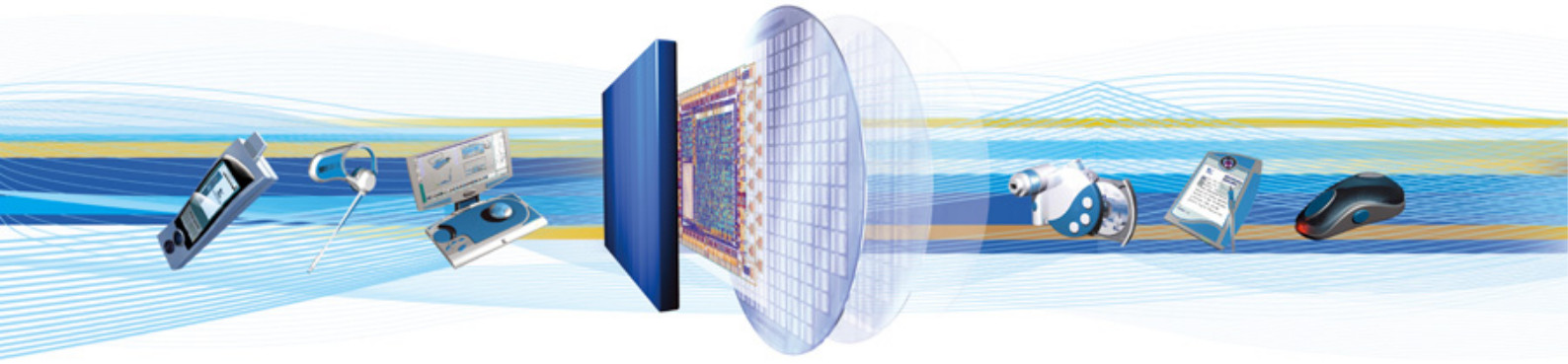
CSR Synergy Framework 3.1.0

SPI Master

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 the CSR Synergy Framework mandates for the protocols communicating via the SPI Master interface.

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

1.2 Prerequisites

This document makes assumes that the reader has a fundamental and thorough understanding of the SPI Specification.

2 SPI Master Interface

2.1 Introduction

Figure 1 illustrates the SPI Master API/interface (denoted by the line containing csr_spi_master.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.

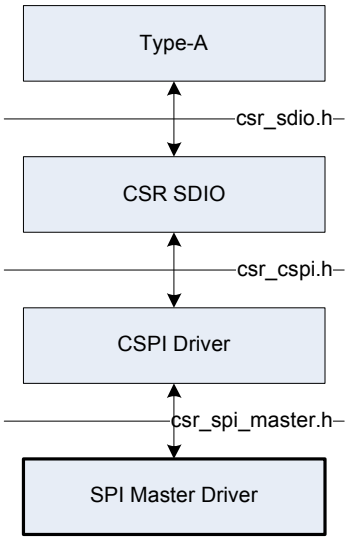


Figure 1: Illustration of SPI Master interface and the BDB2 BSP implementation

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

2.2 General Overview

This section provides a general overview of how the CSPI Driver interacts with the SPI Master implementation as shown in Figure 1. 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 in great lengths.

From the point of view of the CSPI Driver the following steps are required to establish communication with a specific SPI Device.

1. Initialise the SPI Master implementation using the CsrSpiMasterInitialise() call. The SPI Master implementation will indicate how many devices are available and provide specific details about each device allowing the CSPI Driver to do proper bus sharing (if a multi controller setup is available on the given platform)

2. The CSPI Driver will intialise and configure each SPI device using the CSPI protocol.

3. The CSPI Driver is now able to use the SPI Master implementation to transfer data.

2.2.1 CSPI /SPI Master Interaction

The SPI Master implementation is completely unaware of the CSPI Driver and uses the CsrSpiMasterDevice struct instance to abstract away the CSPI Driver. The CsrSpiMasterDevice struct instance contains a specification of a SPI device:

```
typedef struct
```

```
{
    CsrUint8  deviceIndex;
    CsrUint8  busIndex;
    CsrUint32 features;
    void      *driverData;
    void      *priv;
} CsrSpiMasterDevice;
```

The `CsrUint8 deviceIndex` and `CsrUint8 busIndex` uniquely identifies a SPI device. The value of theses parameters indicate the bus topology and are completely defined by the SPI Master implementation. However the implementation must satisfy the following:

- All SPI devices sharing a bus must have the same value for the '`busIndex`' parameter. This is used by the CSPI Driver to allow concurrent transfer on different busses.

The '`features`' parameter indicate special features/requirements that the SPI Master implementation has. The following flags can be specified in the '`features`' bitmap:

```
#define CSR_SPI_MASTER_FEATURE_DMA_CAPABLE_MEM_REQUIRED 0x00000001
```

The '`CSR_SPI_MASTER_FEATURE_DMA_CAPABLE_MEM_REQUIRED`' bit indicates that the SPI Master implementation requires the payload pointer passed to it can be used for DMA transfer.

The field '`priv`' field is only meant to be used by the SPI Master implementation itself and may not be changed by the CSPI Driver.

The '`driverData`' field of this struct can be used by the CSPI Driver to store internal state data about the device.

2.3 Callback Contexts

All callback functions supported by the SPI Master API are called either in low or high level context. It's important to note the difference since these contexts defines what the user of the SPI Master API may and may not do.

2.3.1 High Level Callback

This callback type is used when TRUE is returned from a CsrSpiMasterCallback callback:

```
typedef void (*CsrSpiMasterDsrCallback)(CsrSpiMasterDevice *device);
```

The exact properties are system dependent, but the following criterion shall be met for the high level callback execution context:

- It is possible to set a background interrupt in the CSR Scheduler.

Note: execution may or may not be interruptible in this execution context, and as a consequence suspending on any thread locking mechanisms may deadlock.

2.3.2 Low Level Callback

This callback type is used when an asynchronous data transfer operation completes or an interrupt is signalled. Depending on implementation, this function may be called in low level interrupt context. This means that very few, if any, operating system services can be used safely:

```
typedef CsrBool (*CsrSpiMasterCallback)(CsrSpiMasterDevice *device);
```

The following criteria shall be met for the low level callback execution context:

1. It is possible to start another SPI data transfer operation by calling one of the asynchronous SPI data transfer functions directly from the callback (see CsrSpiMaster*Async() functions described in Section 2.5.13 and 2.5.15)

2. It is possible to acknowledge interrupts by calling the appropriate function directly from the callback (CsrSpiMasterInterruptAcknowledge() function described in section 2.5.4)

3. It is possible to assert and negate chip select by calling the appropriate functions directly from the callback (CsrSpiMasterChipSelectAssert() and CsrSpiMasterChipSelectNegate() functions described in section 2.5.5 and 2.5.6)

Alternatively to item 1, 2 and 3, it is possible to schedule a CsrSpiMasterDsrCallback, which provides a higher level interrupt service context, by returning TRUE from this callback. If this option is exercised, it is not possible to also exercise item 1, 2 and 3 in the same 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.

2.4 SPI Master Bus Sharing

The SPI Master implementation must be able to handle each bus as given by the 'busIndex' field in the CsrSpiMasterDevice struct instance independently. This in effect allows the CSPI driver to handle several SPI devices concurrently as long as they reside on a different bus. Starting an operation on a SPI device is referred to as 'occupying the SPI Master' on the bus the SPI device is attached to and the caller has to wait for the operation to complete before starting another.

2.5 SPI Master API Description

This section describes the SPI Master API functions in great details. Each function description contains:

- Prototype of the function
- Number and type of each parameter
- Return value

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

- CSR_RESULT_SUCCESS
- CSR_RESULT_FAILURE

2.5.1 CsrSpiMasterInitialise

Prototype

```
#include ”csr_spi_master.h”

void CsrSpiMasterInitialise(CsrUint8 *devicesCount, CsrSpiMasterDevice **devices);
```

Description

CsrSpiMasterInitialise initialises the SPI Master driver. This must be called once to initialise the driver, before calling any other function of the interface. Following initialisation all chip select signals shall be negated. The SPI Master must return the number of devices present in the ‘deviceCount’ parameter and array of devices in the ‘devices’ parameter. The length of the array is indicated by the ‘deviceCount’ parameter. See section 2.2.1 for a description of the CsrSpiMasterDevice struct instance.

After initialisation Interrupts are disabled. To fully enable Interrupt reception, the user must write to the INT Enable bit of the CCCR in Card Function 0, as well as call CsrSpiMasterInterruptHandlerRegister to enable the host to receive the interrupts generated by the card.

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
`devicesCount *`	`CsrUint8`	Pointer to CsrUint8 which will contain the number of devices present when the function returns.
`CsrSpiMasterDevice **`	`devices`	Pointer to CsrSpiMasterDevice* which will contain an array of SPI devices when the functions return.

Table 1: Arguments to CsrSpiMasterInitialise

Returns

This function does not return a value.

2.5.2 CsrSpiMasterDeinitialise

Prototype

```
#include ”csr_spi_master.h”

void CsrSpiMasterDeinitialise(void);
```

Description

When the SPI Master driver is no longer required, it can be deinitialised by calling CsrSpiMasterDeinitialise. This must only be called if the SPI Master is not occupied. When CsrSpiMasterDeinitialise returns, no further operation completion callbacks will be triggered, until the SPI Master driver has been initialised by calling CsrSpiMasterInitialise again.

Interrupts are signalled separately, and the CsrSpiMasterInterruptHandlerRegister function must be called to initialise and deinitialise the Interrupt callback mechanism. Calling CsrSpiMasterDeinitialise will *not* cancel or prevent interrupt callbacks.

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

None.

Returns

This function does not return a value.

2.5.3 CsrSpiMasterCallbackRegister

Prototype

```
#include ”csr_spi_master.h”

void CsrSpiMasterCallbackRegister(CsrSpiMasterDevice *device, CsrSpiMasterCallback
interruptCallback, CsrSpiMasterDsrCallback interruptDsrCallback,
CsrSpiMasterDsrCallback operationDsrCallback);
```

Description

CsrSpiMasterInterruptHandlerRegister() register several handlers for a specific device. Passing NULL as a callback parameter will unregister the given handler for the given SPI device.

When an interrupt occurs, interrupts will be masked and the registered ‘`interruptCallback`’ function will be called. No further interrupts will be signalled until CsrSpiMasterInterruptAcknowledge() has been called to unmask the interrupts.

The ‘`interruptDsrCallback`’ will be called if the ‘`interruptCallback`’ return TRUE.

The ‘`operationDsrCallback` will be called if a lowlevel operation callback (See section section 2.5.13 and 2.5.15) return TRUE.

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
`CsrSpiMasterDevice *`	`device`	A device handle that uniquely describes a SPI device
`CsrSpiMasterCallback *`	`interruptCallback`	A lowlevel callback that we will be called when an interrupt occurs from the given SPI device
`CsrSpiMasterDsrCallback *`	`interruptDsrCallback`	A highlevel callback that we will be called if an lowlevel interrupt callback returns TRUE
`CsrSpiMasterDsrCallback *`	`operationDsrCallback`	A highlevel callback that we will be called if a lowlevel operation callback returns TRUE

Table 2: Arguments to CsrSpiMasterInterruptHandlerRegister

Returns

This function does not return a value.

2.5.4 CsrSpiMasterInterruptAcknowledge

Prototype

```
#include ”csr_spi_master.h”

void CsrSpiMasterInterruptAcknowledge(CsrSpiMasterDevice *device);
```

Description

When an interrupt occurs, the registered callback function will be called. When the interrupt has been handled and the function specific interrupt condition on the card has been cleared, interrupts must be unmasked again, by calling CsrSpiMasterInterruptAcknowledge to allow further interrupts to be received. If an interrupt occurs while interrupt callbacks are masked the callback will be postponed until interrupt callbacks are unmasked.

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 safely be called from callbacks that are triggered in response to a data transfer operation or the interrupt callback itself.

Parameters

Type	Argument	Description
`CsrSpiMasterDevice *`	`device`	A device handle that uniquely describes a SPI device

Table 3: Arguments to CsrSpiMasterInterruptAcknowledge

Returns

This function does not return a value.

2.5.5 CsrSpiMasterChipSelectAssert

Prototype

```
#include ”csr_spi_master.h”

void CsrSpiMasterChipSelectAssert(CsrSpiMasterDevice *device);
```

Description

CsrSpiMasterChipSelectAssert() asserts the chip select associated with a specified device. Calling this function occupies the SPI Master for the entire duration of the operation; that is, until the function call returns. No other function that occupies the SPI Master can be called for this duration, and this function shall not be called while other functions are occupying the SPI Master.

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 safely be called from callbacks that are triggered in response to a data transfer operation or the interrupt callback itself.

Parameters

Type	Argument	Description
`CsrSpiMasterDevice *`	`device`	A device handle that uniquely describes a SPI device

Table 4: Arguments to CsrSpiMasterChipSelectAssert

Returns

This function does not return a value.

2.5.6 CsrSpiMasterChipSelectNegate

Prototype

```
#include ”csr_spi_master.h”

void CsrSpiMasterChipSelectNegate(CsrSpiMasterDevice *device);
```

Description

CsrSpiMasterChipSelectNegate() negates the chip select associated with a specified device. Calling this function occupies the SPI Master for the entire duration of the operation; that is, until the function call returns. No other function that occupies the SPI Master can be called for this duration, and this function shall not be called while other functions are occupying the SPI Master.

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 safely be called from callbacks that are triggered in response to a data transfer operation or the interrupt callback itself.

Parameters

Type	Argument	Description
`CsrSpiMasterDevice *`	`device`	A device handle that uniquely describes a SPI device

Table 5: Arguments to CsrSpiMasterChipSelectNegate

Returns

This function does not return a value.

2.5.7 CsrSpiMasterCallbackInhibitEnter

Prototype

```
#include ”csr_spi_master.h”

void CsrSpiMasterCallbackInhibitEnter(CsrSpiMasterDevice *device);
```

Description

The CsrSpiMasterCallbackInhibitEnter function can be used for inhibiting all callbacks (interrupt and operation callbacks) associated with the given SPI device. Other interrupts may also be disabled. This will allow the caller to execute atomically.

This function can be called recursively, but the caller must ensure that calls to CsrSpiMasterCallbackInhibitEnter and CsrSpiMasterCallbackInhibitLeave match evenly.

Important: This function can be called from any context.

Parameters

Type	Argument	Description
`CsrSpiMasterDevice *`	`device`	A device handle that uniquely describes a SPI device

Table 6: Arguments to CsrSpiMasterCallbackInhibitEnter

Returns

This function does not return a value.

2.5.8 CsrSpiMasterCallbackInhibitLeave

Prototype

```
#include "csr_spi_master.h"

void CsrSpiMasterCallbackInhibitLeave(CsrSpiMasterDevice *device);
```

Description

The CsrSpiMasterCallbackInhibitLeave function can be used for allowing callbacks (interrupt and operation callbacks) associated with the given SPI device.

This function can be called recursively, but the caller must ensure that calls to CsrSpiMasterCallbackInhibitEnter and CsrSpiMasterCallbackInhibitLeave match evenly.

Important: This function can be called from any context.

Parameters

Type	Argument	Description
`CsrSpiMasterDevice *`	`device`	A device handle that uniquely describes a SPI device

Table 7: Arguments to CsrSpiMasterCallbackInhibitLeave

Returns

This function does not return a value.

2.5.9 CsrSpiMasterPowerOff

Prototype

```
#include "csr_spi_master.h"

void CsrSpiMasterPowerOff(CsrSpiMasterDevice *device);
```

Description

The CsrSpiMasterPowerOff function can be used for powering off a SPI device. Once this has been done, no more operations may be issued to the device before it has been powered back on using `CsrSpiMasterPowerOn`. A device can only be powered off when there are no pending operations.

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
`CsrSpiMasterDevice *`	`device`	A device handle that uniquely describes a SPI device

Table 8: Arguments to CsrSpiMasterPowerOff

Returns

This function does not return a value.

2.5.10 CsrSpiMasterPowerOn

Prototype

```
#include "csr_spi_master.h"

CsrResult CsrSpiMasterPowerOn(CsrSpiMasterDevice *device);
```

Description

CsrSpiMasterPowerOn is used by the caller to power a SPI device back on when it has previously been powered off using CsrSpiMasterPowerOff. The return value of this function can be used for determining if power has actually been restored or if the card was never powered down.

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
CsrSpiMasterDevice *	device	A device handle that uniquely describes a SPI device

Table 9: Arguments to CsrSpiMasterPowerOn

Returns

If the device had been powered off and is now been powered on, the function returns

CSR_RESULT_SUCCESS, otherwise it returns CSR_SPI_MASTER_RESULT_DEVICE_NOT_RESET.

2.5.11 CsrSpiMasterBusClockFrequencySet

Prototype

```
#include "csr_spi_master.h"

void CsrSpiMasterBusClockFrequencySet(CsrSpiMasterDevice *device, CsrUint32
*frequency);
```

Description

This function sets the desired frequency of the SPI Master clock. The actual frequency might deviate, based on the properties and capabilities of the specific host hardware. Calling this function occupies the SPI Master for the entire duration of the operation; that is, until the function call returns. No other function that occupies the SPI Master can be called for this duration, and this function shall not be called while other functions are occupying the SPI Master.

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
CsrSpiMasterDevice *	device	A device handle that uniquely describes a SPI device
CsrUint32*	frequency	A pointer to the desired frequency. The actual frequency set by the implementation is also returned in this parameter.

Table 10: Arguments to CsrSpiMasterBusClockFrequencySet

Returns

This function does not return a value.

2.5.12 CsrSpiMasterRead

Prototype

```
#include ”csr_spi_master.h”

void CsrSpiMasterRead(CsrSpiMasterDevice *device, CsrUint8 *data, CsrUint32
length);
```

Description

CsrSpiMasterRead() clock a specified amount of bytes in. This call blocks until the operation is complete. Calling this function occupies the SPI Master for the entire duration of the operation; that is, until the function call returns. No other function that occupies the SPI Master can be called for this duration, and this function shall not be called while other functions are occupying the SPI Master.

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
`CsrSpiMasterDevice *`	`device`	A device handle that uniquely describes a SPI device
`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 11: Arguments to CsrSpiMasterRead

Returns

This function does not return a value.

2.5.13 CsrSpiMasterReadAsync

Prototype

```
#include ”csr_spi_master.h”

void CsrSpiMasterReadAsync(CsrSpiMasterDevice *device, CsrUint8 *data, CsrUint32
length, CsrSpiMasterCallback operationCallback);
```

Description

CsrSpiMasterReadAsync() receive a specified amount of bytes from the device. This function returns immediatly, and the specified callback function will be called when the operation completes. Calling this function occupies the SPI Master for the entire duration of the operation; that is, until the callback function is called. No other function that occupies the SPI Master can be called for this duration, and this function shall not be called while other functions are occupying the SPI Master.

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 safely be called from callbacks that are triggered in response to a data transfer operation or the interrupt callback itself.

Parameters

Type	Argument	Description
`CsrSpiMasterDevice *`	`device`	A device handle that uniquely describes a SPI device
`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.
`CsrSpiMasterCallback`	`operationCallback`	The callback function to call when the operation completes.

Table 12: Arguments to CsrSpiMasterReadAsync

Returns

This function does not return a value.

2.5.14 CsrSpiMasterWrite

Prototype

```
#include ”csr_spi_master.h”

void CsrSpiMasterWrite(CsrSpiMasterDevice *device, CsrUint8 *data, CsrUint32
length);
```

Description

CsrSpiMasterWrite() clock a specified amount of bytes out. This call blocks until the operation is complete. Calling this function occupies the SPI Master for the entire duration of the operation; that is, until the function call returns. No other function that occupies the SPI Master can be called for this duration, and this function shall not be called while other functions are occupying the SPI Master.

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
`CsrSpiMasterDevice *`	`device`	A device handle that uniquely describes a SPI device
`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 13: Arguments to CsrSpiMasterWrite

Returns

This function does not return a value.

2.5.15 CsrSpiMasterWriteAsync

Prototype

```
#include ”csr_spi_master.h”
```

```
void CsrSpiMasterWriteAsync(CsrSpiMasterDevice *device, CsrUint8 *data, CsrUint32
length, CsrSpiMasterCallback operationCallback);
```

Description

CsrSpiMasterWriteAsync() transmits a specified amount of bytes to the device. This function returns immediatly, and the specified callback function will be called when the operation completes. Calling this function occupies the SPI Master for the entire duration of the operation; that is, until the callback function is called. No other function that occupies the SPI Master can be called for this duration, and this function shall not be called while other functions are occupying the SPI Master.

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 safely be called from callbacks that are triggered in response to a data transfer operation or the interrupt callback itself.

Parameters

Type	Argument	Description
`CsrSpiMasterDevice *`	`device`	A device handle that uniquely describes a SPI device
`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.
`CsrSpiMasterCallback`	`operationCallback`	The callback function to call when the operation completes.

Table 14: Arguments to CsrSpiMasterReadAsync

Returns

This function does not return a value.

3 Document References

Ref	Title

Terms and Definitions

[SYN-FRW-COAL-API]	CSR Synergy Framework COAL API. Doc. api-0001-coal
CSR	Cambridge Silicon Radio

Document History

Revision	Date	History
1	30 NOV 09	Ready for release 2.0.0
2	20 APR 10	Ready for release 2.1.0
3	OCT 10	Ready for release 2.2.0
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.