Bitbus Extended Broadcast

(XBC)

A BEUG Recommendation

Description:	*BITBUS Extended Broadcast (XBC)*
Date:	*April 11, 2000*
Last Change:	*April 11, 2000*
Status:	*Proposal*
Authors:	*Volker Goller, mocom software GmbH & Co KG*
Copyright:	*All rights reserved. This document is intellectual property of BEUG - The BITBUS European User's Group This document can be used for documentation purposes by all BITBUS users and manufactures if their implementation and products meets this specification.*
Contact:	*e-mail: wg1@bitbus.org*

Contents

1. Extended Broadcast 4

1.1. What is it? 4

1.2. Where does it fit? 4

1.2.1. Motion control 4
1.2.2. Time triggered control systems 5

1.3. A few definitions 5

2. XBC Layer 2 7

2.1. XBC message formats 7

2.1.1. USER PDU Format 7
2.1.2. Layer2 XBC PDU Format 8
2.1.3. UA - Unnumbered acknowledge control message 8
2.1.4. UP - Unnumbered poll control message 9

2.2. Master protocol 10

2.3. Slave protocol 12

3. XBC Layer 7 17

3.1. Master 17

3.2. Slave 17

4. BAPI Functions for XBC 18

I. BitbusSetRouting 18

1. Extended Broadcast

1.1. What is it?

XBC is a broadcast message with optional response. An XBC is send by the master using the broadcast address FFh. The message contains the node address of a single node that is allowed to respond to this broadcast. If the slave address is FFh, the XBC is handled like a regular broadcast and no node will respond.

XBC services do NOT use retries in case of failure - like the more secure standard BITBUS services do. The basic rule is : lost information is old garbage - nobody need it anymore. So XBC is designed for fast, cyclic information exchange in the first place.

However, XBC is still BITBUS from the users point of view. The message format is only slightly different from standard BITBUS messages. The XBC broadcast is still a ORDER and the slave still sends REPLIES.

Best of all: XBC and standard BITBUS communication can be mixed as needed. Because XBC uses the same structures, similar approaches and does not lock the bus while a XBC is handled, things are easy to control.

XBC does not make any assumption about the time needed to handle a order by a slave or how frequent an XBC is send. You can use this service "from time to time" or as frequent as your system can handle it.

1.2. Where does it fit?

As a rule of thumb XBC fits where:

- Global data is provided by the master (e.g. machine or plant status info)
- Synchronised status information is needed (all slaves will receive at the same time)
- Cyclic data exchange with synchronisation is needed

1.2.1. Motion control

In motion control applications, XBC fits for many purposes:

- continues path run
- synchronised torque control
- synchronised speed control
- synchronised position control

Example:

The XBC will distribute timing information as well as trajectory data. The response will tell the master about the current servo state:

Master sends an XBC every 10ms. The XBC message contains general information plus a section of private data for every attached servo. Assuming private data of 16 bytes in a single XBC, its easy to address 14-15 servos at a time using the full 255 byte message length. With every XBC, another slave will answer. Assuming we have 10 servos out, every 10th XBC the status of a particular servo is reported. This is an update rate of 100ms for all servos in this case:

xbc_request_rate : The time between to XBC's (ms)

nodes_with_response : The slaves that should respond

xbc_response_rate := (xbc_request_rate * nodes_with_response)

Example:

nodes_with_response : 10

xbc_request_rate : 10 [ms]

xbc_response_rate := 10 * 10 = 100 [ms]

1.2.2. Time triggered control systems

More general: Every application where several things have to be done in time or updated in time are well suited for XBC. The selective response from slaves will allow a close control loop for fast reaction and error handling.

1.3. A few definitions

A few words must be defined first:

Order:

With BITBUS an *ORDER* is a message send by the master to a particular slave. It contains data as well as a *command code (in command / response field).*

Reply:

An *ORDER* by the master is handled by an slave. The slave sends a *REPLY* in response to this order. The REPLY will always contain a error code and may contain processed data.

Broadcast:

A *BROADCAST* is an *ORDER* send to ALL slaves - regardless the nodes address. Only the master is allowed to send *BROADCASTS*!

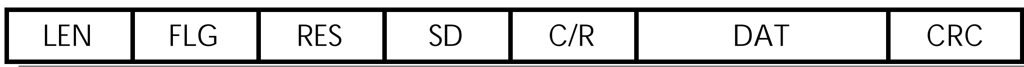
Response / Response permission:

In a master/slave system like BITBUS, the master controls bus access. When a master grants bus access to the slave, the slave got *response permission.*

2. XBC Layer 2

2.1. XBC message formats

2.1.1. USER PDU Format



LEN := length of data + 7, max. 255

FLG := std. BITBUS flags (R, SE, DE, TR)

RES := slave addr with response permission (or FFh for real broadcast)

SD := must be 0 (std. BITBUS source / destination task)

C/R := command/response

DAT := user data

CRC := CRC error check word

The user message format is as compatible as possible with standard IEEE1118 BITBUS messages. The major differences are:

- RES: Every node address is allowed except 0. In case 255 the message is a real broadcast - no replies are allowed. If RES < 255, the node will get response permission.
- The destination task id is not used. This is because there is no guarantee that on different nodes the same tasks use the same task numbers. Usually they do - but we have to stress the point that it is not guaranteed. A broadcast will fail in such a case. The destination task is defined using a different technique. See Chapter *4. BAPI Functions for XBC*. The source task id is stored inside the XBC protocol driver while a XBC is outstanding.

2.1.2. Layer2 XBC PDU Format

ADR	CTL	LEN	FLG	RES	XLEN	C/R	DAT	CRC

ADR := FFh (COMAND), RES (RESPONSE)

CTL := BFh (XID)

LEN := 0. If NOT 0 this is an IEEE1118 XID message!

FLG := std. BITBUS flags (R, SE, DE, TR)

RES := slave address with response permission (or FFh for broadcast)

XLEN := length of data + 7, max. 255

C/R := command/response

DAT := user data

CRC := CRC error check word

The layer 2 message format is what you see on the line if you dive in with a scope or spy. The XID (EXCHANGE IDENTIFIER) Message is used to transfer XBC's. In IEEE1118, XID is used for an node address assign technique. With 1118 XID, the first byte after the CTL field is LEN and this must never be 0. This is where we plug in XBC: With XBC, the LEN field is always 0. The real length is transported in the XLEN field (whitch is the SD field in a normal BITBUS message). So we can still implement XID service!

Please note that

- the basic message format is as close as possible to standard BITBUS and
- the broadcast (ffh) is send by the master only! A slave will set ADR = RES (0 < RES < 255)

2.1.3. UA - Unnumbered acknowledge control message

ADR	CTL	CRC

CTL := 73h (UA)

ADR := 1..250 (slave node address)

CRC := CRC error check word

The UA message is widely used in IEEE1118 for acknowledgement in NDM (NORMAL DISCONNECT MODE).

In XBC, the slave with response permission sends an UA immediately after the XBC receives to signal the master reception. It must be send within a single standard timeout period. The master will start to poll this node using UP message for a reply.

Please note that UA must not be send to the broadcast address 255 (ffh). It always uses the node address of the slave.

2.1.4. UP - Unnumbered poll control message

CTL := 33h (UP)

ADR := 1..250 (slave node address)

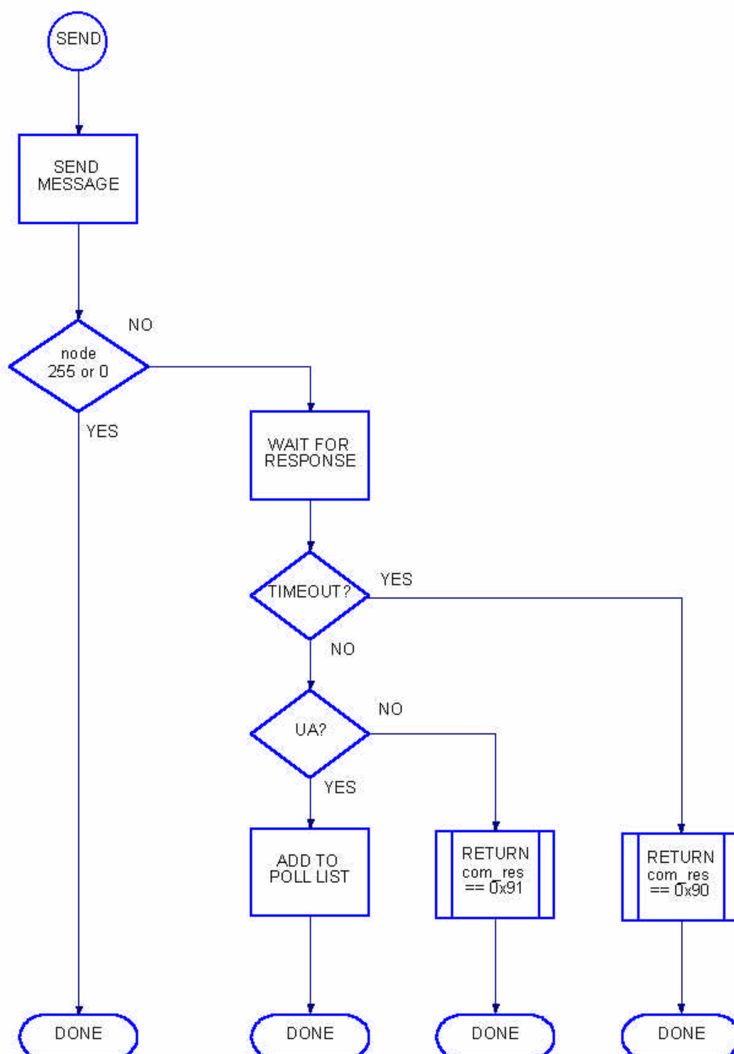
CRC := CRC error check word

The UP message is used by the master to poll a selected node. The slave with response permission uses UP to signals the master that a reply is still not available. If a reply is available the slave answers a UP with an XBC reply.

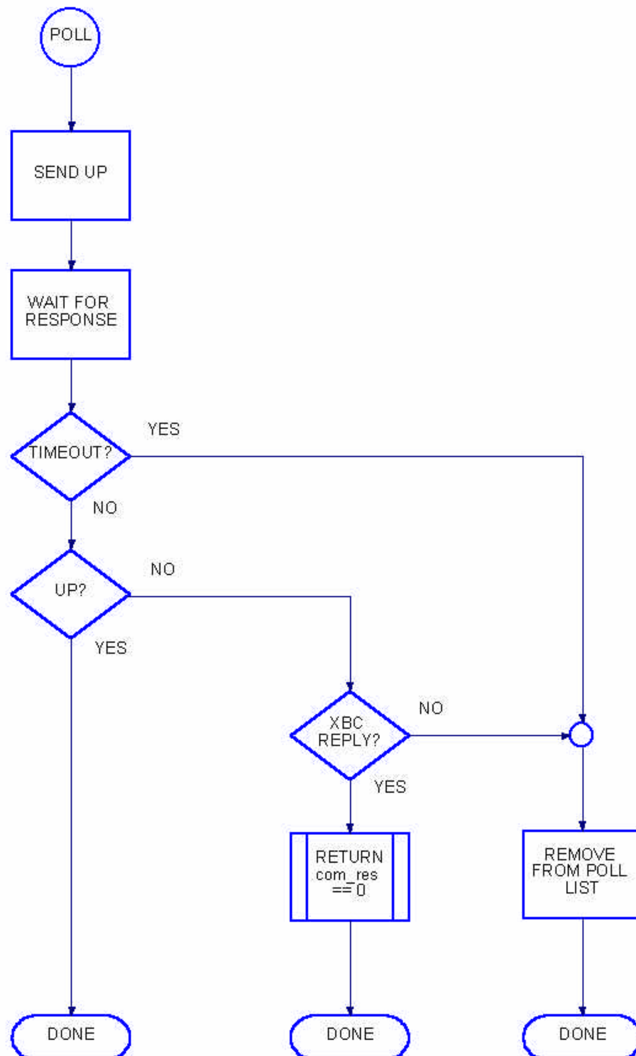
Please note that UP must not be send to the broadcast address 255 (ffh). It always uses the node address of the slave.

2.2. Master protocol

The master will reformat a user PDU to fit the the layer 2 PDU format. Therefore it has to move the length field to the SD field, clear the length field, set CTL to XID and adr to ffh. Then the PDU is send:



- If the response node address in the PDU (RES) is ffh, the message is a "real" broadcast and no answer is expected. However, it is recommended not to send more than one real broadcast back-to-back. The software should guarantee, that at least a 1ms quiet time is issued after a broadcast to prevent message buffer runout on the slave side.



- If the response node address in the PDU (RES) is other than ffh, the master starts an BITBUS layer2 timeout (~14ms @375 kBit/s). If the timeout expires before the slave signals valid reception using an UA message, the PDU is send back to the user with com_res = 0x91 (TIMEOUT). If any other response is received (should NEVER happen), the PDU is send back to the user with com_res = 0x90 (PROTOCOL).

The UA message will tell the master, that the PDU was accepted by the the slave and that a response will soon be available. The master will add the slaves node address to the poll list and mark it as "unnumbered".

When the master polls a slave node for an XBC reply, the UP (unnumbered poll) message is used instead of normal RR/RNR messages. The only possible answer to an UP is another UP or an XBC reply.

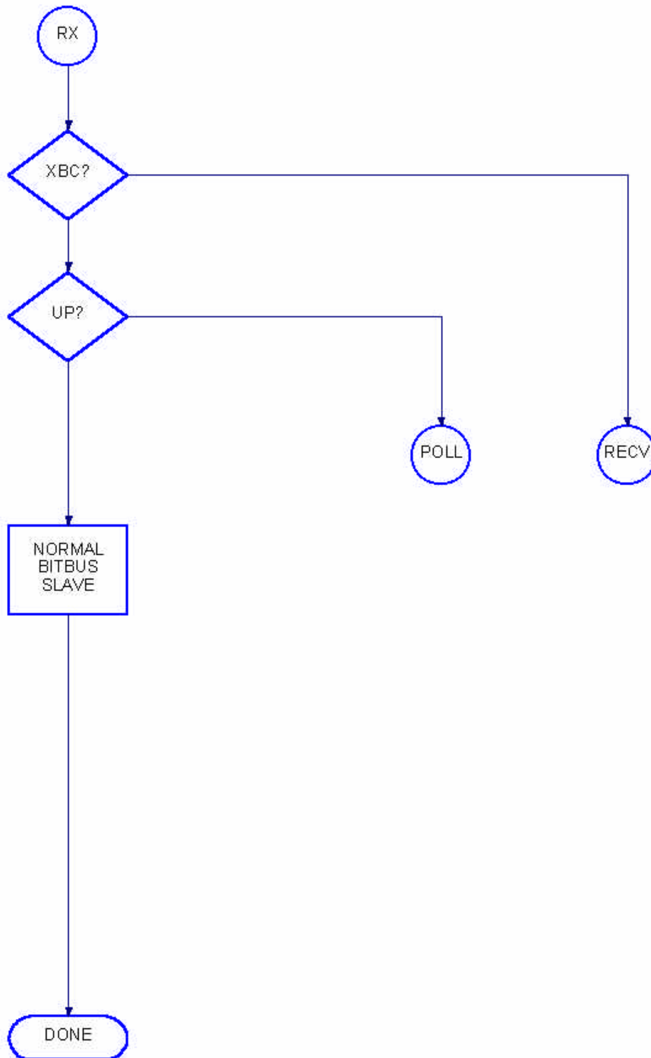
Because the XBC protocol is in any state clearly separated form the normal BITBUS operation, its implementation is easy and its operation is reliable!

If another xbc broadcast is requested by the user while there is still one outstanding, the outstanding one is discarded and the associated slave is removed from the poll list.

2.3. Slave protocol

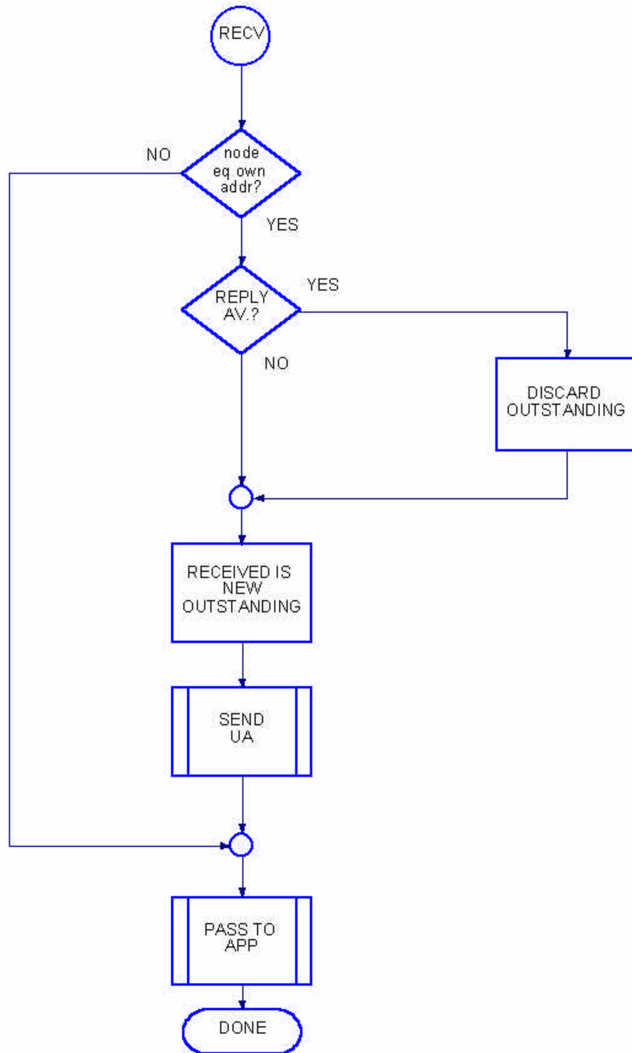
The slave side is a bit more complex, because of the buffer management. An implementor should be careful not to run in buffer management problems like lost buffers.

The XBC does not make any assumptions about the state of a master or slave. Therefore XBC works in both states of BITBUS communication: NDM (NORMAL DISCONNECT MODE) and NRM (NORMAL RESPONSE NODE)



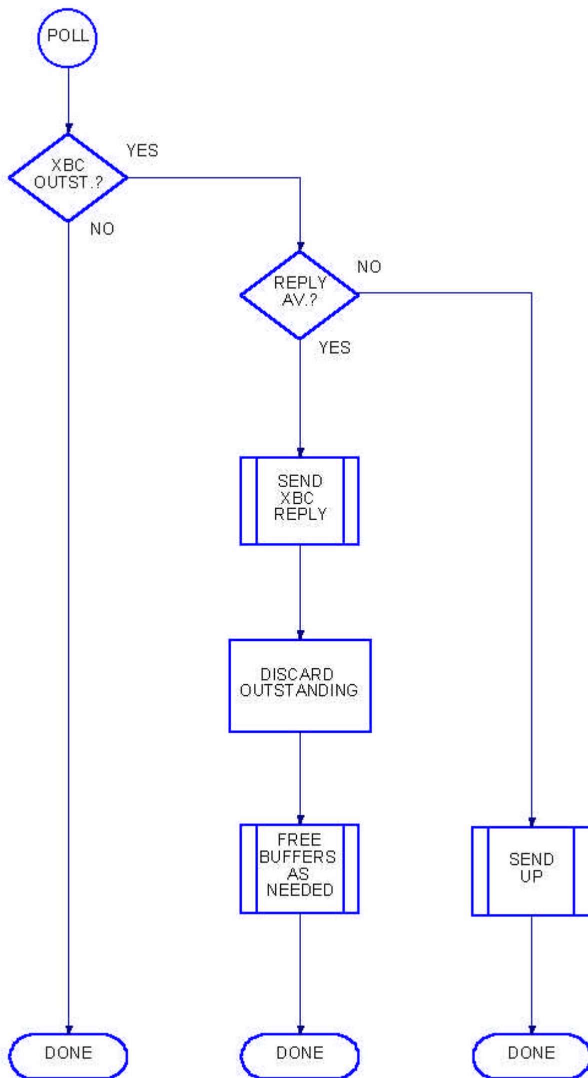
If an XBC is received by a slave, it will check the layer 2 PDU's ADR field first. If it is NOT equal to the own node address, it is a simple broadcast and the layer 2 will pass the message to the user (layer 7) without any further action.

If the node address is equal, the slave has to discard an oustanding XBC (if exists) without any further notification. The XBC will be acknowledged using the UA message and passed to the user (layer 7).

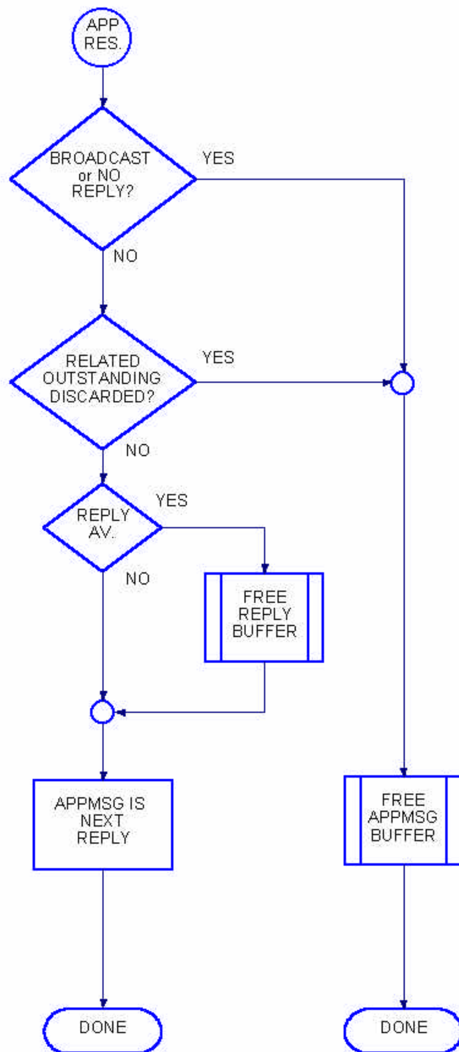


If a slave is polled using the UP command, it checks, if there is an outstanding XBC. If there is no, the slave will not take any action. If there is one, the slave checks whether the user (layer7) has already handled the XBC and if it is ready send. In this case the slave sends the XBC reply otherwise it sends an UP itself.

Please note, that the user (layer 7) will not have to worry about whether the current XBC needs an reply or not (real broadcast). The layer 2 is responsible to keep track of the messages and to free buffers as needed.



XBC SLAVE REPLY STATE DIAGRAM



3. XBC Layer 7

3.1. Master

On the masters side, things are simple. A normal BITBUS PDU is formatted. Instead of using the normal BITBUS SEND command, a different XBC send commend is used (See nect chapter BAPI). The XBC will be send to the slave and a reply by the slave will be passed back to the user. The reply's PDU is reformatted to meet the USER PDU specification. That's all.

3.2. Slave

On the slave side, the task that is designed to service the XBC messages must make itself known to the layer 2 driver. Please note that the SD field is not available. A special operating system call is used to do this job (see BAPI). Only one task can handle XBC's at a time.

The slave task will have to handle the message and to pass the response back to layer 2 even if it was a real broadcast and no answer will be send back to the master! The layer 2 will then free the message buffer as needed - so the user must not worry about buffers!

A typical XBC broadcast will include a global data section with data for all slave nodes (current time, process state, ..) and data sections specific for a specific nodes (control data, commands, ..).

The XBC reply will include node specific data only - like status, current machine statistics, ...

4. BAPI Functions for XBC

The slave provides a BAPI function to provide a common method to make a user task the XBC handler task.

I. BitbusSetRouting

This function is used to set a routing (make a task known as XBC handler). Currently, only the XBC routing can be set - but the function is open for future extensions (for example for XID routing).

Parameters:

"service" is used to identify the serive - currently this value is fixed to BITBUS_SERVICE_XBC.

"taskid" is the id of the task that is capable to handle the XBC orders.

Return value:

-/-

Prototype:

```
#define BITBUS_SERVICE_XBC   1

INT32    BAPICALL BitbusSetRouting   (UINT16    service,
                                      INT16     taskid);
```