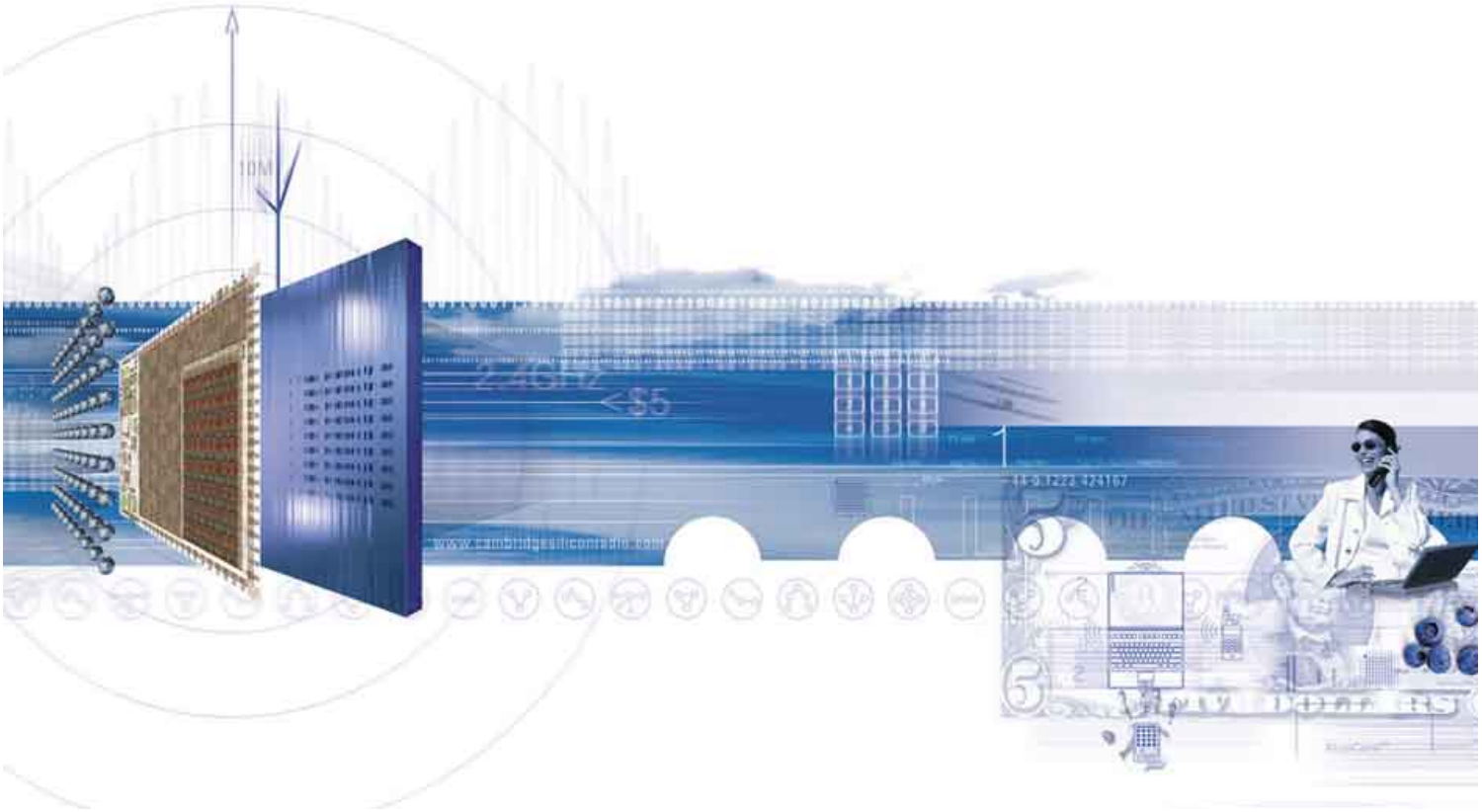
BlueCore ä 01

Porting the BCSP Stack

AN002

May 2000



CSR

Unit 300 Cambridge Science Park
Milton Road
Cambridge
CB4 0XL
United Kingdom

Registered in England 3665875

Tel: +44 (0)1223 424167
Fax: +44 (0)1223 424178

www.csr.com

Introduction

The CSR Bluetooth chip, **BlueCore01**™, uses a UART link to connect to its host. **BlueCore01** Serial Protocol (BCSP) runs over this link, carrying such things as HCI commands, events and data.

BCSP provides a set of reliable and unreliable bi-directional datagram streams.

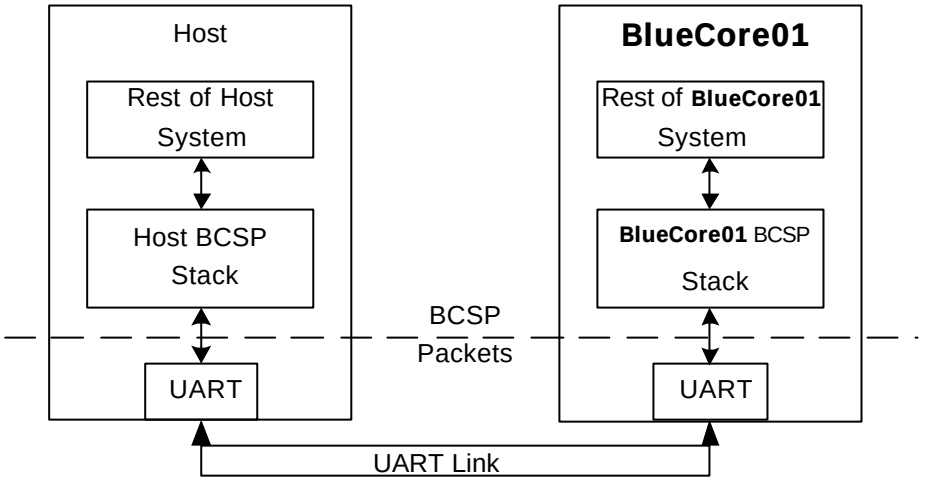
CSR has written a portable implementation of BCSP in `C` to help users interface to the chip. This document describes the BCSP implementation, and provides porting guidance.

Note: Although BCSP was written specifically for the **BlueCore01** Bluetooth chip, BCSP itself is not specific to Bluetooth. Thus it deliberately avoids direct reference to the Bluetooth specification. Similarly, this BCSP implementation deals with Bluetooth terms only in passing.

Context

A BCSP stack provides reliable and unreliable bi-directional datagram channels between the **BlueCore01**[TM] Bluetooth chip and its host.

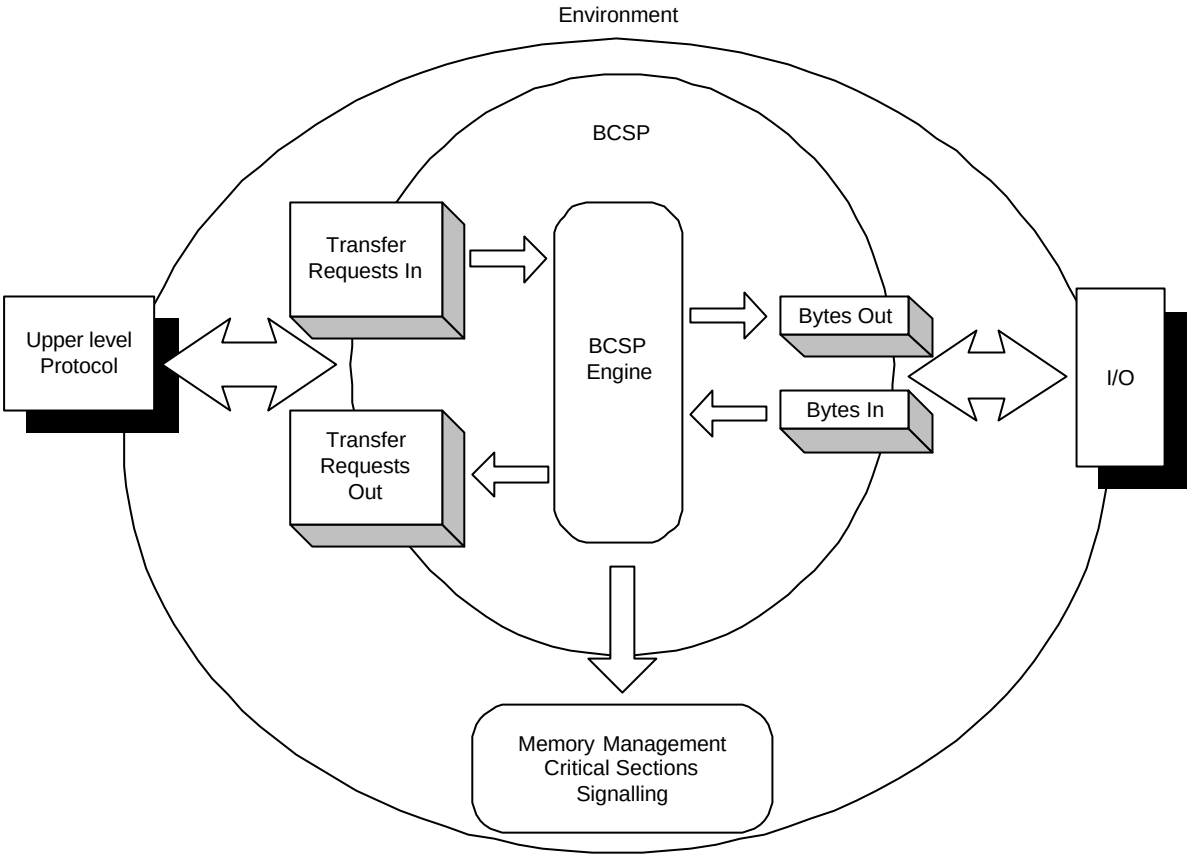
This document describes an implementation of the “Host BCSP Stack” shown in the diagram below.



Stack Organisaton

The stack consists of a core generic engine, which has four I/O points; two byte-oriented buffers and two transfer-request queues. Internally, the stack contains a number of co-operative tasks which share a single stack space and which are managed by a simple scheduler.

By itself, the code in the generic core cannot perform any ‘real’ I/O, so it must be wrapped in an ‘environment’ that provides it with an execution context and helper functions. The following diagram shows the relationship between the environment and the core BCSP stack.



The environment is expected to run the stack within a thread and to fill and empty the stack's I/O buffers as required. The environment may also supply memory-management, critical-section, and signalling routines.

Pseudo-code for the stack thread is shown below:

```
forever {
```

- Call the stack's scheduler – the stack will run until all internal tasks are blocked.
- Transfer bytes from stack's output buffer to UART transmit buffer.
- Transfer bytes from UART receive buffer to stack's input buffer.
- If all stack tasks are still blocked, wait for any of the following:

  1. stack's wakeup-time to arrive
  2. more bytes to arrive from UART
  3. some free space in the UART transmit buffer.
  4. A signal from the stack that a transfer-request has been queued

```
}
```

The stack must be correctly set up before the thread is run:

1. Memory must be allocated for the stack.
2. initialiseStack must be called.
3. A number of packets must be created and added to the stack.
4. An environment must be created and set.
5. A configuration may be created and set.

The Environment

A BCSPEnvironment needs to be created. This is basically a list of function pointers that the stack calls for environment functionality. All functions take a void * pointer as their first parameter and at runtime, the envState field is passed as this parameter. This enables the environment to maintain extra state which it might need.

The functions which should be supplied are:

```
void * (*allocMem)(void * envState,uint32 size);
```

```
void (*freeMem)(void * envState,void*);
```

These two functions are used to allocate and free memory.

```
void (*enterCS)(void * envState) ;
```

```
void (*leaveCS)(void * envState) ;
```

These two functions are used to enter and leave a critical section. This is required to safely manage the transfer-request queues.

```
void (*signal)(void * envState) ;
```

The signal function is called whenever a new transfer-request is added to a queue by the user. Typically this function is used to wake-up the stack-thread.

```
void (*onLinkFailure)(void * envState);
```

This function is called if the maximum number of transmission retries has been reached. Note that after this function is called, the retry-count is reset to zero so the function will continue to be called if the stack-thread is allowed to continue running.

```
void (*releasePacket)(void * envState,Packet * pkt);
```

This function is called by the stack when it is shutdown to free the packet headers that were added by the environment.

```
uint32 clockMask ;
```

This variable should contain a bit set for each significant bit in a time. Eg, if a 16-bit timer is used, this variable should be set to `0xffff`; if a 32 bit timer is used, clockMask should be `0xffffffff`.

```
void * envState ;
```

This pointer will typically be set to point to some state maintained by the environment. All of the above functions are passed this pointer.

Types And Definitions

The code makes widespread use of types such as `uint32, uint18, uint8`. These should be defined in an env/YOURENV/envdefs.h file. See the example in env/templates for further details.

Portability Issues

The code relies quite heavily on the following property of structure-packing:

Given:

```
struct _base
{
  …
} ;
```

and:

```
struct _derived
{
 struct _base baseObj ;
 …
} derivedObj;
```

then assume:

&derivedObj.baseObj == &derivedObj

It is possible (although unlikely) that your compiler may not exhibit this behaviour, in which case, the queue-based structures will need to be rewritten to take explicit pointers to their derived objects.

Improvements

The stack could be improved by adding some or all of the following features:

- At present, no out-of-memory checking is performed when attempting to allocate data for packets; the stack is not expected to recover from an out-of-memory condition.
- Packet data sections could be segmented to remove the need for data copying.
- Data could be written directly into the transfer-requests buffer when receiving a packet if the transfer request has already been queued.

API Reference

Note: This is a subset of the complete BCSP API; these functions are those which would normally be used by those writing an environment wrapper.

```
void initialiseStack(BCSPStack * stack);
```

This function must be called before the stack is run. It sets up the stack state. This function must be called before any other intialisation.

```
void BCSPaddPacket(BCSPStack * theStack,Packet * pkt) ;
```

This function is used to add a packet header to the stack. When the stack is shutdown, the packets will be released through the evironment's releasePacket function.

```
BCSPEnvironment * geDefaultEnvironment();

BCSPEnvironment * getEnvironment(BCSPStack * stack);

void setEnvironment(BCSPStack * stack,BCSPEnvironment * env) ;
```

These functions can be used to get and set the stack's environment structure . The structure is passed in via a pointer so it must be persistent.

```
StackConfiguration * getDefaultStackConfiguration() ;

StackConfiguration * getStackConfiguration(BCSPStack * stack) ;

void setStackConfiguration(BCSPStack*stack,StackConfiguration*cfg);
```

These functions can be used to get and set the stacks configuration structure . The structure is passed in via a pointer so it must be persistent.

```
uint32 scheduler(BCSPStack * theStack,uint32 timeNow);
```

This function calls the BCSP engine. The engine will run until all internal tasks are blocked. This typically happens because either the transmit buffer is full, the receive buffer is empty, a packet has been received for which no transfer request has been queued, or the stack has nothing left to do.

Upon completion of this function , the environment should attempt to empty the transmit buffer and fill the receive buffer. If either of these operations is even partially successful, the stack may have become unblocked and can be run again. The function returns the time (in ms) that it should be woken up in the absence of any interim I/O events.

```
uint16 numBytesInTransmitBuffer(BCSPStack * stack);

uint16 numFreeSlotsInReceiveBuffer(BCSPStack * stack);

uint8 readByteFromTransmitBuffer(BCSPStack * stack);

void writeByteToReceiveBuffer(BCSPStack * stack,uint8 data);

void readFromTransmitBuffer(BCSPStack * stack,
    uint8* dest,uint8 len);

void writeToReceiveBuffer(BCSPStack * stack,uint8 *src,
    uint8 len);
```

These functions are used to read bytes from the stack's transmit buffer and write bytes into the stack's receive buffer. numBytes/FreeSlots *must* be called before attempting to read/write bytes. If an attempt is made to write more bytes than there are free slots in the receive buffer or to read more bytes than are currently in the transmit buffer, the behaviour of the stack is undefined. The read/writeByte functions transfer a single byte whilst the readFrom/writeTo functions write a block.

```
bool isStackIdle(BCSPStack * stack);
```

Returns true if the stack is blocked and false if not. It is important to call this function after running the scheduler and conducting I/O since the I/O may have unblocked the stack.

```
bool BCSPLinkEstablished(BCSPStack * stack) ;
```

Returns true if the link-establishment protocol has received a sync-resp packet.

```
void BCSPshutdownStack(BCSPStack * stack) ;
```

Causes the stack to shutdown and lose all state. All pending transfer-requests are completed with status transferCancelled, all packet-data is freed, and all packet-headers are returned to the environment via the environent releasePacket function. The environment must ensure that no further transfer requests are queued after this function is called and that the scheduler is not run again otherwise memory leaks may occur.

```
#define BTRACELOG(filename)

#define BTRACEENABLE(bits)

#define BTRACEDISABLE(bits)

#define BTRACEID(str)

#define BTRACEn(bit,str)

#define PLAINBTRACEn(bit,str)
```

These macros are used for debugging; by default, debug output is dumped to stdio but BTRACELOG can be used to open a file which will also log the debug output. BTRACEID can be used to set a string which will be prepended to all output – this can be useful when debugging multiple stacks, BTRACEENABLE/DISABLE are used to turn tracing on and off for areas of interest; by default all tracing is enabled. See BTRACE.h for bit-definitions. BTRACEn is used to actually print a trace – this takes a bit-definition and a printf style argument list. n should be replaced by the number of printf arguments, not including the formatting string. Eg

```
BTRACE3((1<<15, “A debug trace with arguments %d
  %d %d\n”,1,2,3) ;
```

This will only produce output if bit 15 is set in the debug bitmask.

The PLAINTRACE macro provides similar functionality except that the TRACEID string will not be prepended to the string.

Definitions and Acronyms

Bluetooth	A set of technologies providing short range audio and data transfer over radio connections.
bc01	**BlueCore01**™, CSR's Bluetooth chip
BCSP	BlueCore Serial Protocol
CSR	Cambridge Silicon Radio Ltd
HCI	Host Controller Interface; part of the Bluetooth Specification

Record of Changes

Date:	Revision:	Reason for Change:
04 MAY 00	a	First draft for comment (CSR reference: bc01-m-022)
10 MAY 00	b	Release for **BlueCore01**™ evaluation system

Porting the BCSP Stack

AN002

May 2000