FUNCTION_NUMBER or OUT_OF_RANGE bits are set.
CSR_SDIO_RESULT_INVALID_VALUE	The CSPI response is available, and any of the FUNCTION_DISABLED or CLOCK_DISABLED bits are set, CSR_SDIO_RESULT_INVALID_VALUE will be returned.
CSR_SDIO_RESULT_NO_DEVICE	The Device does not exist anymore.
CSR_SDIO_RESULT_CRC_ERROR	A CRC error occurred. No data read/written.
CSR_SDIO_RESULT_TIMEOUT	No response from the Device.
CSR_RESULT_FAILURE	The ERROR bit is set (but none of FUNCTION_NUMBER or OUT_OF_RANGE), shall be returned. The ILLEGAL_COMMAND and COM_CRC_ERROR bits shall be ignored.

2.5.19 CsrSdioRead16

Prototype

```
#include ”csr_sdio.h”

CsrResult CsrSdioRead16(CsrSdioFunction *function, CsrUint32 address, CsrUint16
*data);
```

Description

This function provides a blocking 16-bit data transfer. If the underlying physical transport does not support 16 bit register accesses (i.e., SDIO but not CSPI) then the implementation of this function shall perform two 8 bit operations under which the lower (even address) shall be read first.

Important: This function must not be called in any context that is not interruptible, and it must be allowed to suspend on thread locking mechanisms.

Parameters

Type	Argument	Description

CsrSdioFunction*	func	A Function Handle that uniquely represents a given Device Function on the Device.
CsrUint32	address	Address where the data is read from
CsrUint16*	data	Pointer to an address where to store the data from the Device.

Table 19: Arguments to CsrSdioRead16

Returns

Value	Reason
CSR_SDIO_RESULT_INVALID_VALUE	One or more arguments were invalid.
CSR_SDIO_RESULT_INVALID_VALUE	The SDIO R5 response is available, and either of the FUNCTION_NUMBER or OUT_OF_RANGE bits are set.
CSR_SDIO_RESULT_INVALID_VALUE	The CSPI response is available, and any of the FUNCTION_DISABLED or CLOCK_DISABLED bits are set, CSR_SDIO_RESULT_INVALID_VALUE will be returned.
CSR_SDIO_RESULT_NO_DEVICE	The Device does not exist anymore.
CSR_SDIO_RESULT_CRC_ERROR	A CRC error occurred. No data read/written.
CSR_SDIO_RESULT_TIMEOUT	No response from the Device.
CSR_RESULT_FAILURE	The ERROR bit is set (but none of FUNCTION_NUMBER or OUT_OF_RANGE), shall be returned. The ILLEGAL_COMMAND and COM_CRC_ERROR bits shall be ignored.

2.5.20 CsrSdioRead16Async

Prototype

```
#include ”csr_sdio.h”

void CsrSdioRead16Async(CsrSdioFunction *function, CsrUint32 address, CsrUint16
*data, CsrSdioCallback cb);
```

Description

This function provides a non-blocking 16-bit data transfer. If the underlying physical transport does not support 16 bit register accesses (i.e., SDIO but not CSPI) then the implementation of this function shall perform two 8 bit operations under which the lower (even address) shall be read first. When the operation completes, the provided callback is called and passed a result code that indicates the status of the operation. Note that the function does not return a value. All operation status is provided solely to the callback.

If a Function Driver is already performing a data transfer operation, the command may be queued or an error code will be returned if the implementation does not support queuing of commands or if the queue is full.

Parameters

Type	Argument	Description
CsrSdioFunction*	func	A Function Handle that uniquely represents a given Device Function on the Device.
CsrUint32	address	Address where the data is read from.

CsrUint16*	data	Pointer to an address where to store the data from the Device.
CsrSdioCallback	cb	A function pointer to a callback that is called upon operation completion.

Table 20: Arguments to CsrSdioRead16Async

Returns

This function does not return a value but instead passes error codes to the supplied callback:

Value	Reason
CSR_SDIO_RESULT_INVALID_VALUE	One or more arguments were invalid.
CSR_SDIO_RESULT_INVALID_VALUE	The SDIO R5 response is available, and either of the FUNCTION_NUMBER or OUT_OF_RANGE bits are set.
CSR_SDIO_RESULT_INVALID_VALUE	The CSPI response is available, and any of the FUNCTION_DISABLED or CLOCK_DISABLED bits are set, CSR_SDIO_RESULT_INVALID_VALUE will be returned.
CSR_SDIO_RESULT_NO_DEVICE	The Device does not exist anymore.
CSR_SDIO_RESULT_CRC_ERROR	A CRC error occurred. No data read/written.
CSR_SDIO_RESULT_TIMEOUT	No response from the Device.
CSR_RESULT_FAILURE	The ERROR bit is set (but none of FUNCTION_NUMBER or OUT_OF_RANGE), shall be returned. The ILLEGAL_COMMAND and COM_CRC_ERROR bits shall be ignored.

2.5.21 CsrSdioWrite16

Prototype

```
#include ”csr_sdio.h”

CsrResult CsrSdioWrite16(CsrSdioFunction *function, CsrUint32 address, CsrUint16
data);
```

Description

This function provides a blocking 16-bit data transfer. If the underlying physical transport does not support 16 bit register accesses (i.e., SDIO but not CSPI) then the implementation of this function shall perform two 8 bit operations under which the upper (odd address) byte shall be written first.

Important: This function must not be called in any context that is not interruptible, and it must be allowed to suspend on thread locking mechanisms.

Parameters

Type	Argument	Description
CsrSdioFunction*	func	A Function Handle that uniquely represents a given Device Function on the Device.
CsrUint32	address	Address where the data is read from
CsrUint16	data	A 16 bit value to write to the Device.

Table 21: Arguments to CsrSdioWrite16

Returns

Value	Reason
CSR_SDIO_RESULT_INVALID_VALUE	One or more arguments were invalid.
CSR_SDIO_RESULT_INVALID_VALUE	The SDIO R5 response is available, and either of the FUNCTION_NUMBER or OUT_OF_RANGE bits are set.
CSR_SDIO_RESULT_INVALID_VALUE	The CSPI response is available, and any of the FUNCTION_DISABLED or CLOCK_DISABLED bits are set, CSR_SDIO_RESULT_INVALID_VALUE will be returned.
CSR_SDIO_RESULT_NO_DEVICE	The Device does not exist anymore.
CSR_SDIO_RESULT_CRC_ERROR	A CRC error occurred. No data read/written.
CSR_SDIO_RESULT_TIMEOUT	No response from the Device.
CSR_RESULT_FAILURE	The ERROR bit is set (but none of FUNCTION_NUMBER or OUT_OF_RANGE), shall be returned. The ILLEGAL_COMMAND and COM_CRC_ERROR bits shall be ignored.

2.5.22 CsrSdioWrite16Async

Prototype

```
#include "csr_sdio.h"

void CsrSdioWrite16Async(CsrSdioFunction *function, CsrUint32 address, CsrUint16
data, CsrSdioCallback cb);
```

Description

This function provides a non-blocking 16-bit data transfer. If the underlying physical transport does not support 16 bit register accesses (i.e., SDIO but not CSPI) then the implementation of this function shall perform two 8 bit operations under which the upper (odd address) byte shall be written first. When the operation completes, the provided callback is called and passed a result code that indicates the status of the operation. Note that the function does not return a value. All operation status is provided solely to the callback.

If a Function Driver is already performing a data transfer operation, the command may be queued or an error code will be returned if the implementation does not support queuing of commands or if the queue is full.

Parameters

Type	Argument	Description
CsrSdioFunction*	func	A Function Handle that uniquely represents a given Device Function on the Device.
CsrUint32	address	Address where the data is read from.
CsrUint16	data	A 16 bit value to write to the Device.
CsrSdioCallback	cb	A function pointer to a callback that is called upon operation completion.

Table 22: Arguments to CsrSdioWrite16Async

Returns

This function does not return a value but instead passes error codes to the supplied callback:

Value	Reason

CSR_SDIO_RESULT_INVALID_VALUE	One or more arguments were invalid.
CSR_SDIO_RESULT_INVALID_VALUE	The SDIO R5 response is available, and either of the FUNCTION_NUMBER or OUT_OF_RANGE bits are set.
CSR_SDIO_RESULT_INVALID_VALUE	The CSPI response is available, and any of the FUNCTION_DISABLED or CLOCK_DISABLED bits are set, CSR_SDIO_RESULT_INVALID_VALUE will be returned.
CSR_SDIO_RESULT_NO_DEVICE	The Device does not exist anymore.
CSR_SDIO_RESULT_CRC_ERROR	A CRC error occurred. No data read/written.
CSR_SDIO_RESULT_TIMEOUT	No response from the Device.
CSR_RESULT_FAILURE	The ERROR bit is set (but none of FUNCTION_NUMBER or OUT_OF_RANGE), shall be returned. The ILLEGAL_COMMAND and COM_CRC_ERROR bits shall be ignored.

2.5.23 CsrSdioF0Read8

Prototype

```
#include ”csr_sdio.h”

CsrResult CsrSdioF0Read8(CsrSdioFunction *function, CsrUint32 address, CsrUint8
*data);
```

Description

This function provides a blocking byte oriented data transfer using SDIO CMD52 or register operation for CSPI. However, it is operates on function 0 instead of whatever function the `func` handle is referring to.

Important: This function must not be called in any context that is not interruptible, and it must be allowed to suspend on thread locking mechanisms.

Parameters

Type	Argument	Description
CsrSdioFunction*	func	A Function Handle that uniquely represents a given Device Function on the Device.
CsrUint32	address	Address where the data is read from
CsrUint8*	data	Pointer to an address where to store the data from the Device.

Table 23: Arguments to CsrSdioF0Read8

Returns

Value	Reason
CSR_SDIO_RESULT_INVALID_VALUE	One or more arguments were invalid.
CSR_SDIO_RESULT_INVALID_VALUE	The SDIO R5 response is available, and either of the FUNCTION_NUMBER or OUT_OF_RANGE bits are set.
CSR_SDIO_RESULT_INVALID_VALUE	The CSPI response is available, and any of the FUNCTION_DISABLED or CLOCK_DISABLED bits are set, CSR_SDIO_RESULT_INVALID_VALUE will be returned.
CSR_SDIO_RESULT_NO_DEVICE	The Device does not exist anymore.
CSR_SDIO_RESULT_CRC_ERROR	A CRC error occurred. No data read/written.

CSR_SDIO_RESULT_TIMEOUT	No response from the Device.
CSR_RESULT_FAILURE	The ERROR bit is set (but none of FUNCTION_NUMBER or OUT_OF_RANGE), shall be returned. The ILLEGAL_COMMAND and COM_CRC_ERROR bits shall be ignored.

2.5.24 CsrSdioF0Read8Async

Prototype

```
#include "csr_sdio.h"

void CsrSdioF0Read8Async(CsrSdioFunction *function, CsrUint32 address, CsrUint8
*data, CsrSdioCallback cb);
```

Description

This function provides a non-blocking byte oriented data transfer using SDIO CMD52 or register operation for CSPI. However, it is operates on function 0 instead of whatever function the `func` handle is referring to. When the operation completes, the provided callback is called and passed a result code that indicates the status of the operation. Note that the function does not return a value. All operation status is provided solely to the callback.

If a Function Driver is already performing a data transfer operation, the command may be queued or an error code will be returned if the implementation does not support queuing of commands or if the queue is full.

Parameters

Type	Argument	Description
CsrSdioFunction*	func	A Function Handle that uniquely represents a given Device Function on the Device.
CsrUint32	address	Address where the data is read from.
CsrUint8*	data	Pointer to an address where to store the data from the Device.
CsrSdioCallback	cb	A function pointer to a callback that is called upon operation completion.

Table 24: Arguments to CsrSdioRead8Async

Returns

This function does not return a value but instead passes error codes to the supplied callback:

Value	Reason
CSR_SDIO_RESULT_INVALID_VALUE	One or more arguments were invalid.
CSR_SDIO_RESULT_INVALID_VALUE	The SDIO R5 response is available, and either of the FUNCTION_NUMBER or OUT_OF_RANGE bits are set.
CSR_SDIO_RESULT_INVALID_VALUE	The CSPI response is available, and any of the FUNCTION_DISABLED or CLOCK_DISABLED bits are set, CSR_SDIO_RESULT_INVALID_VALUE will be returned.
CSR_SDIO_RESULT_NO_DEVICE	The Device does not exist anymore.
CSR_SDIO_RESULT_CRC_ERROR	A CRC error occurred. No data read/written.
CSR_SDIO_RESULT_TIMEOUT	No response from the Device.

CSR_RESULT_FAILURE	The ERROR bit is set (but none of FUNCTION_NUMBER or OUT_OF_RANGE), shall be returned. The ILLEGAL_COMMAND and COM_CRC_ERROR bits shall be ignored.

2.5.25 CsrSdioF0Write8

Prototype

```
#include ”csr_sdio.h”

CsrResult CsrSdioF0Write8(CsrSdioFunction *function, CsrUint32 address, CsrUint8
data);
```

Description

This function provides a blocking byte oriented data transfer using SDIO CMD52 or register operation for CSPI. However, it is operates on function 0 instead of whatever function the `func` handle is referring to.

Important: This function must not be called in any context that is not interruptible, and it must be allowed to suspend on thread locking mechanisms.

Parameters

Type	Argument	Description
CsrSdioFunction*	func	A Function Handle that uniquely represents a given Device Function on the Device.
CsrUint32	address	Address on the Device where the data is written to.
CsrUint8	data	An 8 bit value to write to the Device.

Table 25: Arguments to CsrSdioF0Write8

Returns

Value	Reason
CSR_SDIO_RESULT_INVALID_VALUE	One or more arguments were invalid.
CSR_SDIO_RESULT_INVALID_VALUE	The SDIO R5 response is available, and either of the FUNCTION_NUMBER or OUT_OF_RANGE bits are set.
CSR_SDIO_RESULT_INVALID_VALUE	The CSPI response is available, and any of the FUNCTION_DISABLED or CLOCK_DISABLED bits are set, CSR_SDIO_RESULT_INVALID_VALUE will be returned.
CSR_SDIO_RESULT_NO_DEVICE	The Device does not exist anymore.
CSR_SDIO_RESULT_CRC_ERROR	A CRC error occurred. No data read/written.
CSR_SDIO_RESULT_TIMEOUT	No response from the Device.
CSR_RESULT_FAILURE	The ERROR bit is set (but none of FUNCTION_NUMBER or OUT_OF_RANGE), shall be returned. The ILLEGAL_COMMAND and COM_CRC_ERROR bits shall be ignored.

2.5.26 CsrSdioF0Write8Async

Prototype

```
#include ”csr_sdio.h”

void CsrSdioF0Write8Async(CsrSdioFunction *function, CsrUint32 address, CsrUint8
*data, CsrSdioCallback cb);
```

Description

This function provides a non-blocking byte oriented data transfer using SDIO CMD52 or register operation for CSPI. However, it is operates on function 0 instead of whatever function the `func` handle is referring to. When the operation completes, the provided callback is called and passed a result code that indicates the status of the operation. Note that the function does not return a value. All operation status is provided solely to the callback.

If a Function Driver is already performing a data transfer operation, the command may be queued or an error code will be returned if the implementation does not support queuing of commands or if the queue is full.

Parameters

Type	Argument	Description
CsrSdioFunction*	func	A Function Handle that uniquely represents a given Device Function on the Device.
CsrUint32	address	Address on the Device where the data is written to.
CsrUint8	data	An 8 bit value to write to the Device.
CsrSdioCallback	cb	A function pointer to a callback that is called upon operation completion.

Table 26: Arguments to CsrSdioWrite8Async

Returns

This function does not return a value but instead passes error codes to the supplied callback:

Value	Reason
CSR_SDIO_RESULT_INVALID_VALUE	One or more arguments were invalid.
CSR_SDIO_RESULT_INVALID_VALUE	The SDIO R5 response is available, and either of the FUNCTION_NUMBER or OUT_OF_RANGE bits are set.
CSR_SDIO_RESULT_INVALID_VALUE	The CSPI response is available, and any of the FUNCTION_DISABLED or CLOCK_DISABLED bits are set, CSR_SDIO_RESULT_INVALID_VALUE will be returned.
CSR_SDIO_RESULT_NO_DEVICE	The Device does not exist anymore.
CSR_SDIO_RESULT_CRC_ERROR	A CRC error occurred. No data read/written.
CSR_SDIO_RESULT_TIMEOUT	No response from the Device.
CSR_RESULT_FAILURE	The ERROR bit is set (but none of FUNCTION_NUMBER or OUT_OF_RANGE), shall be returned. The ILLEGAL_COMMAND and COM_CRC_ERROR bits shall be ignored.

2.5.27 CsrSdioRead

Prototype

```
#include ”csr_sdio.h”

CsrResult CsrSdioRead(CsrSdioFunction *function, CsrUint32 address, CsrUint8 *data,
CsrUint32 length);
```

Description

This function provides a blocking data transfer using the SDIO CMD53 or burst operation for CSPI.

The maximum transfer size is limited only by system memory constraints. The SDIO implementation must automatically break a large operation into many smaller operations using either block mode or stream mode, while optimising the time required to transfer the specified data. For best performance, use a transfer size that is a multiple of the block size for the Device Function, as this will minimise the overhead considerably in most cases.

Important: This function must not be called in any context that is not interruptible, and it must be allowed to suspend on thread locking mechanisms.

Note: The SDIO implementation must never try to deallocate the memory area given as the data parameter. It is the responsibility of the Function Driver to allocate and deallocate this memory. The data must be retained until the callback has been triggered as the SDIO implementation will continue to reference the given pointer.

Parameters

Type	Argument	Description
CsrSdioFunction*	func	A Function Handle that uniquely represents a given Device Function on the Device.
CsrUint32	address	Address on the device where the data is read from.
CsrUint8*	data	Pointer to an address where data is written to.
CsrUint32	length	Specifies the length of the pointer referred to by the data parameter.

Table 27: Arguments to CsrSdioRead

Returns

Value	Reason
CSR_SDIO_RESULT_INVALID_VALUE	One or more arguments were invalid.
CSR_SDIO_RESULT_INVALID_VALUE	The SDIO R5 response is available, and either of the FUNCTION_NUMBER or OUT_OF_RANGE bits are set.
CSR_SDIO_RESULT_INVALID_VALUE	The CSPI response is available, and any of the FUNCTION_DISABLED or CLOCK_DISABLED bits are set, CSR_SDIO_RESULT_INVALID_VALUE will be returned.
CSR_SDIO_RESULT_NO_DEVICE	The Device does not exist anymore.
CSR_SDIO_RESULT_CRC_ERROR	A CRC error occurred. No data read/written.
CSR_SDIO_RESULT_TIMEOUT	No response from the Device.
CSR_RESULT_FAILURE	The ERROR bit is set (but none of FUNCTION_NUMBER or OUT_OF_RANGE), shall be returned. The ILLEGAL_COMMAND and COM_CRC_ERROR bits shall be ignored.

2.5.28 CsrSdioReadAsync

Prototype

```
#include "csr_sdio.h"

void CsrSdioReadAsync(CsrSdioFunction *function, CsrUint32 address, CsrUint8 *data,
CsrUint32 length, CsrSdioCallback cb);
```

Description

This function provides a non-blocking data transfer using the SDIO CMD53 or burst operation for CSPI. When the operation completes, the provided callback is called and passed a result code that indicates the status of the operation. Note that the function does not return a value. All operation status is provided solely to the callback.

The maximum transfer size is limited only by system memory constraints. The SDIO implementation must automatically break a large operation into many smaller operations using either block mode or stream mode, while optimising the time required to transfer the specified data. For best performance, use a transfer size that is a multiple of the block size for the Device Function, as this will minimise the overhead considerably in most cases.

If some Function Driver is already performing an data transfer operation, the command may be enqueued or an error code will be returned if the implementation does not support queuing of commands or if the queue is full.

Note: The SDIO implementation must never try to deallocate the memory area given as the data parameter. It is the responsibility of the Function Driver to allocate and deallocate this memory. The data must be retained until the callback has been triggered as the SDIO implementation will continue to reference the given pointer.

Parameters

Type	Argument	Description
CsrSdioFunction*	func	A Function Handle that uniquely represents a given Device Function on the Device.
CsrUint32	address	Address on the device where the data is read from.
CsrUint8*	data	Pointer to an address where data is written to.
CsrUint32	length	Specifies the length of the pointer referred to by the data parameter.
CsrSdioCallback	cb	The callback function to call when the operation completes.

Table 28: Arguments to CsrSdioReadAsync

Returns

This function does not return a value but instead passes error codes to the supplied callback:

Value	Reason
CSR_SDIO_RESULT_INVALID_VALUE	One or more arguments were invalid.
CSR_SDIO_RESULT_INVALID_VALUE	The SDIO R5 response is available, and either of the FUNCTION_NUMBER or OUT_OF_RANGE bits are set.
CSR_SDIO_RESULT_INVALID_VALUE	The CSPI response is available, and any of the FUNCTION_DISABLED or CLOCK_DISABLED bits are set, CSR_SDIO_RESULT_INVALID_VALUE will be returned.
CSR_SDIO_RESULT_NO_DEVICE	The Device does not exist anymore.

CSR_SDIO_RESULT_CRC_ERROR	A CRC error occurred. No data read/written.
CSR_SDIO_RESULT_TIMEOUT	No response from the Device.
CSR_RESULT_FAILURE	The ERROR bit is set (but none of FUNCTION_NUMBER or OUT_OF_RANGE), shall be returned. The ILLEGAL_COMMAND and COM_CRC_ERROR bits shall be ignored.

2.5.29 CsrSdioWrite

Prototype

```
#include "csr_sdio.h"

CsrResult CsrSdioWrite(CsrSdioFunction *function, CsrUint32 address, CsrUint8
*data, CsrUint32 length);
```

Description

This function provides a blocking data transfer using the SDIO CMD53 or burst operation for CSPI.

The maximum transfer size is limited only by system memory constraints. The SDIO implementation must automatically break a large operation into many smaller operations using either block mode or stream mode, while optimising the time required to transfer the specified data. For best performance, use a transfer size that is a multiple of the block size for the Device Function, as this will minimise the overhead considerably in most cases.

Important: This function must not be called in any context that is not interruptible, and it must be allowed to suspend on thread locking mechanisms.

Note: The SDIO implementation must never try to deallocate the memory area given as the data parameter. It is the responsibility of the Function Driver to allocate and deallocate this memory. The data must be retained until the callback has been triggered as the SDIO implementation will continue to reference the given pointer.

Parameters

Type	Argument	Description
CsrSdioFunction*	func	A Function Handle that uniquely represents a given Device Function on the Device.
CsrUint32	address	Address on the device where the data is written to.
CsrUint8*	data	Pointer to an address where data is read from.
CsrUint32	length	Specifies the length of the pointer referred to by the data parameter.

Table 29: Arguments to CsrSdioWrite

Returns

Value	Reason
CSR_SDIO_RESULT_INVALID_VALUE	One or more arguments were invalid.
CSR_SDIO_RESULT_INVALID_VALUE	The SDIO R5 response is available, and either of the FUNCTION_NUMBER or OUT_OF_RANGE bits are set.
CSR_SDIO_RESULT_INVALID_VALUE	The CSPI response is available, and any of the FUNCTION_DISABLED or CLOCK_DISABLED bits are set, CSR_SDIO_RESULT_INVALID_VALUE will be returned.
CSR_SDIO_RESULT_NO_DEVICE	The Device does not exist anymore.
CSR_SDIO_RESULT_CRC_ERROR	A CRC error occurred. No data read/written.
CSR_SDIO_RESULT_TIMEOUT	No response from the Device.
CSR_RESULT_FAILURE	The ERROR bit is set (but none of FUNCTION_NUMBER or OUT_OF_RANGE), shall be returned. The ILLEGAL_COMMAND and COM_CRC_ERROR bits shall be ignored.

2.5.30 CsrSdioWriteAsync

Prototype

```
#include ”csr_sdio.h”

void CsrSdioWriteAsync(CsrSdioFunction *function, CsrUint32 address, CsrUint8
*data, CsrUint32 length, CsrSdioCallback cb);
```

Description

This function provides a non-blocking data transfer using the SDIO CMD53 or burst operation for CSPI. When the operation completes, the provided callback is called and passed a result code that indicates the status of the operation. Note that the function does not return a value. All operation status is provided solely to the callback.

The maximum transfer size is limited only by system memory constraints. The SDIO implementation must automatically break a large operation into many smaller operations using either block mode or stream mode, while optimising the time required to transfer the specified data. For best performance, use a transfer size that is a multiple of the block size for the Device Function, as this will minimise the overhead considerably in most cases.

If some Function Driver is already performing an data transfer operation, the command may be enqueued or an error code will be returned if the implementation does not support queuing of commands or if the queue is full.

Note: The SDIO implementation must never try to deallocate the memory area given as the data parameter. It is the responsibility of the Function Driver to allocate and deallocate this memory. The data must be retained until the callback has been triggered as the SDIO implementation will continue to reference the given pointer.

Parameters

Type	Argument	Description
CsrSdioFunction*	func	A Function Handle that uniquely represents a given Device Function on the Device.
CsrUint32	address	Address on the device where the data is written to.
CsrUint8*	data	Pointer to an address where data is read from.
CsrUint32	length	Specifies the length of the pointer referred to by the data parameter.
CsrSdioCallback	cb	The callback function to call when the operation completes.

Table 30: Arguments to CsrSdioReadAsync

Returns

This function does not return a value but instead passes error codes to the supplied callback:

Value	Reason
CSR_SDIO_RESULT_INVALID_VALUE	One or more arguments were invalid.
CSR_SDIO_RESULT_INVALID_VALUE	The SDIO R5 response is available, and either of the FUNCTION_NUMBER or OUT_OF_RANGE bits are set.
CSR_SDIO_RESULT_INVALID_VALUE	The CSPI response is available, and any of the FUNCTION_DISABLED or CLOCK_DISABLED bits are set, CSR_SDIO_RESULT_INVALID_VALUE will be returned.
CSR_SDIO_RESULT_NO_DEVICE	The Device does not exist anymore.
CSR_SDIO_RESULT_CRC_ERROR	A CRC error occurred. No data read/written.
CSR_SDIO_RESULT_TIMEOUT	No response from the Device.
CSR_RESULT_FAILURE	The ERROR bit is set (but none of FUNCTION_NUMBER or OUT_OF_RANGE), shall be returned. The ILLEGAL_COMMAND and COM_CRC_ERROR bits shall be ignored.

2.6 Semantics for queueing of Operations

An implementation of CSR SDIO must have the following semantics regarding scheduling of operations:

If an operation is available on the queue and the completion callback function for an asynchronous operation schedules another asynchronous operation, the operation on the queue is delayed until no operation is scheduled by completion callback functions.

An example scenario that demonstrates the required semantics:

1. Function Driver A starts an asynchronous operation, A1

2. Function Driver B schedules another operation, B1, which goes on the queue
3. A1 completes and schedules another asynchronous operation, A2
4. A2 is started
5. A2 completes and returns without scheduling another operation
6. B1 is started
7. Function Driver A schedules an asynchronous operation, A3
8. B1 completes and schedules another asynchronous operation, B2
9. B2 completes and returns without scheduling another operation
10. A3 is scheduled
11. A3 completes and returns without scheduling another operation

3 Document References

Ref	Title

Terms and Definitions

[SDIO-SPEC]	SDIO Simplified Specification
CSR	Cambridge Silicon Radio

Document History

Revision	Date	History
1	01 JUL 09	Ready for release 1.1.0
2	30 NOV 09	Ready for release 2.0.0
3	20 APR 10	Ready for release 2.1.0
4	DEC 10	Ready for release 3.0.0
5	Aug 11	Ready for release 3.1.0

TradeMarks, Patents and Licences

Unless otherwise stated, words and logos marked with ™ or ® are trademarks registered or owned by CSR plc or its affiliates. Bluetooth® and the Bluetooth logos are trademarks owned by Bluetooth SIG, Inc. and licensed to CSR. Other products, services and names used in this document may have been trademarked by their respective owners.

The publication of this information does not imply that any licence is granted under any patent or other rights owned by CSR plc.

CSR reserves the right to make technical changes to its products as part of its development programme.

While every care has been taken to ensure the accuracy of the contents of this document, CSR cannot accept responsibility for any errors.

No statements or representations in this document are to be construed as advertising, marketing, or offering for sale in the United States imported covered products subject to the Cease and Desist Order issued by the U.S. International Trade Commission in its Investigation No. 337-TA-602. Such products include SiRFstarIII™ chips that operate with SiRF software that supports SiRFInstantFix™, and/or SiRFLoc® servers, or contains SyncFreeNav functionality.

Life Support Policy and Use in Safety-critical Compliance

CSR's products are not authorised for use in life-support or safety-critical applications. Use in such applications is done at the sole discretion of the customer. CSR will not warrant the use of its devices in such applications.

Performance and Conformance

Refer to www.csrsupport.com for compliance and conformance to standards information.