ABCSP – Another BCSP Stack

bc01-me-038e

18th June 2002

1 INTRODUCTION

BCSP is a proprietary UART protocol used on CSR's Bluetooth chips. It can be considered an alternative to the two UART host transports defined in the Bluetooth 1.1 Specification.

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 bc01 users have commented that it consumes too much RAM for use in small embedded applications. In particular, the stack has its own scheduler based on `longjmp()`, and this tends to consume a significant amount of the host's main C runtime stack.

This document describes an alternative BCSP stack written for embedded Bluetooth hosts with limited supplies of RAM. Where the original BCSP stack promotes portability and configurability, abcsp ("another BCSP" stack) requires complex integration with its host environment, and is biased to minimise its consumption of the host's resources.

The abcsp BCSP stack implements BCSP itself, described in document AN004 (aka bc01-s-006). It also implements the BCSP Link Establishment protocol, described in document AN005 (aka bc01-s-010). Document AN003 (aka bc01-s-020) describes the allocation of BCSP's channels.

The original BCSP stack is described in documents AN001 (aka bc01-m-21) and AN002 (aka bc01-m-22).

2 CONDITIONS OF USE

The abcsp 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.

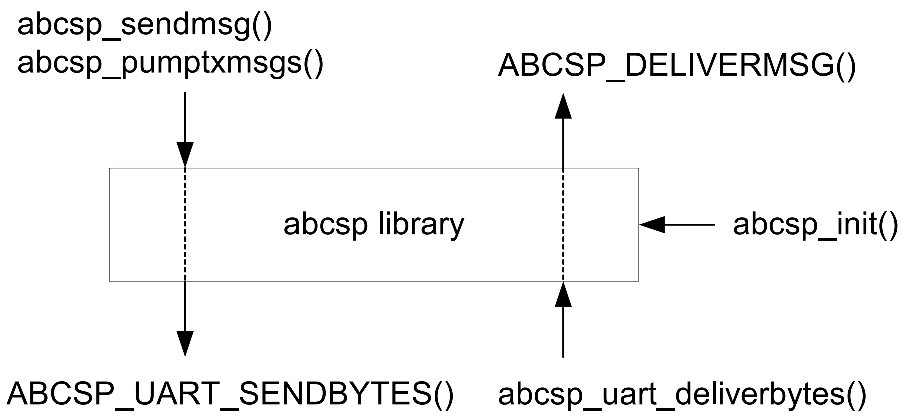
CSR provides no formal support for the code. There is no intention to extend the code with a set of platform-specific #define porting options. However, CSR appreciates bug reports and suggestions for code improvements.

The code has been tested with Casira hardware. It has also been used in an embedded application by several of CSR's customers. However, the code has not been used as heavily or extensively as the first CSR BCSP stack, and so its quality must be correspondingly suspect. As such, the following standard statement of quality and fitness for purpose applies:

> 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 BASIC STRUCTURE

This section gives an overview of the abcsp stack. The figure below shows the stack's main external interfaces:



The abcsp code's primary task is to translate between higher layer format messages (`HCI ACL`, `HCI SCO`, `HCI CMD/EVT`, etc.) and the corresponding `BCSP` wire (`UART`) format messages.

The abcsp library must first be initialised by a call to `abcsp_init()`.

Transmit path:

> To send a message, higher layer code calls `abcsp_sendmsg()`; this places the message into a queue within the library. The higher layer code then repeatedly calls `abcsp_pumptxmsgs()` to translate the message into its BCSP wire format and push these bytes out of the bottom of the library via `ABCSP_UART_SENDBYTES()`.

Receive path:

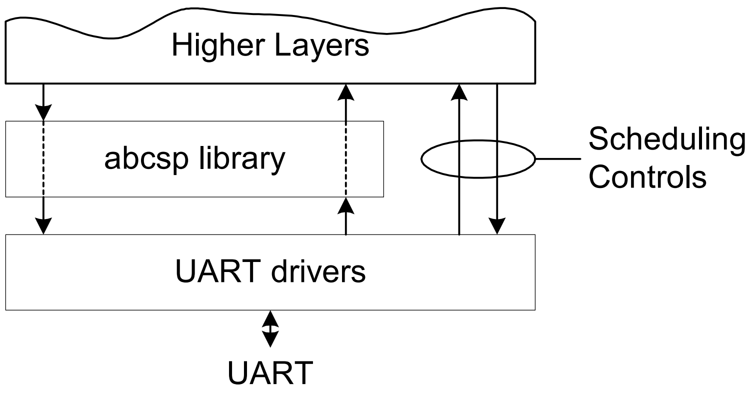
> For inbound messages the `UART` driver code passes `BCSP` wire format bytes into the library via calls to `abcsp_uart_deliverbytes()`. When the library has all of the bytes to form a complete higher layer message it calls `ABCSP_DELIVERMSG()` to pass this to higher layer code.

Except for the library's initialisation call, `abcsp_init()`, the code blocks implementing the transmit and receive paths are largely independent.

In the diagram, function names in lower case are part of the library code. Function names in upper case are macros within the code - the external environment must implement these according to definitions given in abcsp source header files.

The library contains no internal scheduler; it depends on the function calls described above to drive the code. The transmit path is driven ("down") by calls to `abcsp_sendmsg()` and abcsp_pumptxmsgs(); these result in calls to ABCSP_UART_SENDBYTES(). Similarly, the receive path is driven ("up") by calls to abcsp_uart_deliverbytes(), resulting in calls to ABCSP_UART_SENDBYTES().

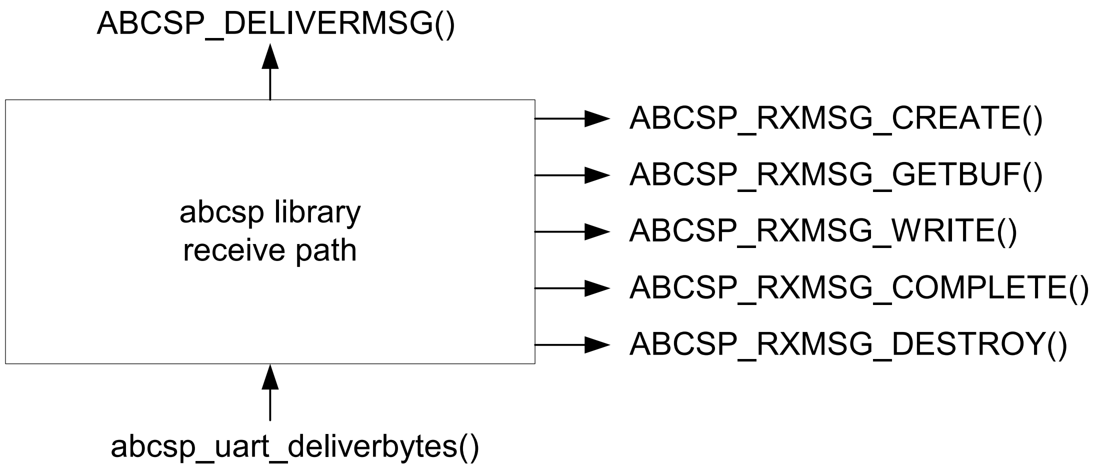
A key point about scheduling: the paths' two output functions, ABCSP_UART_SENDBYTES() and ABCSP_DELIVERMSG(), have no means of detecting failure to deliver their payloads. Consequently the surrounding code must not call abcsp_sendmsg(), abcsp_pumptxmsgs() or abcsp_uart_deliverbytes(), unless it knows the corresponding output function can accept data. A fuller view of the abcsp library's environment is thus:



An associated point: a call to one of the abcsp library's input functions makes, at most, one call to the corresponding output function, thus the surrounding code only needs to be sure that the output function can accept one payload.

4 RECEIVE PATH

The previous section above gave a rather simple description of the library's receive path. The diagram below gives more detail, but this still isn't the full story:



The UART driver code calls:

```
        unsigned abcsp_uart_deliverbytes(char *buf, unsigned n);
```

This passes the "n" bytes in the buffer "buf" into the abcsp library and returns the number of bytes consumed from the buffer. The library attempts to build a higher layer message from the bytes. It may do this in just one call to abcsp_uart_deliverbytes(), or it may take several calls. However, eventually a call is (normally) made to:

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

This delivers the message "msg" on BCSP channel "chan" to the higher layers of code. If "rel" is 0 then the message was received on an unreliable BCSP channel, else it was received on a reliable channel.

Key point: ABCSP_DELIVERMSG() has no means of refusing the message. Thus the UART driver code must not call abcsp_uart_deliverbytes() unless it is sure that ABCSP_DELIVERMSG() can accept a message. However, the abcsp library guarantees to make at most one call to ABCSP_DELIVERMSG() for each call to abcsp_uart_deliverbytes(). This gives external code fairly fine control over the library's use of resources. (A similar pattern is used on the transmit path.)

As noted in the introduction, functions named in lower case are provided by the abcsp library, however, upper case names must be provided by the external code. Thus abcsp_uart_deliverbytes() is part of the abcsp code, but ABCSP_DELIVERMSG() is #defined as a macro in an abcsp code header file, and its implementation must be provided by code external to the abcsp library itself.

The two functions also illustrate the abcsp code's convention of using basic C types wherever possible: "unsigned" rather than more informative "uint16" and "bool" types. This convention is used on the code's external interfaces to attempt to improve the code's portability.

The patent exception is the type ABCSP_RXMSG*. This is really just a #define for a void*. The abcsp library code deals with higher layer message *references*, allowing external code to manage these messages' (buffer) structure. This approach requires the five other function calls shown in the diagram:

- ABCSP_RXMSG_CREATE() creates a new higher layer message, returning the corresponding ABCSP_RXMSG* reference.

- ABCSP_RXMSG_GETBUF() takes the message reference as an argument, and asks the external code for access to a buffer in the message. The external code chooses the size of the buffer, i.e., knowledge of how the message's bulk storage is structured is delegated to the external code. This approach allows the use of lots of modest buffers to form a single large BCSP message – an approach that significantly aids embedded systems where RAM is in short supply. (Experience (on bc01b!) shows that using lots of small blocks of RAM makes much better use of limited memory than requiring a few larger blocks.)

- ABCSP_RXMSG_WRITE() tells the external code how much of the buffer obtained by a preceding ABCSP_RXMSG_GETBUF() call has been consumed.
- ABCSP_RXMSG_COMPLETE() tells the external code that the message is complete, and thus that no more calls will be made to ABCSP_RXMSG_GETBUF() or ABCSP_RXMSG_WRITE() for that message.
- ABCSP_RXMSG_DESTROY() destroys a message. This is invoked if the abcsp library encounters in irrecoverable error when constructing the message, e.g., a CRC failure.

The abcsp code's normal usage pattern is thus something like:

> One or more calls to abcsp_uart_deliverbytes() provoke:
>
> > One call to ABCSP_RXMSG_CREATE() to create a fresh higher layer message.
> >
> > While the higher layer message is incomplete:
> >
> > > Call ABCSP_RXMSG_GETBUF() to obtain a byte buffer.
> > >
> > > Write to the byte buffer.
> > >
> > > Call BCSP_RXMSG_WRITE() to inform external code how much of the buffer has been written to.
> >
> > One call to ABCSP_RXMSG_COMPLETE() to mark the higher layers message as complete.
> >
> > One call to ABCSP_DELIVERMSG() to pass the message (reference) to higher layer code.

At any time, the receive path uses these (external/macro) functions to construct at most one message.

Not all BCSP messages result in calls to these functions – notably BCSP Link Establishment and "Ack" messages.

5 TRANSMISSION PATH

Higher layer code passes a message into the stack using the function:

```
        unsigned abcsp_sendmsg(ABCSP_TXMSG *msg, unsigned chan,
                               unsigned rel);
```

This pushes the message "msg" into the top of the stack, requiring it to be transmitted on BCSP channel "chan" (acceptable values 2 to 15). If "rel" is zero then it is passed on the unreliable channel "chan", else it is passed on the reliable channel "chan". The function returns 1 if message has been accepted, else it returns 0.

In a manner similar to the Receive Path, the type ABCSP_TXMSG* is really just a void*, and is treated as an opaque message reference by the abcsp code.

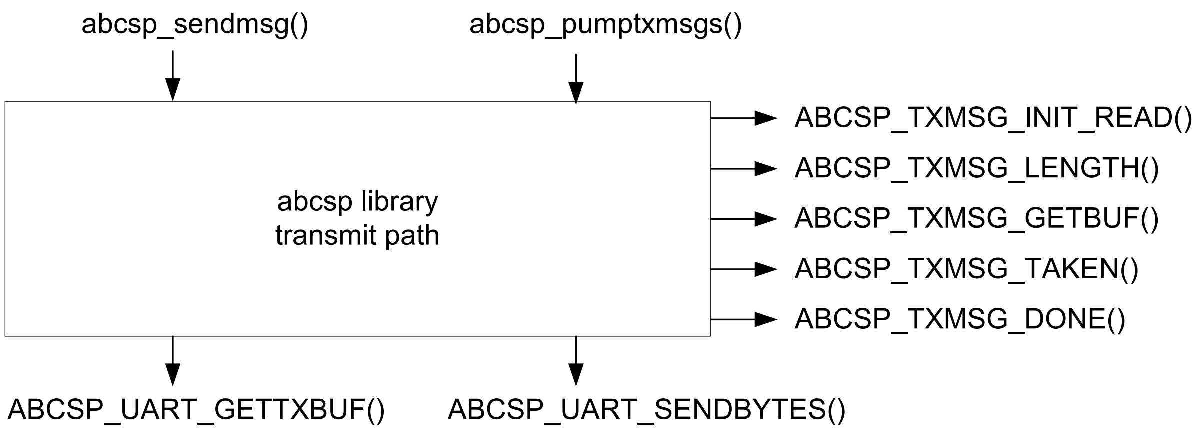
The call to `abcsp_sendmsg()` places a message (reference) into a queue with the abcsp library, but it requires one or more calls to:

```
        unsigned abcsp_pumptxmsgs(void);
```

to convert the message into its wire format, eventually making a call to:

```
        void ABCSP_UART_SENDBYTES(char *buf, unsigned n);
```

to pass a byte buffer to the UART. The following diagram gives a fuller picture of the transmit path:



The transmit path doesn't contain an internal buffer in which to build the `UART` byte stream. Instead it asks the external `UART` driver code to provide a buffer with the `ABCSP_UART_GETTXBUF()` call. It then writes to the buffer, then finally it returns the same buffer to the `UART` driver via a call to `ABCSP_UART_SENDBYTES()`. This approach gives external code fairly fine control over the library's use of transmit `UART` buffer memory.

The five transmit message handling functions are similar in style to those used by the receive path. However, these are used to access a message already prepared by higher layer code (and passed into the abcsp library via `abcsp_sendmsg()`), whereas the receive path's functions are used to *create* a higher layer message.

- `ABCSP_TXMSG_INIT_READ()` takes a message reference parameter from a `abcsp_sendmsg()` call and tells external code that it is about to start to reading from it.
- `ABCSP_TXMSG_LENGTH()` asks how many payload bytes are in a message.
- `ABCSP_TXMSG_GETBUF()` obtains access information for a raw byte buffer in a transmit message. The external code provides the address of a buffer and tells the abcsp library how long the buffer is. This mirrors `ABCSP_RXMSG_GETBUF()`, allowing the external code to hold a single large `BCSP` message payload as a sequence of smaller message

fragments.

- `ABCSP_TXMSG_TAKEN()` tells external code how many bytes have been consumed from the buffer obtained by the preceding call to `ABCSP_TXMSG_GETBUF()`.

- `ABCSP_TXMSG_DONE()` tells external code that it has finished all work with a message. For an unreliable `BCSP` message (`SCO`) this means that the message has been sent. For a reliable `BCSP` messages this means that the message has been sent to the peer and that the peer has returned acknowledgement of its reception. This can be used as the basis of a "recorded delivery" mechanism.

This last point hints at why the transmit path is significantly more complex than the receive path. The `abcsp_sendmsg()` function places a (reliable) message into a queue then makes calls to `abcsp_pumptxmsgs()` to translate *messages* (plural!) to wire format. Then, when the receive path tells the transmit path that reliable messages have been acknowledged, the messages have to be removed from the reliable message queue and corresponding calls made to `ABCSP_TXMSG_DONE()`.

For reliable messages, the queue behind `abcsp_sendmsg()` can grow to a fixed (`#define`d) maximum length that matches the `BCSP` transmit window size, thus the queue is used to hold all messages that are either in the `BCSP` transmit window (sent, but not acknowledged), or which are due to be added to the window. This means that a call to `abcsp_sendmsg()` may refuse to take a message, and so higher layer code must be able to handle a refusal.

For unreliable messages (`SCO`) the queue will hold only one message. If the queue is empty when an unreliable message is passed into `abcsp_sendmsg()` then it is placed in the queue. Unreliable messages have high priority so it is probable that the message will be transmitted next.

If an unreliable message is submitted to `abcsp_sendmsg()` when there is already a message in the queue then the new message discards and replaces the existing message. This reflects the unreliable channel's bias in favour of supporting `SCO` (voice), where data freshness is important.

The abcsp code's normal usage pattern of the message access functions is something like:

> One call to `abcsp_sendmsg()` to queue a (reliable) message.
>
> One or more calls to `abcsp_pumptxmsgs()` provoke:
>
> > One call to `ABCSP_TXMSG_INIT_READ()` to tell the message manager that abcsp is about to start reading a message. Typically this is used to set/rewind its read pointer to zero.
> >
> > Each call to `abcsp_pumptxmsgs()` provokes something like:
> >
> > > One call to `ABCSP_TXMSG_GETBUF()` to obtain a byte buffer.
> > >
> > > One call to `ABCSP_UART_GETTXBUF()` to obtain a `UART` byte

buffer.

Read from the higher layer message's byte buffer, translate to wire format and write to the UART byte buffer.

When the UART byte buffer is full, or when the source buffer has been drained, call ABCSP_RXMSG_TAKEN() to inform external code how much of the buffer has been consumed and call ABCSP_UART_SENDBYTES() to stoke the UART.

When the peer BCSP stack acknowledges reception of the reliable message, call ABCSP_RXMSG_DONE() to report that the abcsp library is no longer interested in the message.

This is a much simplified description of how the transmit path works; this is given simply to illustrate how the five ABCSP_TXMSG_*() functions are used.

6 EVENTS

Two function calls allow the abcsp library code to send alerts to the external code:

```
void ABCSP_REQ_PUMPTXMSGS(void);

void ABCSP_EVENT(unsigned e);
```

The first reports to the external code that there is work pending that requires a call to be made to abcsp_pumptxmsgs() when convenient. For example, if the receive path accepts a reliable BCSP message then it (sets an internal flag and) calls ABCSP_REQ_PUMPTXMSGS() to ask the external code to call abcsp_pumptxmsgs() to send acknowledgement of the message back to the peer.

The ABCSP_REQ_PUMPTXMSGS() call must not be wired directly to a call into abcsp_pumptxmsgs(). Rather, it is expected that the ABCSP_REQ_PUMPTXMSGS() will set a flag in external code, then the external code will use the flag to schedule a call to abcsp_pumptxmsgs() after the current abcsp function call has returned. If a direct wiring is made then the code will probably go re-entrant and bomb.

The second function, ABCSP_EVENT(), is scattered liberally throughout the abcsp code to report significant events to the external environment: achieving SLIP sync, achieving BCSP Link Establishment sync, etc. There is no need to act on any of these messages, indeed the macro can be #defined to be nothing and all of the alerts will drop harmlessly from the code. The only event that can be of operational importance to the external code is BCSP Link Establishment loss – this indicates that the peer BCSP stack has been restarted, for which the only reasonable response is to restart the local BCSP stack.

7 BCSP LINK ESTABLISHMENT

The receive path of the library includes an implementation of the BCSP Link Establishment entity. This prevents bulk BCSP traffic from flowing until it is sure that both sides of the BCSP link have initialised themselves. It also allows the local BCSP stack to determine that the peer BCSP stack has restarted.

This collects some messages from the peer on the receive path. It also requests the transmit path to send some messages to the peer (via internal signalling and calls to ABCSP_REQ_PUMPTXMSGS()).

The Link Establishment implementation is isolated to a single file, and it would be simple to remove it from the source if needed. This might be the case where it was decided that a system didn't need to use BCSP-LE at all (unwise) or where BCSP-LE was implemented in higher layer code (unlikely).

8 TIMED EVENTS

The abcsp library makes use of three timed events; two for the BCSP Link Establishment entity, and one to provoke retransmission of BCSP reliable messages.

For each timer the abcsp code has three functions. Here are the functions for the BCSP-LE Tshy timer:

```
void ABCSP_START_TSHY_TIMER(void);

void ABCSP_CANCEL_TSHY_TIMER(void);

void abcsp_tshy_timed_event(void);
```

The abcsp code calls ABCSP_START_TSHY_TIMER() to request it to start an external Tshy timer. If the timer expires it should call abcsp_tshy_timed_event(). Alternatively, if the abcsp code decides that the timed event should be prevented from occurring then it calls ABCSP_CANCEL_TSHY_TIMER().

There are similar sets of three calls for BCSP-LE Tconf and BCSP reliable packet retransmission.

The abcsp code itself isn't concerned by how the timed events are implemented; presumably these will be based on the external code's timed event support. Also the abcsp code doesn't specify the timed events' periods, so the library needs no concept of how time values are described in the external code. However, the timers' recommended periods are:

BCSP Link Establishment Tshy	1 second
BCSP Link Establishment Tconf	1 second
BCSP Reliable Message Retransmit	0.25 seconds

9 DYNAMIC MEMORY ALLOCATION

The abcsp library makes very limited use of pool/heap memory. At the time of writing this document pool/heap memory is only used to encapsulate message references in the transmit path's two message queues. The library does not allocate pool/heap memory for bulk message storage.

External code needs to provide implementations of the following function calls:

`void *ABCSP_MALLOC(unsigned n);`	Like `malloc(n).`
`void *ABCSP_ZMALLOC(unsigned n);`	Like `calloc(1, n).`
`void  ABCSP_FREE(void *p);`	Like `free(p).`

10 LIBRARY INITIALISATION

The library's `abcsp_init()` must be called before any other abcsp library function. This initialises the code's state machines and internal variables.

The `abcsp_init()` function can be called at any other time to reinitialise the library. In this case its operation is more complex, as it releases/destroys any heap/pool memory and message references it holds. It also cancels any pending timers.

Initialisation kicks the `BCSP` Link Establishment engine into life.

11 SCHEDULING

The bulk of the abcsp library is driven via functions:

```
abcsp_pumptxmsgs()
abcsp_uart_deliverbytes()
```

Extra functionality is invoked by calling:

```
abcsp_init()
abcsp_sendmsg()
abcsp_tshy_timed_event()
abcsp_tconf_timed_event()
abcsp_bcsp_timed_event()
```

At the time of writing this document the abcsp library code is not thread safe. It presumes that external code will be calling at most one of these functions at any instant.

However, it should be fairly simple to enhance the code so that the transmit and receive paths can run on separate threads. There would need to be guards for:

- `abcsp_init()` – initialises both transmit and receive path elements.
- `RAM` shared by the transmit and receive paths. This database lives in file `txrx.c`.
- `ABCSP_EVENT().` This macro is called by both sides of the abcsp library.
- `ABCSP_MALLOC()`, etc. Currently these are only called from the transmit path, but this may change.

12 BCSP SPECIFICATION VIOLATIONS

The abcsp library (deliberately!) violates three elements of the `BCSP` specifications:

- The `BCSP` reliable message transmit specification (Sequencing Layer state machine) allows peer failure to be detected by counting the number of times the local state machine retransmits a message. If this exceeds a threshold it presumes the peer has failed.

  This has never been much use – most implementations ignore the option. Also, the `BCSP` Link Establishment's ability to detect a peer restart is much more valuable.

  The abcsp library doesn't implement this feature.

- The `BCSP` specification requires the `MUX` Layer to provide an output describing when it last received a message from the peer. This is not implemented.

- The `BCSP` specification requires the `MUX` Layer to implement the "choke", which prevents most traffic flowing to and from the peer. In the abcsp library this is implemented higher in the `BCSP` stack than the specification requires. (It's implemented in abcsp_sendmsg().) This is of no real consequence, and will only change the stack's behaviour when the BCSP link fails.

DOCUMENT HISTORY

Version	Date	Description
c	17th August 2001	For publication with version 1 of abcsp sources.
d	18th February 2002	Removed reference to bc01 – abcsp works with bc2. Clarified priorities of BCSP and SCO traffic flows. Typos.
e	18th June 2002	Added Document History section. For publication with version 1.1 of abcsp sources.