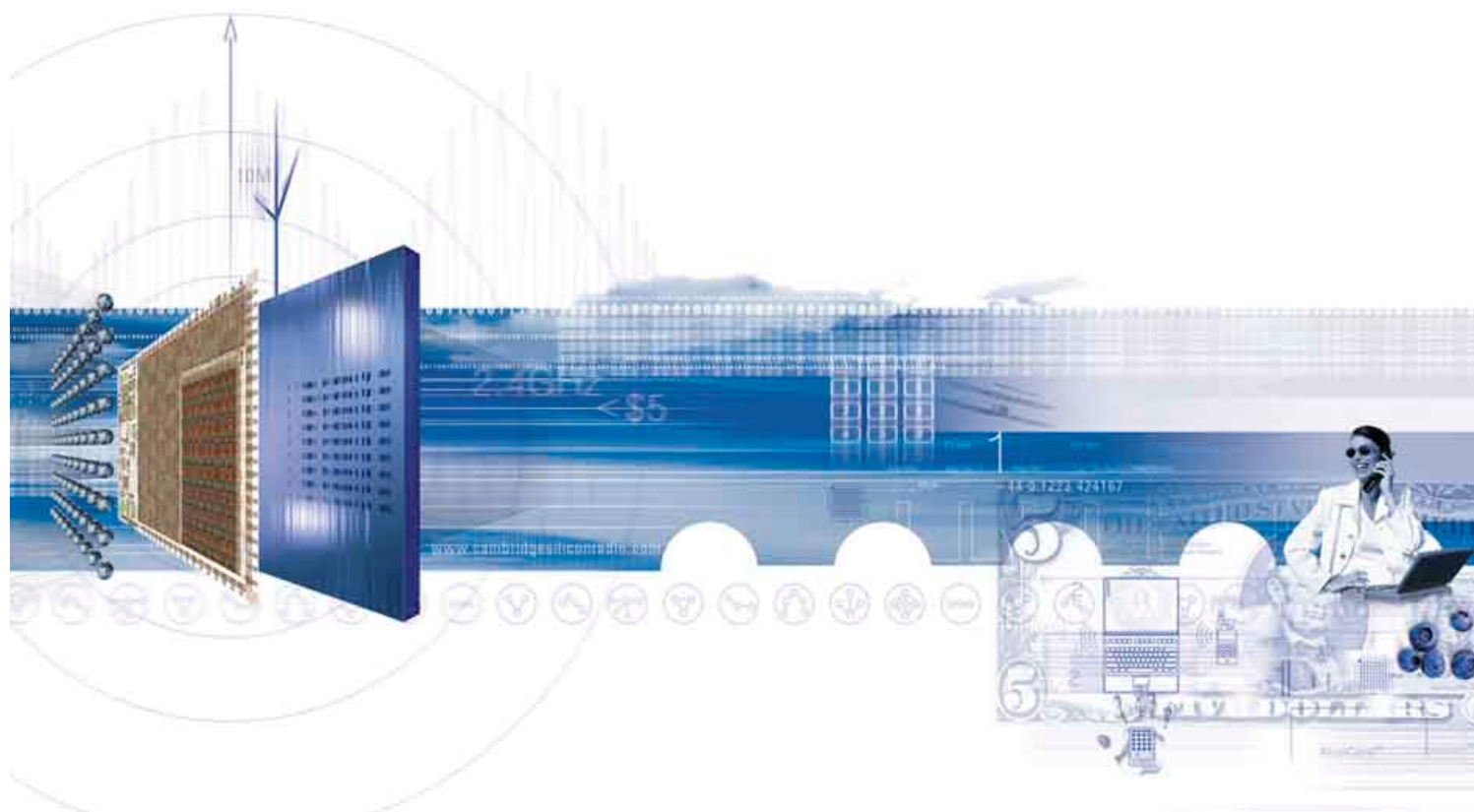




**BlueCore™**

# YABCSP Overview

January 2003



**CSR**

Unit 400 Cambridge Science Park  
Milton Road  
Cambridge  
CB4 0WH  
United Kingdom

Registered in England 3665875

Tel: +44 (0)1223 692000

Fax: +44 (0)1223 692001

[www.csr.com](http://www.csr.com)

## Contents

<b>1</b>	<b>Introduction.....</b>	<b>3</b>
<b>2</b>	<b>Conditions of Use.....</b>	<b>4</b>
<b>3</b>	<b>Difference between ABCSP and YABCSP .....</b>	<b>5</b>
3.1	Changes in the Interface due to Multiple Instances Support .....	5
3.1.1	Functions .....	5
3.1.2	Macros .....	5
3.2	Other Changes in the Interface .....	6
<b>4</b>	<b>Example .....</b>	<b>7</b>
4.1	Implementation Details for the Support Functions .....	7
4.1.1	Message Structure .....	7
4.1.2	Events.....	7
4.1.3	Deliver .....	7
4.1.4	Timers .....	8
4.2	Example Overview .....	8
4.3	MicroSched Overview .....	8
<b>5</b>	<b>Document References .....</b>	<b>9</b>
	<b>Acronyms and Definitions .....</b>	<b>10</b>
	<b>Record of Changes .....</b>	<b>11</b>

# 1 Introduction

BCSP (BlueCore Serial Protocol) is a proprietary UART (Universal Asynchronous Receiver Transmitter) protocol used on CSR's **BlueCore™** Bluetooth™ chips. It can be considered an alternative to the two UART host transports defined in the Bluetooth 1.1 Specification [BT11].

CSR publishes the source code of an implementation of a BCSP host stack used in all of CSR's host programs: demos, configuration tools and test tools. This stack has served this role well, but some BlueCore users have commented that it consumes too much RAM (Random Access Memory) for use in small, embedded applications. Hence, CSR provides other BCSP implementations optimized for different applications:  $\mu$ BCSP (micro BCSP), ABCSP (Another BCSP) and the version described in this document, YABCSP (Yet Another BCSP).

$\mu$ BCSP is optimised to limit the ROM (Read Only Memory) and to some extent RAM footprints. The penalty for the reduced size is that the throughput is limited, mainly by a sliding window size of 1.  $\mu$ BCSP is good for embedded applications where ROM usage is more important than throughput.  $\mu$ BCSP has a very simple interface and is easy to port.

ABCSP is optimized to limit the RAM usage and can work on small memory pool buffers. The penalty for this is that it uses more processing power. ABCSP is good for embedded applications where RAM usage is important and will, if sufficient processing power is available, provide good throughput. ABCSP requires complex integration with its host environment.

YABCSP is optimised to limit the processing power requirements. The penalty for the optimisation is that it uses more and larger chunks of RAM than ABCSP. YABCSP supports multiple instances. YABCSP was written with almost the same interface as ABCSP. Hence, switching to/from ABCSP to YABCSP is extremely easy. This document only contains the differences between ABCSP and YABCSP. Please, refer to the ABCSP Overview, [AN111], for detailed explanation of the interface.

For more information about BCSP, please refer to the BCSP documentation on our website [www.csrsupport.com](http://www.csrsupport.com).

## 2 Conditions of Use

The YABCSP stack is provided as C source code and may be freely used for BlueCore chip applications. It is expected that users will change the code for their own applications. CSR provides no formal support for the code. There is no intention to extend the code with a set of platform-specific `#define` porting options. However, CSR appreciates bug reports and suggestions for code improvements.

The code has been tested with CSR's Casira™ hardware. It has also been used in embedded applications by CSR's customers. The following standard statement of quality and fitness for purpose applies:

**Note:**

Use of the software is at your own risk. This software is provided "as is," and CSR cautions users to determine its suitability for themselves. CSR makes no warranty or representation whatsoever of merchantability or fitness of the product for any particular purpose or use. In no event shall CSR be liable for any consequential, incidental or special damages whatsoever arising out of the use of or inability to use this software, even if the user has advised CSR of the possibility of such damages.

## 3 Difference between ABCSP and YABCSP

This document only contains the differences between ABCSP and YABCSP. Please, refer to the ABCSP Overview [AN111], for detailed explanation of the interface.

The number of files is reduced in YABCSP compared to ABCSP. Hence, the directory structure used by ABCSP (`config`, `include`, `src` and `make`) is now a flat directory structure of only one directory, `src` contains all YABCSP files. The files that the user shall provide are called `config_x.h`. In addition, the file, `chw.h`, must be updated if the machine type is not known. The example is located in the example directory.

### 3.1 Changes in the Interface due to Multiple Instances Support

YABCSP supports multiple instances. A pointer to a specific instance is given in the API to/from the YABCSP library.

The following prototypes are only changed due to the multiple instances support feature:

#### 3.1.1 Functions

The following function prototypes are different from ABCSP:

- `void abcspp_init(void);` has become `void abcspp_init(abcspp *_this);`
- `unsigned abcspp_sendmsg(ABCSP_TXMSG *msg, unsigned chan, unsigned rel);`  
has become  
`unsigned abcspp_sendmsg(abcspp *_this, ABCSP_TXMSG *msg, unsigned chan, unsigned rel);`
- `unsigned abcspp_pumptxmsgs(void);` has become  
`unsigned abcspp_pumptxmsgs(abcspp *_this);`
- `unsigned abcspp_uart_deliverbytes(char *buf, unsigned n);` has become  
`unsigned abcspp_uart_deliverbytes(abcspp *_this, char *buf, unsigned n);`
- `void abcspp_bcspp_timed_event(void);` has become  
`void abcspp_bcspp_timed_event(abcspp *_this);`
- `void abcspp_tshy_timed_event(void);` has become  
`void abcspp_tshy_timed_event(abcspp *_this);`
- `void abcspp_tconf_timed_event(void);` has become  
`void abcspp_tconf_timed_event(abcspp *_this);`

#### 3.1.2 Macros

The following macro prototypes are different from ABCSP:

- `void ABCSP_EVENT(unsigned e);` has become  
`void ABCSP_EVENT(abcspp *_this, unsigned e);`
- `void ABCSP_REQ_PUMPTXMSGs(void);` has become  
`void ABCSP_REQ_PUMPTXMSGs(abcspp *_this);`
- `void ABCSP_PANIC(unsigned e);` has become  
`void ABCSP_PANIC(abcspp *_this, unsigned e);`
- `void ABCSP_DELIVERMSG(ABCSP_RXMSG *msg, unsigned chan, unsigned rel);`  
has become

```
void ABCSP_DELIVERMSG(abcsp *_this, ABCSP_RXMSG *msg, unsigned chan,
unsigned rel);
```

- ABCSP\_RXMSG \*ABCSP\_RXMSG\_CREATE(unsigned len); has become  
ABCSP\_RXMSG \*ABCSP\_RXMSG\_CREATE(abcsp \*\_this, unsigned len);
- void ABCSP\_RXMSG\_COMPLETE(ABCSP\_RXMSG \*msg); has become  
void ABCSP\_RXMSG\_COMPLETE(abcsp \*\_this, ABCSP\_RXMSG \*msg);
- void ABCSP\_RXMSG\_DESTROY(ABCSP\_RXMSG \*m); has become  
void ABCSP\_RXMSG\_DESTROY(abcsp \*\_this, ABCSP\_RXMSG \*m);
- void ABCSP\_START\_BCSP\_TIMER(void); has become  
void ABCSP\_START\_BCSP\_TIMER(abcsp \*\_this);
- void ABCSP\_START\_TSHY\_TIMER(void); has become  
void ABCSP\_START\_TSHY\_TIMER(abcsp \*\_this);
- void ABCSP\_START\_TCONF\_TIMER(void); has become  
void ABCSP\_START\_TCONF\_TIMER(abcsp \*\_this);
- void ABCSP\_CANCEL\_BCSP\_TIMER(void); has become  
void ABCSP\_CANCEL\_BCSP\_TIMER(abcsp \*\_this);
- void ABCSP\_CANCEL\_TSHY\_TIMER(void); has become  
void ABCSP\_CANCEL\_TSHY\_TIMER(abcsp \*\_this);
- void ABCSP\_CANCEL\_TCONF\_TIMER(void); has become  
void ABCSP\_CANCEL\_TCONF\_TIMER(abcsp \*\_this);
- void ABCSP\_UART\_SENDBYTES(char \*buf, unsigned n); has become  
void ABCSP\_UART\_SENDBYTES(abcsp \*\_this, unsigned n);
- void ABCSP\_TXMSG\_DONE(ABCSP\_TXMSG \*msg); has become  
void ABCSP\_TXMSG\_DONE(abcsp \*\_this, ABCSP\_TXMSG \*msg);

### 3.2 Other Changes in the Interface

```
char *ABCSP_UART_GETTXBUF(unsigned *bufsiz); has become
size_t ABCSP_UART_GETTXBUF(abcsp *_this);
```

The YABCSP implementation works on an internal transmit buffer, which is a part of the instance data. The ABCSP\_UART\_GETTXBUF( ) macro is now used to obtain the maximum length of the output buffer. The size of internal buffer in the instance data is set according to the ABCSP\_MAX\_MSG\_LEN definition. The macro can be used to limit the size of the output. This can be useful if, for example, the YABCSP library is talking to an UART device which has a limited input buffer. If no such limitations come from the UART device, the macro should be set to return the size of the internal buffer as shown in the example (in the config\_txmsg.h file).

## 4 Example

An example is provided with the YABCSP source code. This example is for the Windows™ platform using Visual C++. A project file is provided for the example.

The example demonstrates how the YABCSP is integrated in a system. In this example the support functions are implemented using dynamic memory allocations.

The example is based on a tiny scheduler, called  $\mu$ Sched (micro scheduler). A single "task" is running in the scheduler. The example works on a serial port connected to a Casira with a HCI (Host Controller Interface) build running on it. It will issue `Read_BD_ADDR` commands whenever `Num_HCI_Command_Packets` are available. For every 100 commands it will write a dot "." on the screen. And for every 100 command complete events it will write an asterisk "\*". The program can be terminated by hitting the escape key (ESC).

### 4.1 Implementation Details for the Support Functions

The example support functions can be found in the `abccsp_support_functions.h` and `abccsp_support_functions.c` files. Most macro definitions in the `config_x.h` files are defined to call functions. All these functions can be found in the `abccsp_support_functions.c` file.

#### 4.1.1 Message Structure

Messages in the example are wrapped in a structure, `MessageBuffer`, which looks like this:

```
typedef struct
{
    unsigned length;
    unsigned index;
    unsigned char * buffer;
} MessageBuffer;
```

When a request for a message to be created (`abccsp_rxmsg_create`) is received, memory is allocated to hold the structure. The buffer in the `MessageBuffer` is also allocated according to the requested length. The length member holds the length of the buffer. The index member holds the current position the library is working on.

When the message is destroyed (`abccsp_rxmsg_destroy`) both the message structure and the buffer member is freed.

#### 4.1.2 Events

The only event (`ABCSP_EVENT`) that should not be ignored is `ABCSP_EVT_LE_SYNC_LOST`. This event indicates that the peer ABCSP side has reset. The example handles this by exiting.

The pump request (`ABCSP_REQ_PUMPTXMSG`) generates a "background interrupt" that the scheduler will pick up and process in due time by calling the `abccsp_pumptxmsg` function. In addition, the background interrupt function calls the `pumpInternalMessage` function, which handles the ABCSP choke.

#### 4.1.3 Deliver

The deliver (`ABCSP_DELIVERMSG`) function in the example decodes the HCI event and maintains a variable, `NumberOfHciCommands`, which holds the `Num_HCI_Command_Packets` parameter. In case the `NumberOfHciCommands` is not zero the scheduler task is scheduled to run. When the scheduler task is called it will issue a new read Bluetooth address command.

When the deliver function is called, ownership of the message buffer is passed on to the function. Hence, the deliver function frees the message buffer before returning from the function.

#### 4.1.4 Timers

If a start timer request is issued and the specific timer is running the support functions will stop the running timer before starting a new timer.

## 4.2 Example Overview

The example executes as 3 threads: The main thread (main program, which runs the scheduler), a RX UART thread and a TX UART thread.

The main thread which runs the scheduler also contains a keyboard handler, which runs on a scheduler timer event. The keyboard handler timer event is started in the main program immediately before the scheduler function is called. The main function, the keyboard handle and the task function are located in the `main.c` module.

The RX UART thread signals data are available to the main thread by using background interrupt #1. This is only done when a BCSP delimiter character (0xC0) is received. The scheduler then calls the `UartDrv_Rx` function when a background interrupt #1 is received. The `UartDrv_Rx` function calls the `abcsp_uart_deliverbytes` function to pass the received data on to the YABCSP library.

The TX UART thread writes data to the UART when data is available. If there is no data to transmit the thread sleeps until the main thread passes data on to it and signals it to wake up.

The UART driver is located in the `SerialCom.c` module.

## 4.3 MicroSched Overview

The scheduler is able of scheduling one task only. The task can be scheduled to run by calling the function `ScheduleTaskToRun()`. At start-up the task can have an initialisation function, which, if present, will be called before the main loop of the scheduler starts running. A task is registered in the scheduler by the `InitMicroSched()` function, which must be called before the `MicroSched()` function.

Background interrupt handlers are registered in the scheduler by the `register_bg_int()` function. Currently, only 2 background handlers can be defined.

The micro scheduler (found in the `uSched.c` file) is simple. It has the following basic structure:

```
void MicroSched(void)
{
    while (not terminated)
    {
        service "background interrupts" if any;
        run task if scheduled;
        service timer events;
        sleep until next timer event or until woken by external event;
    }
}
```

The scheduler can be woken by the UART threads, when they generate background interrupts.

Timers are functions to be called after a specific time. The `StartTimer` function returns a "**handle**" to the started timer. This "**handle**" can be used to stop a running timer before expiry.



## 5 Document References

Reference:	Document:
[AN111]	ABCSP Overview; CSR document, November 2001
[BT11]	Specification of the Bluetooth System, Volume 1, Core v1.1, 22 February 2001

## Acronyms and Definitions

ABCSP	Another <b>BlueCore</b> Serial Protocol
BCSP	<b>BlueCore</b> Serial Protocol
<b>BlueCore</b> ™	CSR's family of Bluetooth chips
Bluetooth™	A set of technologies providing audio and data transfer over short range radio connections
CSR	Cambridge Silicon Radio
HCI	Host Controller Interface
RAM	Random Access Memory
ROM	Read Only Memory
µBCSP	Micro BCSP
YABCSP	Yet Another <b>BlueCore</b> Serial Protocol

## Record of Changes

Date:	Revision	Reason for Change:
20 JAN03	a	Original publication of this document (CSR reference: bcore-an-012Pa).

# BlueCore™

## YABCSP Overview

### bcore-an-012Pa

### January 2003

Bluetooth™ and the Bluetooth logos are trademarks owned by Bluetooth SIG Inc, USA and licensed to CSR.

**BlueCore™** is a trademark of CSR.

All other product, service and company names are trademarks, registered trademarks or service marks of their respective owners.

CSR's products are not authorised for use in life-support or safety-critical applications.