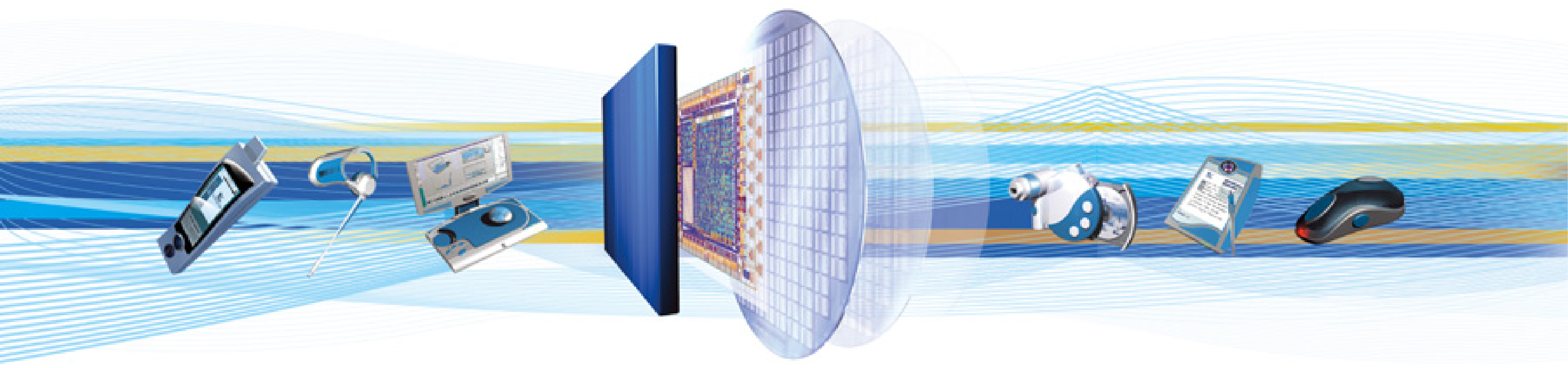
CSR Synergy Framework 3.1.0

UART

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

Appendix

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 its protocols communicating via the UART interface. These protocols currently include BlueCore Serial Protocol (BCSP) and H4 Deep Sleep (H4DS). While the current implemented transports, BCSP and H4DS, does not run in parallel, since only one can interface to a BlueCore at a time, talking to more chips like eg. GPS at the same time as interfacing the BlueCore is required. Therefore, the UART interface is prepared for addition of more transports and for simultaneous transports.

BCSP adds possibilities for *logical communication channels*. The logical communication channels are used for routing signals to and from the host and host controller, and they make it possible to establish a communication path between components in the host and host controller. BCSP makes it possible to use the low-power modes of the BlueCore® chip. Furthermore, BCSP offers reliable communication with retransmission of messages in case of corruption. The BCSP protocol is a CSP specific protocol resembling the H5 / 3-Wire UART protocol.

H4DS is an alternative to BCSP for handling the communication between the host and the host controller. One of the main differences between H4DS and BCSP is that BCSP facilitates retransmission of corrupt messages whereas H4DS assumes error free communication, meaning that H4DS might have a slightly higher throughput, but will fail if a transmission error happens. Due to this BCSP is the preferred communication over serial lines.

For a more thorough description of the basic structure and mechanisms of the protocols running via UART, please refer to [BCSP] and [H4DS] respectively.

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

2 UART Interface

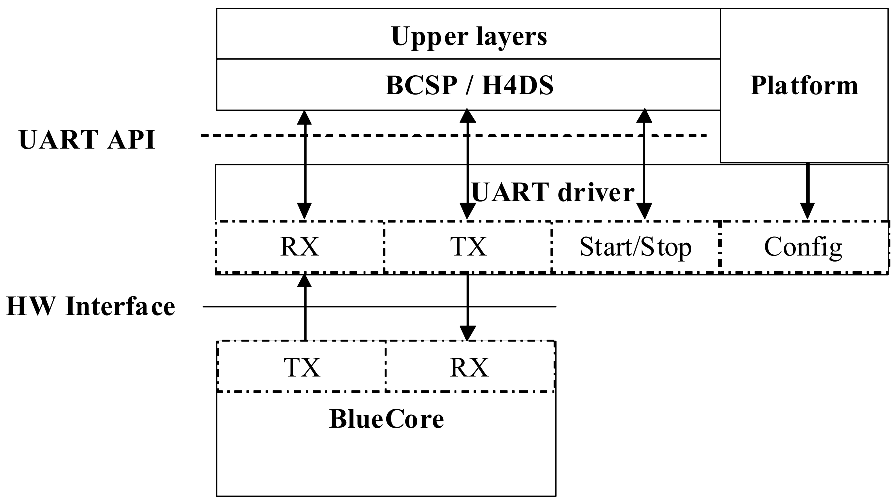


Figure 1: Illustration of where the UART API applies

2.1 Introduction

The UART API is, as mentioned, utilized both by the BCSP and H4DS protocols. These protocols have some shared functionality and some individual functionality. In the following the subset shared by both BCSP and H4DS will be described first and afterwards the additional individual features only relevant for each protocol will be described. As a minimum, full functionality for all the shared functions and the function subset also needed for the chosen transport protocol must be implemented.

The way the UART API handles multi instances is by means of a void pointer handle. This pointer is used for telling the UART driver which instance this is for. The content of the pointer is completely transparent for the transports, it is solely decided by the implementer of the UART driver.

When starting up the platform a uart handle can be assigned to either BCSP or H4DS through CsrTmBlueCoreRegisterUartHandleBcsp(void *handle) or CsrTmBlueCoreRegisterUartHandleH4ds(void*handle) respectively. These protocols will then make sure always to use that handle when interfacing the UART driver from there after. If multi instance is not needed (eg. if only one chip is connected via the CSR Synergy Framework) on a platform, the handle parameter can just be ignored by the UART driver and of course it is not necessary to supply a handle to the transports in this case either.

The prototype for the two functions CsrTmBlueCoreRegisterUartHandleBcsp(void *handle) and CsrTmBlueCoreRegisterUartHandleH4ds(void*handle), can be found in csr_tm_bluecore_bcsp.h and csr_tm_bluecore_h4ds.h respectively.

Besides the required functionality some platforms also needs the ability to reconfigure the UART, e.g. changing the baudrate, between initial startup (Bootstrapping) and normal operation. Since this functionality is not needed on all platforms and because it is not needed by the transports this functionality is not covered by the UART API. However for the purpose of testing and verification the BSPs provided with the CSR Synergy Framework implement such an API which is used by demo applications and which can serve as an inspiration for how such an API could be designed on other platforms. The Configuration API implemented is described in Appendix A.

A set of different parameters must be tuned in order for the UART implementation to work correctly with the different transport protocols. These settings are not something which affects the UART API as such, but they are as mentioned required in order for the UART implementation to work. These are settings such as the UART RX/TX buffer sizes, retransmission settings etc. How these values are derived is described in Appendix B "Optimizing the UART for different transports".

A locked requirement on the UART is its default HW configuration which needs to be used the first time a UART connection is opened to the BlueCore. These are settings as the initial baud rate (can be auto baud!), number of data-, stop- and parity- bits, odd/even parity, hardware flow control etc.

Default values for BCSP are: 8 data bits, 1 stop bit, 1 parity bit, even parity and hardware flow control disabled.
Default values for H4DS are: 8 data bits, 1 stop bit, no parity bits and hardware flow control enabled.

For the exact baud rate values please refer to the datasheet of the exact BlueCore chip used in your product. It should be noted that these default settings can be changed if required during bootstrapping by setting the ps keys `PSKEY_UART_CONFIG_BCSP(0x01BF) and` PSKEY_UART_CONFIG_H4PLUS(0x0077) for BCSP and H4DS respectively. The values of the these ps keys are defined as a unsigned 16 bit pattern. For both BCSP and H4DS the upper 8 bits must always be set as 0x08. The description of the rest of the bits are:

Bit	Meaning
0	0 => one stop bit, 1 => two stop bits.
1	0 => no parity bits, 1 => one parity bit.
2	0 => odd parity, 1 => even parity.
3	0 => h/w flow control disabled, 1 => enabled.
4	Set to 0.
5	0 => RTS deasserted, 1 => RTS asserted.
6	Set to 0.
7	0 => H4DS, 1 => BCSP.

The CSR Synergy Framework is as described delivered with a set of BSPs, one for Windows-x86 (found in /bsp/ports/pcwin), one for Linux-x86 (found in /bsp/ports/pclin) and one for Nucleus-ARM (found in /bsp/ports/bdb2). These BSP all implements full functioning and compliant implementations of the complete CSR Synergy Framework UART API. Please refer to those BSPs (under $(BSP_ROOT)/src/low_level/uart) for examples on how the UART API could be implemented on a platform.

2.2 Shared BCSP and H4DS Functionality

2.2.1 CsrUartDrvStart

Prototype

```
#include "csr_serial_com.h"

CsrBool CsrUartDrvStart(void *handle, CsrUint8 reset);
```

Description

This function is used for starting up the UART. A call to this function happens before any data communication takes place with the UART driver. This is the second function called after system startup (where the driver is assumed to be in its default state and CsrUartDrvRegister has been called to register a callback and a background interrupt handle) or directly after a call to `CsrUartDrvStop()`, which happens in relation to restarting the UART communication.

Parameters

Type	Argument	Description
`void *`	handle	This pointer is used for telling the UART driver which instance this call is for.
`CsrUint8`	`reset`	This parameter tells the driver whether this is the initial start or a subsequent start call. This functionality is useful when baudrates are changed between bootstrapping and normal operation. First time this function is called after system startup the value will be zero (this is before bootstrapping). And in any subsequent calls it will be higher than zero.

Table 1: Arguments to CsrUartDrvStart

Returns

TRUE, if the UART starts successfully. FALSE, if the UART fails to start for any reason. In case of failure, the state of the UART must be the same as before this function call.

2.2.2 CsrUartDrvStop

Prototype

```
#include "csr_serial_com.h"

CsrBool CsrUartDrvStop(void *handle);
```

Description

This function is called to stop (close) the UART driver. It un-does all actions performed by `CsrUartDrvStart()` eg. free any allocated memory.

Note: It is expected that the UART configuration does NOT change when this function is called

Parameters

Type	Argument	Description
`void *`	handle	This pointer is used for telling the UART driver which instance this call is for.

Table 2: Arguments to CsrUartDrvStop

Returns

TRUE, if the UART stops successfully. FALSE otherwise.

2.2.3 CsrUartDrvTx

Prototype

```
#include "csr_serial_com.h"

CsrBool CsrUartDrvTx(void *handle, char *data, CsrUint32 dataLength, CsrUint32
*numSent);
```

Description

This function is used for pushing data onto the UART TX buffer.

Parameters

Type	Argument	Description
`void *`	handle	This pointer is used for telling the UART driver which instance this call is for.
`char*`	`data`	Pointer to the data payload which must be sent
`CsrUint32`	`dataLength`	Number of bytes in theData pointer which should be written to the UART TX buffer
`CsrUint32 *`	`numSent`	If the UART TX buffer can hold all `dataLength` bytes this parameter must be set to `dataLength` by the function, otherwise it must be set to zero.

Table 3: Arguments to CsrUartDrvTx

Returns

If the UART TX buffer cannot hold an extra `dataLength` bytes when this function is called it must return FALSE. If all the bytes are pushed to the TX buffer the function must return TRUE.

2.2.4 CsrUartDrvRegister

Prototype

```
#include "csr_serial_com.h"

void CsrUartDrvRegister(void *handle, CsrUartDrvDataRx rxDataFn, CsrBgint
rxBgintHdl);
```

Where CsrUartDrvDataRx is a call back function prototype defined like:

```
typedef CsrUint32 (*CsrUartDrvDataRx)(CsrUint8 *buf, CsrUint32 len);
```

Description

This function call is used for supplying the UART driver with information about how to deliver and notify the CSR Synergy Framework about new incoming data on the UART.

Note: The use of this function is different from BCSP to H4DS.

Parameters

Type	Argument	Description
`void *`	handle	This pointer is used for telling the UART driver which instance this call is for.
`CsrUartDrvDataRx`	`rxDataFn`	For BCSP this parameter provides the call back function, which should be called from from CsrUartDrvRx(). Note: The buffer provided, when calling this call back function, is purely owned by the UART driver. BCSP only copies, at most "`dataLength`" number of bytes from the given buffer into its own BCSP RX buffer. For H4DS this parameter is set as NULL and is not used.
`CsrBgint`	`rxBgintHdl`	This is the background interrupt handle the UART driver needs to set, when it has new incoming data available for the CSR Synergy Framework. The way to set the bgInt is by means of CsrBgIntSet(), for more information on this function please refer to [SYN-FRW-COAL-API]. Note: For BCSP the optimum way of setting this rx bgInt is by examine the incoming data for `0xC0` and when ever such a byte character is detected this rx bgInt should be set. If for any reason such detection is not possible in the driver, this bgInt should be set every time there is new data available. For H4DS there is no special character to search for so here the bgInt should be set every time there is new data available.

Table 4: Arguments to CsrUartDrvRegister

Returns

Nothing.

2.2.5 CsrUartDrvGetBaudrate

Prototype

```
#include "csr_serial_com.h"

CsrUint32 CsrUartDrvGetBaudrate(void *handle);
```

Description

This function is used for asking the UART driver about the current configured baud rate of the driver.

Note: BCSP uses this only when automatic configuration of the retransmission timer value is enabled.

Parameters

Type	Argument	Description
`void *`	handle	This pointer is used for telling the UART driver which instance this call is for.

Table 5: Arguments to CsrUartDrvGetBaudrate

Returns

The current configured baud rate of the driver eg. 115200 or 9216000.

2.3 BCSP only functionality

2.3.1 CsrUartDrvRx

Prototype

```
#include "csr_serial_com.h"

void CsrUartDrvRx(void *handle);
```

Description

The CSR Synergy Framework BCSP implementation is by design not re-entrant, so when the UART driver receives a message from the BlueCore it must inform the CSR Synergy Framework Scheduler by setting the `rxBgintHdl` received in `CsrUartDrvRegister()`. When the Scheduler is informed it in turn calls this `CsrUartDrvRx()` function. This functionality is illustrated in Figure 2.

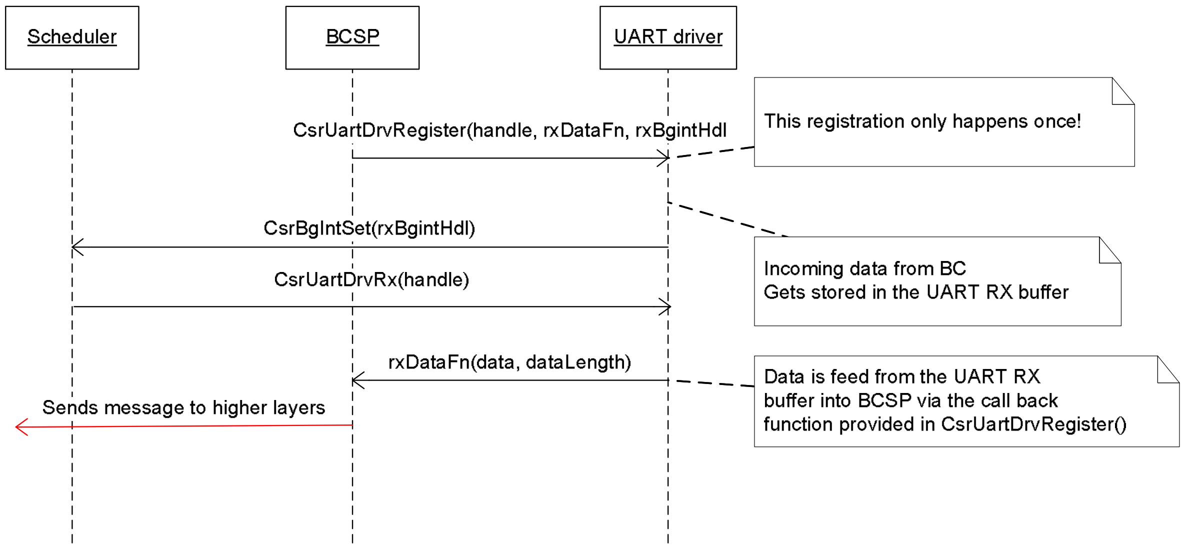

Scheduler
BCSP
UART driver
CsrUartDrvRegister(handle, rxDataFn, rxBgintHdl
This registration only happens once!
CsrBgIntSet(rxBgintHdl)
CsrUartDrvRx(handle)
Incoming data from BC
Gets stored in the UART RX buffer
rxDataFn(data, dataLength)
Data is feed from the UART RX
buffer into BCSP via the call back
function provided in CsrUartDrvRegister()
Sends message to higher layers

Figure 2: Incoming BCSP data example

The function must take the available bytes out from the UART RX buffer and push them to the CSR Synergy Framework BCSP library by calling the call back function received in `CsrUartDrvRegister()`. This call back function then returns the number of bytes it actually consumed of the total number of bytes fed into it.

If bytes are still available in the UART RX buffer after the call back returns, the `CsrUartDrvRx()` function must reschedule itself again by setting the `rxBgintHdl` again.

Parameters

Type	Argument	Description
`void *`	handle	This pointer is used for telling the UART driver which instance this call is for.

Table 6: Arguments to CsrUartDrvRx

Returns

Nothing.

2.4 H4DS only functionality

2.4.1 CsrUartDrvGetRxAvailable

Prototype

```
#include "csr_serial_com.h"

CsrUint32 CsrUartDrvGetRxAvailable(void *handle);
```

Description

This is used by H4DS to monitor the UART RX buffer level.

Parameters

Type	Argument	Description
`void *`	handle	This pointer is used for telling the UART driver which instance this call is for.

Table 7: Arguments to CsrUartDrvGetRxAvailable

Returns

The amount of bytes waiting to be read in the UART RX buffer.

2.4.2 CsrUartDrvLowLevelTransportRx

Prototype

```
#include "csr_serial_com.h"

CsrUint32 CsrUartDrvLowLevelTransportRx(void *handle, CsrUint32 noOfBytes,CsrUint8
*buffer);
```

Description

This function is used for retrieving data from the UART driver when something is available. When data is received the RX implementation in the UART driver must set the `rxBgintHdl` received in `CsrUartDrvRegister()`. When the Scheduler is informed it in turn make the H4DS library call this `CsrUartDrvLowLevelTransportRx()` function. If one call to this function does not empty the buffer completely the function must set the `rxBgintHdl` again. This behaviour is illustrated in Figure 3.

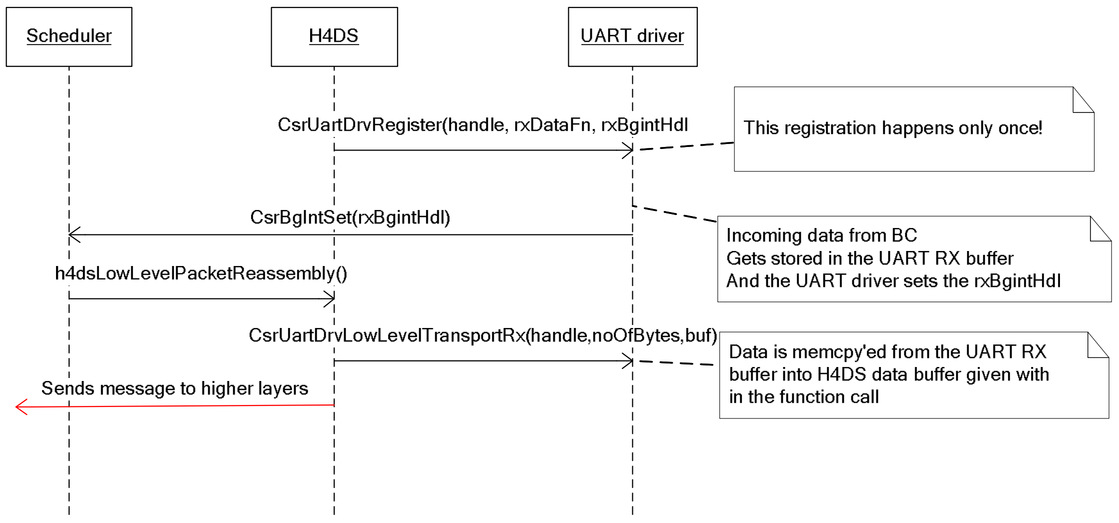

Scheduler
H4DS
UART driver
CsrUartDrvRegister(handle, rxDataFn, rxBgintHdl
This registration happens only once!
CsrBgIntSet(rxBgintHdl)
Incoming data from BC
Gets stored in the UART RX buffer
And the UART driver sets the rxBgintHdl
h4dsLowLevelPacketReassembly()
CsrUartDrvLowLevelTransportRx(handle,noOfBytes,buf)
Data is memcpy'ed from the UART RX buffer into H4DS data buffer given with in the function call
Sends message to higher layers

Figure 3: Incoming H4DS data example

Parameters

Type	Argument	Description
`void *`	handle	This pointer is used for telling the UART driver which instance this call is for.
`CsrUint32`	`bufferLength`	The amount of bytes which can be copied into the data buffer

`CsrUint8 *`	`buffer`	The data buffer that incoming data should be copied to. Note: This buffer is owned purely by H4DS and should therefore not be free'ed by the UART driver.

Table 8: Arguments to CsrUartDrvLowLevelTransportRx

Returns

The actual number of bytes written to the data buffer.

2.4.3 CsrUartDrvGetTxSpace

Prototype

```
#include "csr_serial_com.h"

CsrUint32 CsrUartDrvGetTxSpace(void *handle);
```

Description

This function is used for getting the amount of free space in the UART TX buffer at present time.

Parameters

Type	Argument	Description
`void *`	handle	This pointer is used for telling the UART driver which instance this call is for.

Table 9: Arguments to CsrUartDrvGetTxSpace

Returns

The amount of bytes possible to store in the UART TX buffer at present time.

2.4.4 CsrUartDrvLowLevelTransportTxBufLevel

Prototype

```
#include "csr_serial_com.h"

CsrUint32 CsrUartDrvLowLevelTransportTxBufLevel(void *handle);
```

Description

This function is used for investigating how many bytes are currently waiting to be sent.

Parameters

Type	Argument	Description
`void *`	handle	This pointer is used for telling the UART driver which instance this call is for.

Table 10: Arguments to CsrUartDrvLowLevelTransportTxBufLevel

Returns

The current amount of bytes in the UART TX buffer waiting to be sent.

Appendix A API for configuring the UART

As mentioned earlier configuring/reconfiguring the UART is not a requirement for all platforms and hence it is not a requirement the CSR Synergy Framework mandates the platform to implement. For the purpose of testing and verification the BSPs provided with the CSR Synergy Framework have implemented an API which is used by the demo applications and which can serve as an inspiration for how such an API could be designed on other platforms.

The prototype for this API can be found in `csr_serial_init.h`. This API is named `CsrUartDrvConfigure()` and is defined as.

An example of how this API is used is illustrated in Figure 4.

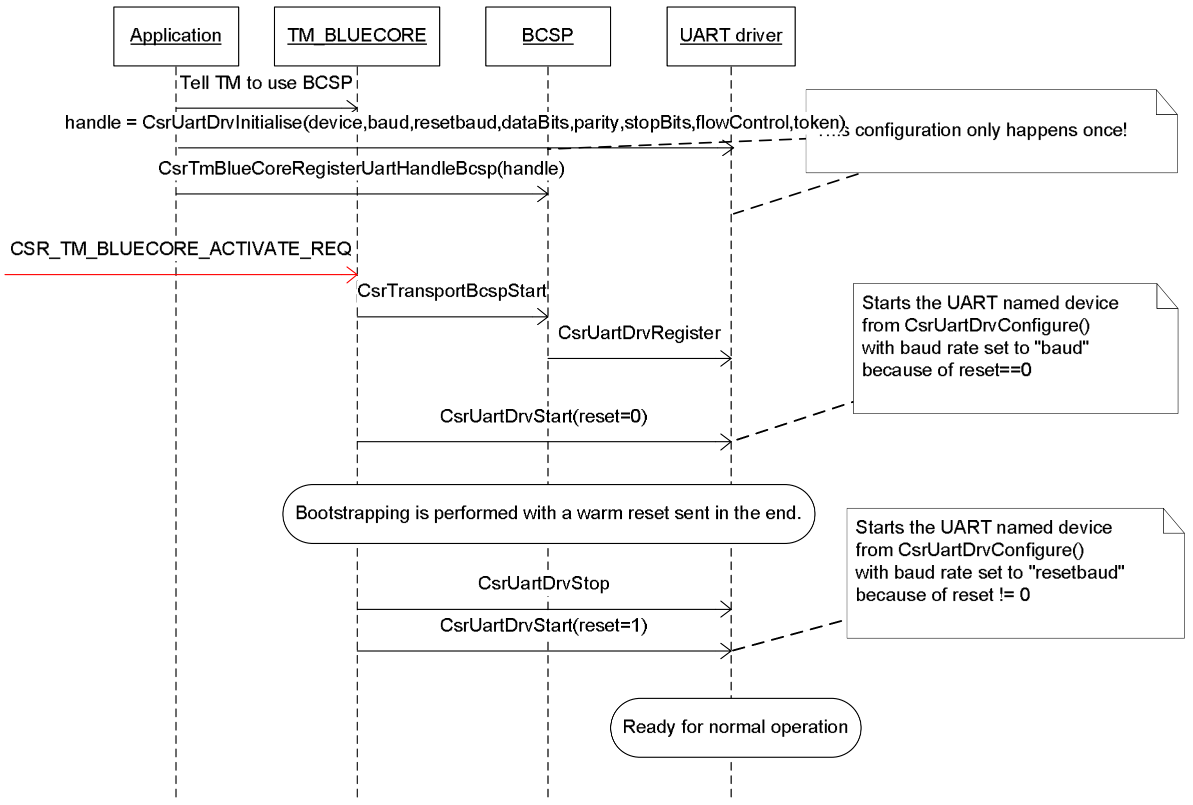

Application
TM_BLUECORE
BCSP
UART driver
Tell TM to use BCSP
handle = CsrUartDrvInitialise(device,baud,resetbaud,dataBits,parity,stopBits,flowControl,token)
…configuration only happens once!
CsrTmBlueCoreRegisterUartHandleBcsp(handle)
CSR_TM_BLUECORE_ACTIVATE_REQ
CsrTransportBcspStart
CsrUartDrvRegister
Starts the UART named device from CsrUartDrvConfigure() with baud rate set to "baud" because of reset==0
CsrUartDrvStart(reset=0)
Bootstrapping is performed with a warm reset sent in the end.
Starts the UART named device from CsrUartDrvConfigure() with baud rate set to "resetbaud" because of reset != 0
CsrUartDrvStop
CsrUartDrvStart(reset=1)
Ready for normal operation

Figure 4: Example of the usage of CsrUartDrvConfigure

Prototype

```
#include "csr_serial_init.h"

void *CsrUartDrvInitialise(const CsrCharString *device,

                           CsrUint32 baud, CsrUint32 resetBaud,

                           CsrUartDrvDataBits dataBits,

                           CsrUartDrvParity parity,

                           CsrUartDrvStopBits stopBits,
                           CsrBool flowControl,

                           const CsrUint8 *token);
```

Description

This function is used for setting the configuration data for the UART.

Parameters

Type	Argument	Description
`const CsrCharString *`	device	The name of the device eg `COM1`.
`CsrUint32`	`baud`	Specify the initial baud rate to be used prior to bootstrapping ie when `CsrUartDrvStart()` is called with `reset=0`
`CsrUint32`	`resetbaud`	Specified the baud rate to be used after the bootstrapping has completed ie when `CsrUartDrvStart()` is called with `reset > 0`.
`CsrUartDrvDatabits`	`dataBits`	Number of data bits (see header)
`CsrUartDrvParity`	`parity`	Parity (see header)
`CsrUartDrvStopBits`	`stopBits`	Number of stop bits (see header)
`CsrBool`	`flowControl`	Whether to use hardware flow control
`const CsrUint8 *`	`token`	If not NULL, instruct the driver to use the supplied token. If a token is supplied the receive background interrupt will only be triggered when this token is encountered in the received data stream, instead of every time there is new data available.

Table 11: Arguments to CsrUartDrvInitialise

Returns

Nothing.

Appendix B Optimizing the UART for different transports

A different set of parameters exists which must be tuned in order for the UART implementation to work correctly with the different transport protocols. These are settings as the UART RX/TX buffer sizes and the retransmission settings. In the following the retransmission settings for the BCSP transport will be explained and afterwards the calculation of the UART RX/TX buffer sizes for both transports will be explained.

Retransmission time for BCSP:

The BCSP retransmission timer is defined by `CSR_BCSP_RETRANSMISSION_TIMER` in `csr_usr_config_default.h` and is given in micro seconds. The default value is 250000 us. If the BCSP retransmission timer is changed on the host side it should also be changed on the BlueCore side by means of the `PSKEY_UART_ACK_TIMEOUT(0x01C4)` which is given in ms. Additional it is important that this retransmission timer value is set lower than the `PSKEY_UART_SLEEP_TIMEOUT(0x0222)`. By default it is recommend to set the retransmission timer at least 4 times lower than the sleep timeout.

The BCSP retransmission timer depends on many things, like baud rate, number of start, stop and parity bits, size of transmitted packages, how long time the BlueCore has been configured to wait before going into deep-sleep, number of packets that can be send before retransmission etc.

The formula below is based on the assumption that the packet length used for transmitting and receiving is the same.

$$t_{trans} = \frac{((S_{DATA} + S_{BCSP} + S_{CRC}) \times 2 + S_{0xCO})) \times \#bits}{baudrate} + L_{platform}$$

where:

- the “x 2” is due to byte stuffing
- S_{DATA} is the maximum size of any given data payload that is to be transmitted/received in any Synergy tech in the system. Please refer to the user guide of the technologies running on top of the CSR Synergy Framework.
- S_{BCSP} is the size of the BCSP header which is 4 bytes
- S_{CRC} is the number of CRC bytes. This is 2 bytes if CRC is used else 0 bytes
- S_{0xC0} is the number of 0xC0 bytes being send, which is 2 bytes
- #bits is the number of bits used, that is the number of data, start, stop and parity bits
- baudrate is the baudrate used in bits/sec.
- $L_{platform}$ is the latency time it takes on a given platform from the data packet leaves the BCSP TX buffer till the actually transmission starts on the UART + the time it takes the acknowledge message getting from the UART RX buffer and back into the BCSP RX buffer. Eg. the latency related to context switching.

The BCSP retransmission timer value will be a trade-off between the BlueCore going into deep-sleep (triggered by sleep timeout) as often as possible and being able to get it out of deep-sleep as fast as possible.

Important: There is a lower bound for the retransmission timer which should be kept. It should not be set lower than the highest value of the following two requirements:

1) the time it takes the BC to wakeup from deep-sleep which is approx. 5 ms

2) 2 x the precision of the CsrGetTime() implementation

UART RX/TX buffer sizes for BCSP:

In order to get the BCSP protocol running, the UART TX and RX buffers need to have a minimum size. The minimum buffer size in a direction is calculated as:

$$S_{buffer} = ((S_{DATA} + S_{BCSP} + S_{CRC}) \times 2 + S_{0xCO}) \times S_{SLIDING\_WINDOW} + S_{ACK} + S_{UNRELIABLE}$$

where:

- the “x 2” is due to BCSP-SLIP byte stuffing
- S_{DATA} is the maximum size of any given data payload that is to be transmitted/received in any Synergy tech in the system. Please refer to the user guide of the technologies running on top of the CSR Synergy Framework in your system.
- S_{BCSP} is the size of the BCSP header which is 4 bytes
- S_{CRC} is the number of CRC bytes. This is 2 bytes if CRC is used else 0 bytes[1]
- S_{0xC0} is the number of 0xC0 bytes being send, which is 2 bytes
- $S_{SLIDING\_WINDOW}$ is the size of the sliding windows. Default is 4 in both directions[2]
- S_{ACK} is the size of an acknowledge message which is approx. 10 bytes. The number of ACKs depends on the $S_{SLIDING\_WINDOW}$ in the opposite direction.
- $S_{UNRELIABLE}$ is the maximum size of any given unreliable data payload that is to be transmitted/received in any Synergy tech in the system. Please refer to the user guide of the technologies running on top of the CSR Synergy Framework.[3]

If different packet sizes are used for Tx and Rx the TX and RX buffers may differ in size.

UART RX/TX buffer sizes for H4DS:

In order to get the H4DS protocol running, the UART TX and RX buffers also require a minimum size. The minimum buffer size in a direction is calculated as:

$$S_{buffer} = (S_{ACL} + S_{HCI} + S_{H4DS}) \times S_{ACL\_PKT} + (S_{SCO} + S_{HCI} + S_{H4DS}) \times S_{SCO\_PKT} + S_{FP}$$

where:

- S_{H4DS} is the size of the H4DS header which is 1 byte
- S_{HCI} is the size of a HCI header which is 4 bytes.
- S_{ACL} is the maximum allowed size of an ACL packet[4]
- S_{ACL_PKT} is the maximum allowed number of outstanding ACL packets[5]
- S_{SCO} is the maximum allowed size of a SCO packet[6]

[1] In the TX direction, this depends on the define `CSR_ABCSP_TXCRC` found in `csr_usr_config_default.h` and in the RX direction on the PSKEY_UART_TX_CRCS(0x01C3)
[2] Defined in TX direction by the define `CSR_ABCSP_TXWINSIZE` found in `csr_usr_config_default.h` and in the RX direction by the PSKEY_UART_TX_WINDOW_SIZE(0x01C6)
[3] An example of unreliable data defining the upper limit of could be SCO data packets which only is used if SCO over HCI is used in the system. So normally this can be assumed to be 0.
[4] Is defined in TX direction by PSKEY_H_HC_FC_MAX_ACL_PKT_LEN(0x0011) and in the RX direction as the lowest value of the PSKEY_HOSTIO_PROTOCOL_INFO6 (0x019A) and what is sent in the HCI_HOST_BUFFER_SIZE_CMD.
[5] Is defined in TX direction by PSKEY_H_HC_FC_MAX_ACL_PKTS(0x0013) and in the RX direction since there is no host controller to host flow control a sufficiently high value should be assumed.

- S_{SCO_PKT} is the maximum allowed number of outstanding SCO packets[7]
- S_{FP} is the maximum allowed number of bytes used by Fastpipe[8]. If Fastpipe is not used in the system this value can be assumed to be 0.

[6] Is defined in TX direction by PSKEY_H_HC_FC_MAX_SCO_PKT_LEN(0x0012) and in the RX direction as the lowest value of the PSKEY_HOSTIO_PROTOCOL_INFO7 (0x019B) and what is sent in the HCI_HOST_BUFFER_SIZE_CMD.
[7] Is defined in TX direction by PSKEY_H_HC_FC_MAX_SCO_PKTS(0x0014) and in the RX direction since there is no host controller to host flow control a sufficiently high value should be assumed.
[8] Is defined in TX direction by PSKEY_FASTPIPE_LIMIT_CONTROLLER(0x01EF) and in RX direction it is defined by the Synergy technologies using fastpipe so please refer to their user guides.

3 Document References

Ref	Title

Terms and Definitions

[SYN-FRW-COAL-API]	CSR Synergy Framework COAL API. Doc. Api-0001-coal
[BCSP]	**BCSP BlueCore Serial Protocol (bcore-sp-012Pb)**
[H4DS]	**H4DS Implementation (bcore-me-035Pc)**
BCSP	BlueCore Serial Protocol
H4DS	HCI UART Transport Layer Protocol (H4) with Deep Sleep
BlueCore®	Group term for CSR's range of Bluetooth wireless technology chips
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	OCT 10	Ready for release 2.2.0
5	NOV 10	Ready for release 3.0.0

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.