GSBL Guide

Galacticomm's Software Breakthrough Library

for Worldgroup

Online Interactive Software

Galacticomm, Inc.

Information in this document is subject to change without notice. This document and any related software are sold “as is.” Galacticomm, Inc. makes no representations or warranties with respect to the contents of this document, or the software described, and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Liability for the information in this document, and for the software described herein, shall be limited to the purchase price of the document or software.

This document discusses computer software and other copyrighted materials that may be restricted by license agreements or other protections. This document shall not be used as a written exception to any license agreement, and shall not be construed by any party as a license, authorization, or waiver of rights by the owner or copyright holder of the software discussed.

Table of Contents

Introduction

The Galacticomm Software Breakthrough Library lets you take direct control of a DOS-based PC's communications hardware. The GSBL is a family of highly optimized assembler subroutines which you access as C-language functions.

The GSBL is a multi*user* communications tool as opposed to multi*tasking*. Many DOS and BIOS function calls were written on the assumption of only one user per PC. The GSBL, by contrast, was written on the assumption that there may be as many as 256 users simultaneously making requests and expecting responses from a single DOS PC.

The GSBL is the communications interface for Worldgroup Online Interactive Software. One of Galacticomm's architectural strategies is to isolate low-level communications considerations from higher-level applications considerations. The GSBL's assembler code handles the low-level hardware while Worldgroup's C code handles the higher-level issues of servicing user queries.

Most of this manual is the detailed descriptions of these routines as used in C-language programs. See pages 197 and 207 for examples of a very basic multiuser teleconferencing program written with GSBL routines.

To efficiently handle a very large number of channels on a personal computer, we've had to make some breakthroughs in the hardware as well the software. For example, the GalactiBox can hold sixteen inexpensive internal modems, all configured at the same COM port, without interrupt conflicts. Or a GalactiBoard can provide eight serial ports at the same COM port.

Architecture

Basically, you will write software which calls GSBL functions. The GSBL will govern data traffic flow to and from the communications hardware via data buffers. Your communications hardware will then connect with users' communications hardware. Users' software will send requests to your software, which will provide responses.

Buffers Implemented for Each Channel

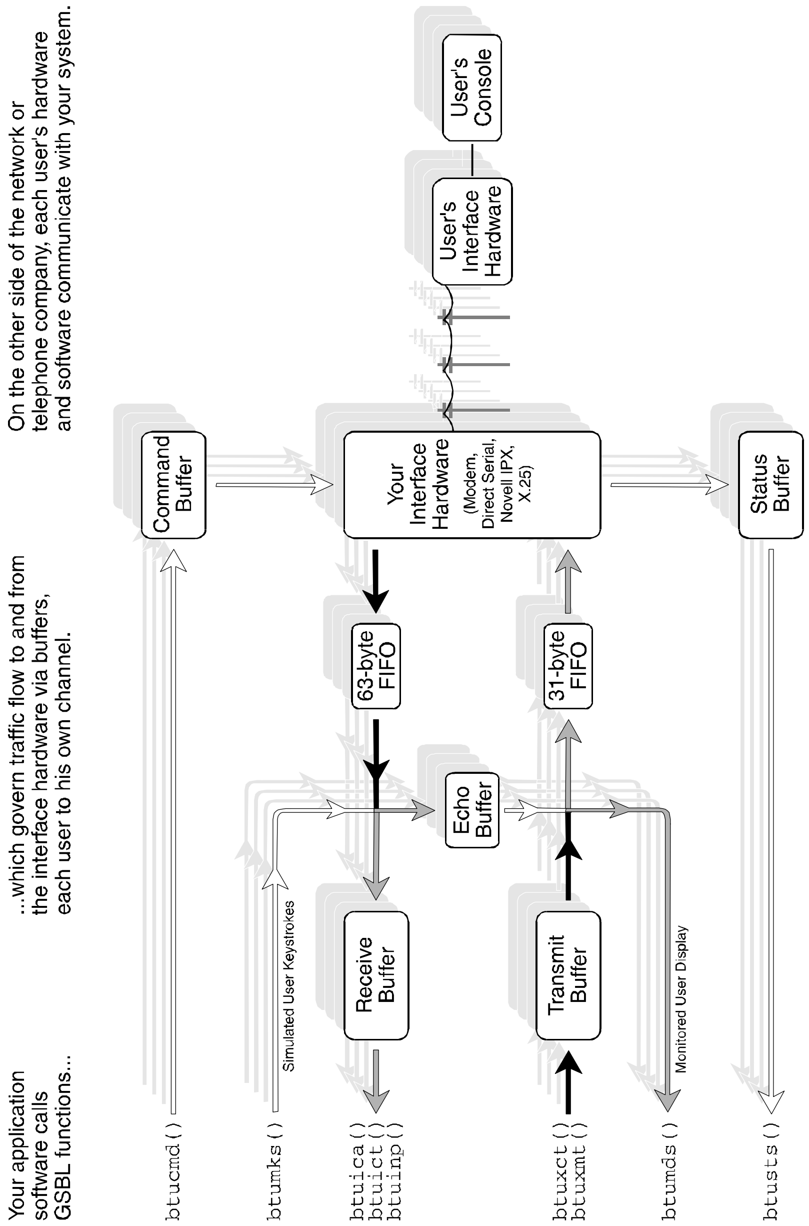
Buffer	Access via routine	Direction of access
Status	`btusts()`	input
Command	`btucmd()`	output
Receiver	`btuinp()` / `btuict()`	input
Transmitter	`btuxmt()` / `btuxct()`	output
Echo	(transparent)	—

The following buffers are only implemented for two channels — the "monitored" channels — as selected by btumon() (page 126) and btumon2() (page 128). Under normal circumstances, the first monitored channel is 00, the Local Session. The second monitored channel is any other channel the Sysop chooses to emulate.

Buffers Implemented for only One Channel at a Time

Buffer	Access via routine	Direction of access
Monitored display	`btumds()`	input
Simulated keystrokes	`btumks()`	output
Monitored display #2	`btumds2()`	input
Simulated keystrokes #2	`btumks2()`	output

Figure 2-1: How GSBL Fits Into Your Software Design


Your application software calls GSBL functions...
...which govern traffic flow to and from the interface hardware via buffers, each user to his own channel.
On the other side of the network or telephone company, each user's hardware and software communicate with your system.
btucmd()
btumks()
btuica()
btuict()
btuinp()
btuxct()
btuxmt()
btumds()
btusts()
Simulated User Keystrokes
Monitored User Display
Command Buffer
Receive Buffer
Transmit Buffer
Echo Buffer
63-byte FIFO
31-byte FIFO
Your Interface Hardware
(Modem, Direct Serial, Novell IPX, X.25)
Status Buffer
User's Interface Hardware
User's Console

Hardware Categories

The Galacticomm Software Breakthrough Library supports multiple users on DOS-based IBM-compatible PCs using several different kinds of hardware, which fall into these categories:

Hardware Category	Description	Examples
HAYES	Hayes-protocol modems	Equinox SuperSerial or GalactiBoard with external modems installed, GalactiBox with internal modems installed
XECOM	“Xecom-protocol” modems	Galacticomm Model 16 or Model 4 (no longer sold)
UART	RS-232 serial ports	GalactiBoard
LAN	Direct & virtual circuits	Novell NetWare Local Area Networks
X.25	Packet switching networks	PC XNet Card (also requires X.25 Software Option)

Figure 2-2: Summary of Differences Between Hayes and Xecom

	XECOM Category Hardware	HAYES Category Hardware
Baud rates available	300, 1200	any rate from 75 to 57,600
Phone ringing (see btusts())	Status 1	Status 3 input string: "RING"
Carrier detect (results of btucmd(chan,"A"))	Status 2	Status 3 input string: "CONNECT" for 300 baud "CONNECT 1200" for 1200 baud "CONNECT 2400" for 2400 baud etc. Status 12
Lost carrier (see btusts())	Status 1	Status 11
Busy	Status 66 ('B')	Status 3 input string: "BUSY" or "NO CARRIER"
Timeout waiting for dial tone	Status 84 ('T')	Status 3 input string: "NO DIALTONE" or "NO CARRIER"
Timeout waiting for carrier	Status 84 ('T')	Status 3 input string: "NO ANSWER" or "NO CARRIER"
Command complete successfully (see btucmd())	Status 2	Status 12 Status 3 input string: "OK" (in most cases)
Successful Reset of modem (bturst() return code meaning OK)	0	1
Short pause btucmd("..p..") and Long pause btucmd("..P..") within a string of commands	Takes place AFTER the commands preceding it	Takes place BEFORE all other commands in the string
Commands not supported on Hayes hardware (see btucmd())	^M Frequency monitor ^O set 110 baud speed ^X Analog loopback, originate mode ^Y Analog loopback, answer mode H Hold I Identify version i Identify revision L Line analyze l ("ell") 1200 baud error statistics	(these commands will generate a status code 13 with Hayes hardware)
Commands supported only on Hayes hardware (see btucmd())	(this command will generate a status code 63 "?" on XECOM category hardware)	^F set speed to 2400 baud

Hayes-Protocol Modems

In this category of hardware, the computer is connected in series with three devices: an RS-232-C serial interface device (called a UART); a Hayes-compatible modem; and a telephone line. The two functions of controlling the modem, and communicating through it, are handled using the same signals. This is accomplished using two modes of operation: the modem is either in command mode or in online mode.

Most modems used with PCs support the command protocol originated by Hayes Microcomputer Products, Inc., Atlanta, Georgia, for their Smartmodem series. The Galacticomm Software Breakthrough Library will work with modems that are compatible with the Hayes Smartmodem series of modems.

When Hayes-compatible modems are in the command mode, you instruct them to perform tasks by transmitting to them a string of characters of the form:

```
AT<command><parameters>ENTER
```

See your modem manual for a complete description of command and online modes, and the entire AT command set.

If you have a Hayes-compatible modem, and have connected it to your PC using a serial port that is equivalent to IBM's Asynchronous Communications Adapter (usually identified as COM1 or COM2 by the operating system), then you fit into the HAYES hardware category for the purposes of this manual. Many companies now make internal modems: a single card containing both a Hayes-compatible modem and a UART. An internal modem is plugged into the internal bus of the PC, and connected directly to the telephone lines. This architecture also fits into the HAYES category.

All other hardware in the HAYES category must include the 8250-type UART required by the UART category of hardware. The 16450 and 16550

UARTs satisfy this requirement. More on that in the upcoming discussion of RS-232 serial ports.

Xecom-Protocol Modems

The Galacticomm Breakthrough card Model 16 and Model 4 contained, respectively, 16 and 4 complete modems on a single ISA card. The computer talks to each of these modems over an 8-bit parallel bus. The functions of command and communications are not implemented over the same signals, but take four distinct paths: command, status, transmit and receive. This method was developed by Xecom Inc., Milpitas, California, for their compact MOSART 300/1200 baud modems.

Although the Xecom protocol may be a more efficient arrangement than the Hayes protocol, the Xecom technique is not in widespread use as is the protocol developed by Hayes. The syntax of commands on a Xecom-protocol modem is completely different from that on a Hayes-protocol modem.

The Galacticomm Software Breakthrough Library attempts to make those differences of as little concern as possible to the systems design, but some considerations must be made, particularly in the following areas:

Areas of Difference between Hayes and Xecom Protocols

- Starting an outgoing call
- Answering an incoming call
- Sensing loss of carrier (user has hung up)
- Certain test and measurement functions

Figure 2-2 (page 6) summarizes the differences between the Hayes and Xecom protocols. The teleconferencing example program on page 207 works on Xecom-compatible hardware.

UART Category Hardware: RS-232 Serial Ports

You could also use the Galacticomm Software Breakthrough Library to talk directly with any RS-232 compatible device, such as a terminal, printer, plotter, another computer, or many other types of devices.

In fact, Hayes-compatible modems are just one type of device that you may control using an RS-232 serial port, except that the Galacticomm Software Breakthrough Library provides extensive support for these particular devices (see the Hayes-protocol modem discussion, above).

You will need an interface card in your PC, equivalent to the IBM Asynchronous Communications Adapter. Most serial cards for the IBM PC family meet this requirement. They are designed to support what the operating system identifies as COM1 or COM2 (although the Software Breakthrough does not use DOS or BIOS to interface with these ports). One thing that the card must have is an integrated circuit called a UART of the 8250 family, which includes model numbers like 16450 and 16550. The 16550 model UART is highly desirable for its character buffering capability. The GalactiBoard 8-port serial card can supply you with a channel group of eight 16550 UARTs.

Novell Netware LAN Access

You can use the GSBL to interface to other programs across a Novell Local Area Network.

WARNING: Direct communication with LAN channels may be dangerous. Some sockets are reserved for file servers or other system functions. Be sure that the software always uses appropriate socket numbers (page 213).

The standard GSBL comes with the capability to handle IPX Direct, IPX Virtual, and SPX circuits.

IPX Direct Circuits are relatively hard-wired connections between two machines on the network, or multi-network internet. Each party must know ahead of time what network address to connect to.

IPX Virtual Circuits are connections that can be easily made and remade between machines on a network or internet. Also when no connection is specified, a “raw packet mode” reports packets received from any other machine, and can be used to transmit packets to any other machine.

SPX Circuits are higher-level connections with automatic error correction. Also under SPX, sessions are established and terminated between parties, which amongst other things allows either party to detect the presence or absence of the other party.

The baseline also supports GTC (Galacticomm Terminal Configuration) for IPX Direct, IPX Virtual, and SPX channels. This allows, for example, a terminal program to preprocess input for a server, one line at a time. That way every single keystroke doesn't get transmitted over the LAN as a separate packet.

See page 213 for more on LAN programming with the GSBL.

X.25 Packet Switching Network Access

X.25 is a way for computers to talk to other computers over a packet switching network. A packet switching network is very much like a local or long-distance telephone network, but with a different emphasis. A telephone network is called a "circuit switching" network. During a phone call, one handset is theoretically in continuous electrical contact with another handset that could be anywhere else in the world.

On a packet switching network, calls are also begun and ended — that's called switching. But during a call, the network does not need to maintain a connection during pauses in the data exchange. Neither is it a big problem if the network handles occasional peaks in traffic by slowing down all traffic just a little. However, the packet switching network is exchanging digital data and must detect and correct all transmission errors. Because of these differences, a packet switching network can be more economical for data exchange than a circuit switching network.

CCITT recommendation X.25 is titled "Interface between Data Terminal Equipment (DTE) and Data Circuit Terminating Equipment (DCE) for terminals operating in the packet mode and connected to public data networks by dedicated circuit". It was written by the International Telephone and Telegraph Consultative Committee (French abbreviation

CCITT) in Geneva Switzerland in 1976 and is amended every four years. For more information, you may want to read:

- "Technical Aspects of Data Communication", by John E. McNamara, published 1988 by Digital Equipment Corporation, chapter 24: Packet Switching
- "X.25 Explained", by R.J. Deasington, published 1988 by Ellis Horwood Books

See page 227 for more on programming with the GSBL/X25 communications interface.

Throughout the detailed sections of this manual, you will see annotations in the far left margin like:

HAYES to point out something specific to Hayes compatible hardware, or

XECOM to point out something specific to Xecom compatible hardware including the Galacticomm Breakthrough Models 16 and 4, or

UART to point out something specific to RS-232 serial ports such as the IBM Asynchronous Communications Adapter, or most any PC serial card, or

HAYES UART to point out something which applies to both the HAYES and the UART categories, or

LAN to point out something which applies only to Novell LANs, or

X.25 to point out something which applies only to the GSBL with the X.25 Software Option, GSBL/X25.

All variables and functions are available in all variations of the GSBL.

For example, the btux29() function (page 179) is useful only with the X.25 Software Option. Even if you do not own that option, there is a btux29() function that does nothing. This allows you to write C code that is compatible with all variations of the GSBL, and not get any undefined symbols when you go to link. See the variables btulan (page 21) and btux25 (page 25) for ways your software can determine what variation of the GSBL it is linked with.

ASCII versus Binary Input Modes

There are two methods for receiving data using the Galacticomm Software Breakthrough Library.

ASCII method means that you are expecting *text* from the user: information which consists mainly of printable characters (letters, numbers and punctuation). In this case, certain control characters are defined, and will be handled specially by the GSBL when they are received from the user. For example, CR (carriage return) terminates each input line from a user. Also, a user can use the BACKSPACE key to delete characters on his input line. You would use the ASCII method for menus, or question-and-answer sessions with a user.

Binary input mode means that there are no special characters or translations. You might use this mode to download or upload data files to/from a user, using a transparent binary protocol such as XMODEM.

In ASCII input mode, the following characters are treated specially when received:

ASCII Mode Special Input Characters

Hex	Dec	ASCII	Function	Assignment
00H	0	NUL	(ignored)	
08H	8	BS	backspace	`btubse()`, page 37
0DH	13	CR	carriage return	`btutrm()`, page 169
0FH	15	SI	output-abort	`btutru()`, page 171
11H	19	XON	output-resume	`btuxnf()`, page 193
13H	21	XOFF	output-pause	`btuxnf()`, page 193
7FH	127	DEL	backspace	`btuxlt()`, page 184

These GSBL routines may be used to assign the same functions to other characters. The default characters are shown here.

There are other functions that only apply to ASCII input mode:

ASCII Mode Special Function	Routine That Controls the Function
Input translation	`btuxlt()`, page 184
Custom handling	`btuchi()`, page 44
Input interception	`btuchi()`, page 44
Idle receiver handling	`btuche()`, page 42
Input line length limit	`btumil()`, page 122
Input word-wrap	`btumil()`, page 122
Automatic linefeed	`btulfd()`, page 114
Echo on/off	`btuech()`, page 84

See page 46 for the exact sequence of input processing in ASCII mode.

ASCII input is achieved using the GSBL routine btuinp() (page 108). This routine will extract a string of characters from the receive buffer of a channel. The input string ends with a CR, but when read by btuinp(), the CR is replaced with a 0-byte terminator. A status 3 is generated (refer to btusts(), page 154) when a complete line has been received, and btuinp() is used to retrieve that line.

Binary method means that none of the input characters listed above are treated specially, nor does any kind of character translation take place. Binary input is achieved using the GSBL routines btutrg() (page 167) and btuict() (page 105). None of the special handling shown in the above two tables occurs when these routines are used to receive data.

btutrg() chooses between Binary and ASCII input modes. With a second parameter of zero, ASCII mode is chosen. With a second parameter greater than zero, Binary mode is chosen, and that nonzero parameter is the number of bytes you want to receive at a time, in one block. A status 4 is generated (see btusts(), page 155) when a complete block has been received, and btuict() is used to retrieve that block. Figure 2-3 (page 14) illustrates the differences between the ASCII and Binary input methods.

Figure 2-3: Comparison of ASCII to Binary INput Modes

	ASCII Input Mode	Binary Input Mode
How to select the input mode:	`btutrg(chan,0)`	`btutrg(chan,NBYT)`
Input unit:	One `CR`-terminated line	`NBYT` bytes
When is an input unit complete?	When a carriage return has been received	When `NBYT` bytes have been received
Status generated to indicate that the input unit has been received:	3	4
How to get the received input unit out of the receive buffer:	`btuinp(chan,inbuff)`	`btuict(chan,inbuff)`
Return code if no data is available:	-1	-2
Return code if some, but not enough, data is available:	-1	-2
Return code if a complete input unit is available (and has been moved to the `inbuff` buffer):	number of bytes received (less the `CR`)	`NBYT`
How many bytes were transferred into `inbuff`, in either of the above 2 cases?	`strlen(inbuff)`	global variable `ictact`
Format of data in the `inbuff` buffer:	One `CR`-terminated line, with the `CR` replaced by a 0-byte	`NBYT` bytes

ASCII versus Binary Output Modes

The ASCII output mode includes such features as:

- Automatically appending linefeeds to carriage returns
- Output word-wrap and paragraph reflow
- Suspending output while input is in-progress
- ANSI-graphics discrimination
- Output “abort” character
- Single-character prompts

This mode would be used during conversational interaction with a user: asking questions, parsing replies, accepting commands, reporting results.

When the GSBL transmits in Binary output mode, no character is treated differently from any other. This mode would be used during a transparent computer-to-computer interchange, such as an XMODEM protocol upload or download.

The selection of ASCII versus Binary output modes is as follows:

Use btuxmt() for ASCII output (refer to page 189)
Use btuxct() for Binary output (refer to page 182)

NOTE: The actual transition from Binary to ASCII output modes occurs only when the output buffer empties completely. For example:

```
btuxct(chan,1,"\7");
btuxmt(chan,"WARNING\r");
```

This will probably transmit the “WARNING\r” message in Binary mode, since the buffer will probably not be empty when that message is stuffed into it. Either a status 5 (page 156) or a btuoba(chan) == outsiz-1 condition is necessary to be absolutely sure that a subsequent btuxmt() will transmit in ASCII mode.

The following routines are associated with ASCII output and affect the operation of btuxmt():

`btutsw()`	Output word wrap
`btuhcr()`	Hard carriage-returns
`btuscr()`	Soft carriage-returns
`btulfd()`	Linefeeds appended to carriage returns
`btutru()`	Output abort character
`btuxnf()`	XON/XOFF handshaking
`btupmt()`	Prompt characters
`btucmd()`	Enable or disable ANSI graphics with the `[` and `]` commands

Most of the processing for these features is performed by btuxmt() — and not by the interrupt-service routines, thereby increasing channel throughput. None of these features are in effect when you use btuxct().

Global Variables

To use any of the following variables in a program, you should declare them in the variable declaration section of your C-language program in the manner shown under **DECLARATION**. The declaration must appear before any use of the variable. The header file BRKTHU.H includes all of these declarations.

btudtr

VARIABLE NAME

btudtr — DTR level after channel reset

DECLARATION

extern int btudtr;	what to do after bturst(): 0 = lower DTR for 2 seconds and then raise 1 = leave DTR high always

DESCRIPTION

HAYES UART

This command affects all channels that are based on the 8250-type UART. It specifies what happens after the channels are reset with bturst() (page 138). After reset, the DTR signal is set to this value for 2 seconds, then it is set high (active).

To vary your handling of individual ports, you could set btudtr just before every call to bturst().

btuhrt

VARIABLE NAME

btuhrt — high-rate 65536 Hz counter

DECLARATION

extern unsigned long btuhrt;

DESCRIPTION

This 32-bit integer increments at approximately 65536 Hertz. You'll probably make the most use out of this by taking two samples and subtracting them — the unsigned difference reflects the time between the samples — similar to what ticker does (page 30), only faster.

You can use the upper 16 bits of btuhrt as a 1 Hz counter.

Actually btuhrt only simulates this rate. It really changes at a rate controlled by btumxs() (page 129). btuhrt increments in spurts, in a manner calculated to keep btuhrt looking like a 65536 Hertz counter. You may not be able to use btuhrt to measure time too finely. With btumxs(2400), for example, about 290 times a second, btuhrt increases by approximately 226 (226 x 290 = 65540, which is about 65536). A new btuhrt value will be available only 290 times a second, not 65536 times a second, so you couldn't measure 1 millisecond intervals.

CAUTIONS

When reading this variable, you should temporarily disable interrupts to avoid skew between the 16-bit halves of the value. For example, we use this code in Worldgroup:

```
unsigned long
hrtval(void)
{
     unsigned long hrtsmp;

     dsairp();
     hrtsmp=btuhrt;
     enairp();
```

```
        return(hrtsmp);
}
```

btulan

VARIABLE NAME

btulan — LAN capability and status flags

DECLARATION

```
extern int btulan;

#define BTLIPXD  0x0001   /* bit 0:  IPX Direct circuits supported   */
#define BTLIPXV  0x0002   /* bit 1:  IPX Virtual circuits supported  */
#define BTLSPX   0x0004   /* bit 2:  SPX circuits supported          */
#define BTLI7A   0x0100   /* bit 8:  IPX driver is loaded            */
#define BTLSPXL  0x0200   /* bit 9:  SPX is loaded                   */
```

DESCRIPTION

Your C program could use btulan to make a report on these conditions:

```
printf("Your GSBL supports:\n");
if (btulan&BTLIPXD) {
     printf("   IPX Direct Circuit channels\n");
}
if (btulan&BTLIPXV) {
     printf("   IPX Virtual Circuit channels\n");
}
if (btulan&BTLSPX) {
     printf("   SPX channels\n");
}
if ((btulan&(BTLIPXD+BTLIPXV+BTLSPX)) == 0) {
     printf("   no LAN access at all\n");
}
printf("The IPX driver %s been loaded at interrupt vector hex 7A\n",
       btulan&BTLI7A ? "has" : "has not");
printf("SPX %s installed on this node\n",
       btulan&BTLSPXL ? "is" : "is not");
```

CAUTIONS

Note that the first three bits are always available when your program is running. The last two are available only after calling btuitz() and before calling btuend(). The above program must include the header file BRKTHU.H, and be run between calls to btuitz() and btuend().

bturno

VARIABLE NAME

bturno — registration number

DECLARATION

```
extern char bturno[];      /* 8 digits plus a <NUL> */
```

DESCRIPTION

This character array contains the 8-character software registration number issued to you when you purchased the Galacticomm Software Breakthrough Library. It consists of 8 ASCII digits followed by a NUL (ASCII 0) terminator.

CAUTIONS

Don’t let your copy of the Galacticomm Software Breakthrough Library fall into unscrupulous hands. Unauthorized copying of this software or documentation is a violation of federal law, specifically Title 17, USC Section 506. Violators may be subject to a $25,000 fine, or imprisonment, or both.

btusrs

VARIABLE NAME

btusrs — number of users licensed

DECLARATION

extern int btusrs;

DESCRIPTION

This variable is the number of users licensed for your copy of the GSBL.

In Worldgroup, this variable is used to display the version code suffix (e.g. -32) in the SERVER UP audit trail message.

Channels defined beyond this number will always be non-hardware channels (bturst() will always return -10, see page 138). Non-hardware channels may be useful in simulating a user channel. See about the monitor feature on page 126 (and page 128).

btuver

VARIABLE NAME

btuver — GSBL version code

DECLARATION

extern char btuver[];

DESCRIPTION

This string is the software revision for the GSBL.

You could use it to display the version somewhere:

```
printf("Based on GSBL-%s for %d users",btuver,btusrs);
```

Or, you could use this variable to do different things based on the version code of the GSBL, for example:

```
if (strcmp(btuver,"J") == 0) {
     jspecific();
}
else if (strcmp(btuver,"K") == 0) {
     kspecific();
}
else {
     allelse();
}
```

btux25

VARIABLE NAME

btux25 — whether or not GSBL supports X.25

DECLARATION

```
extern int btux25;          0=GSBL does not support X.25
                               1=GSBL supports X.25
```

DESCRIPTION

This global variable can be used to take special action based on whether the GSBL supports X.25 or not. For example, you might code your program to generate an error message when it is configured to talk to X.25 channels but the GSBL does not support them. The return value of btusdf() for channel type 4 (page 147) may be used in the same way.

CAUTIONS

The value of btux25 is not defined until after you call btuitz().

ictact

VARIABLE NAME

ictact — actual number of bytes read by btuict()

DECLARATION

extern unsigned ictact;

DESCRIPTION

When you use btuict() (page 105) to retrieve data bytes from the input buffer, as many as are available are copied, up to the trigger count (the trigger count is set by btutrg(), page 167). When less than the trigger count of bytes are available, the bytes are transferred anyway, but they are not taken out of the input buffer. This is when btuict() returns -2 (indicating insufficient bytes are available). In this situation, the global variable ictact is the way you have of determining how many bytes were copied into your buffer.

lanrev

VARIABLE NAME

lanrev — LAN SPX revision

DECLARATION

extern char lanrev[2];

DESCRIPTION

LAN

This array contains the SPX version numbers, minor version first:

```
printf("SPX version %d.%d\n",lanrev[1],lanrev[0]);
```

CAUTIONS

This array is only available after btuitz() has been called and only if SPX is available.

lansca

VARIABLE NAME

lansca — LAN SPX connections available

DECLARATION

extern int lansca;

DESCRIPTION

LAN

This is the number of SPX sessions that your program can support. To increase this number, you might be able to increase the SPX CONNECTIONS parameter in your Netware SHELL.CFG or NET.CFG file:

```
SPX CONNECTIONS = 100
```

Refer to your Novell documentation for proper placement of this line.

CAUTIONS

This value is only available after btuitz() has been called.

lansop

VARIABLE NAME

lansop — LAN socket opened by btusdf()

DECLARATION

extern int lansop;

DESCRIPTION

If you use btusdf() to define a Netware LAN channel with a socket of 0, then IPX will assign an unopened socket for you. The socket it picks is then retrievable from this lansop variable.

The socket number is in convenient “lo-hi” byte order.

CAUTIONS

lansop is only defined for hardware channels (i.e. bturst() returns non-negative), and only after calling btusdf().

ticker

VARIABLE NAME

ticker — second timer

DECLARATION

```
extern unsigned ticker;  /* 0 to 65535 and back again*/
```

DESCRIPTION

Every second, the contents in this variable increments. After it reaches 65535, it goes back to 0 and starts over again. According to the properties of 16-bit unsigned arithmetic, you can sample the value of ticker at two different times, and the difference between the values will reflect the time between the samplings, as long as that time is less than 65535 seconds (about 18 hours).

If you need more resolution in your time measurement, use btuhrt (page 19).

CAUTIONS

This variable increments at a rate very close to 1 Hz (once a second), but don't set your watch by it.

x25ign

VARIABLE NAME

x25ign — count of ignored incoming packets

DECLARATION

extern int x25ign;

DESCRIPTION

An incoming packet will not be recognized if:

- the packet is not a DATA, CALL REQUEST, CALL CONFIRM, or CLEAR packet.
- an incoming data packet has a logical channel number (virtual circuit number) that does not agree with that of any channel defined (by btusdf()) for the GSBL.

You can set x25ign to 0 also. It will be incremented every time a packet is ignored.

x25udt

VARIABLE NAME

x25udt — user data reporting flag

DECLARATION

extern int x25udt;	set to 1 to capture the User Data Field of incoming packets

DESCRIPTION

If you set this flag to a 1, then the User Data field of an incoming call request packet is tacked onto the end of the RING xxx CALLING xxx string as a fifth word, as in:

```
RING 81020551 CALLING 30448512 IGNEOUS-7
```

CAUTIONS

The User Data field should be ASCII and have no imbedded control characters. The input buffer for the channel must be large enough (as specified by btusiz(), btulsz(), or btubsz()) to hold the entire string.

Library Routines

This section, by far the largest in this manual, describes each of the GSBL routines in detail.

The prototypes for these routines, plus many of the constants and symbols you can use in programming with the GSBL, are in the header file BRKTHU.H.

btubrt

SUBROUTINE NAME

btubrt — set channel’s baud rate

SYNOPSIS

err=btubrt(chan,bdrate);	
int err;	zero means OK, -3 means bad baud rate
int chan;	channel number
unsigned bdrate;	baud rate, in bits per second For the Model 2408, bdrate must be one of the values: 300 600 1200 or 2400 For 8250-interfaced serial devices, bdrate must be between 75 and 57,600. For the higher baud rates, the rate should divide evenly into 115,200. The default baud rate is 2400 baud, after a channel is reset by bturst().

DESCRIPTION

HAYES UART

Use this routine only on Hayes or RS-232 serial channels (page 7), not on Xecom modems. This function specifies the bits per second rate for a specific communications channel. Only the UART is affected. If you are also using a modem, the modem’s baud rate must be set another way. To the blackboard for a moment: Computer talks to UART, which talks to Modem. The UART (Universal Asynchronous Receiver/Transmitter) translates characters from the computer into a single signal with a pulse for each bit. The modem translates this into something that can be transmitted over phone lines (or other long distance media).

HAYES

The two most common ways that a modem’s baud rate changes are:

Case 1: incoming calls. When you send the command to answer an incoming call, the modem responds with CONNECT (for 300 baud) or with CONNECT <baud rate> for other baud rates (1200, 2400,...). After sending that message, the modem communicates at the specified baud rate — you should use btubrt() to change the UART to that baud rate.

Case 2: on demand. With the modem in command mode, simply change the UART baud rate using btubrt() and issue the null Hayes modem command **AT**

ENTER. You should get an OK response from the modem (followed by CR LF) at the new baud rate.

The baud rates for a Model 2408 Breakthrough card are: 300 600 1200 and 2400. These are the valid values for the bdrate parameter of btubrt(), when operating on a channel assigned to the Model 2408.

HAYES UART

Valid baud rates for 8250-interfaced serial devices are between 75 and 57,600. If the value you supply does not evenly divide into 115,200 then you will have some truncation error, particularly with higher baud rates. For example, if you supply 110 for the bdrate parameter, the actual baud rate will be 110.03 (plenty good enough). But if you supply 35000, then the actual baud rate will be 38,400 (way off for sure!).

Technical Note: the number [115200/bdrate] is programmed into the 8250 baud rate divisor to achieve the baud rate bdrate. That is, “the greatest integer less than or equal to 115200/bdrate”. To compute:

actual baud rate = 115200 / [115200/bdrate]

Non-Technical Note: the following are perfectly good values for bdrate on 8250-interfaced channels:

75	110	150	300	600	1200	1800
2000	2400	4800	9600	19,200	38,400	57,600

CAUTIONS

When dealing with *non*-16550 UART serial hardware, you should not give btubrt() a baud rate value greater than the value set by btumxs() (maximum data speed, see page 129). If you do, you will lose characters on input, and have slow output. This is because btumxs() actually sets the modem service rate — the frequency at which all modems are polled for input and output — with a 21% safety margin. But btubrt() sets the frequency of the communication clocks in the UART device, and therefore only affects the bit rate within each byte.

When dealing with 16550 UART serial hardware, however, you can give btubrt() a baud rate value up to 4 times greater than that set by btumxs(). Since 16550s buffer data, you can get away with servicing them less often and still not lose characters.

RETURNS

-3	baud rate is not valid for the hardware being used
0	baud rate has been set correctly

btubse

SUBROUTINE NAME

btubse — set backspace-echo character

SYNOPSIS

```
err=btubse(chan,bschar);
int err;                zero means OK
int chan;               channel number
char bschar;            how to handle a user’s backspace keystroke:
                        00      like any other character
                        08      delete last character (if any)
                                echo backspace - space - backspace
                                (default condition)
                        NN      delete last character (if any)
                                echo NN
```

DESCRIPTION

This function specifies how to handle the user backspace keystroke. It applies only to ASCII input mode, not to binary input mode (page 12). While in ASCII input mode, any input character that translates to a value of 8 will be taken as a logical BACKSPACE — that is, it will delete the most recently accepted character on the current input line, if there is one. Note: the translate feature (see btuxlt(), page 184) can make the DEL character (7F hex) be treated just like the backspace character. In fact, this is the case in the default translate table (page 185).

The question that btubse() attempts to answer is: how will this character-delete operation be echoed back to the users terminal? On a CRT, you will probably want a backspace-space-backspace to do the work, erasing the previous character and leaving the cursor in the position where that character used to be.

If, however, a user is using a hardcopy device such as a teletype machine, it might be better to echo a backslash (\), or something, and let him figure out what he has typed and what he has deleted.

So, if the bschar parameter is 8, the backspace-space-backspace method will be used. This is the case by default, after you reset a channel (see bturst(), page 138). If bschar is some other nonzero value, then that is what's echoed upon each backspace that the user types (when it actually deletes a character on his input line, that is).

Now, if the bschar parameter is 0, then this idea of the user's backspace key deleting characters from his input line is thrown out altogether: ASCII code 8's are simply passed into the input buffer just like any printable character.

CAUTIONS

If you are using standard CRT terminals, or terminal emulators, there is no need to call this routine.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	all is well

btubsz

SUBROUTINE NAME

btubsz — respecify input and output buffer sizes

SYNOPSIS

err=btubsz(chan,isiz,osiz);

int err;	zero means OK, -5 means bad sizes
int chan;	channel number
int isiz;	new size of input data buffer
int osiz;	new size of output data buffer

DESCRIPTION

This command adjusts the size of the input and output data buffers for a particular channel. You might use this to temporarily increase the input or output buffer for special conditions. You must use integral powers of two, and their sum must not exceed that originally specified by btusiz() or by btulsz() (see cautions, below).

Typical application: YMODEM uploads. This protocol requires an unusually large input buffer. Since YMODEM data packets are 1024 bytes long plus overhead, you will require an isiz parameter of 2048, which allows 2047 bytes in the input buffer. Let’s say a much smaller value had been supplied for isiz in your initial call to btusiz():

```
btusiz(NCHAN,256,2048);
```

parameter isiz =   256
parameter osiz = 2048

These are reasonable values for a multiuser online service such as Worldgroup. Particularly in terminal mode, the heaviest traffic on an online service is usually in the direction toward the users. Output traffic may include an entire screenful of text: 80 X 24 = 1920 characters plus overhead. Input traffic, coming from each user’s keyboard, is relatively light, requiring an input buffer big enough to hold an 80-character line of text. C/S mode reduces this disparity to a great extent, but output still generally outnumbers input.

Now when a YMODEM upload is required, you can swap the capacities of the data I/O buffers for channel chan. Here's a subsequent call to btubsz():

```
btubsz(chan,2048,256);
```

parameter isiz = 2048
parameter osiz =   256

Then after YMODEM upload has completed, you return the buffers to their original condition:

```
btubsz(chan,256,2048);
```

parameter isiz =   256
parameter osiz = 2048

CAUTIONS

The isiz and osiz parameters must each be an integral power of two (e.g. 128, 256, 512, 1024, 2048, 4096 etc.).

The sum of the isiz and osiz parameters must not exceed the space reserved for I/O buffers. That is, the sum of the isiz and osiz parameters that you specified in your original call to btusiz() or to btulsz() (whichever you used — refer to page 152 or 117).

The actual capacities of the I/O buffers are one less than the values specified in parameters isiz and osiz. If you mean to temporarily adjust input and output buffer sizes, be sure to use btubsz() to restore the original buffer sizes. Also, bturst() (page 138) restores the original buffer sizes. btubsz() automatically clears the data input and output buffers.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)

-5 buffer sizes are invalid: either one of them is not an integral power of two (128, 256, etc.), or their sum exceeds the total data buffering capacity of the channel, as set by the original call to btusiz() (see page 152)

0 all is well

btuche

SUBROUTINE NAME

btuche — enable calling of btuchi() when echo buffer becomes empty

SYNOPSIS

err=btuche(chan,onoff);	
int err;	zero means OK
int chan;	channel number
int onoff;	1=enable calls; 0=disable

DESCRIPTION

Normally, the routine specified in btuchi() (see page 44) is only invoked when an input character is received, and your program has no way of telling when the channel's echo buffer is empty. This function allows you to turn on or off a feature whereby each time the channel's echo buffer becomes empty, the btuchi() rouadr function is invoked with the channel number and pseudo-key-code of -1 as parameters. For example:

```
:
btuchi(3,&idle);
btuche(3,1);
:
idle(chan,c)
int chan,c;
{
     if (c == -1) {
          chiout(chan,'~');
     }
     else {
          chiout(chan,c);
          return(0);
     }
}
```

In this example, '~' is constantly output on channel 3 when no input is being received. When an input character is received, it echoes normally, then the '~' output continues. Note that this process will not begin until the first character is received.

CAUTIONS

When your btuchi() rouadr function is invoked with key-code equal to -1, no return value from the function is expected. Coding such as the example above may generate warnings by some compilers, since the function will sometimes return with an explicit return value, and at other times not. You can solve this, with only a small loss of efficiency, by coding the routine so that it always returns 0.

This function exists mainly to support certain advanced real-time protocols available for Worldgroup. Be sure that you fully understand the btuchi() function before attempting to use btuche().

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	all is well

btuchi

SUBROUTINE NAME

btuchi — set input character interceptor

SYNOPSIS

err=btuchi(chan,rouadr);	
int err;	zero means OK
int chan;	channel number
char (*rouadr)();	address of character interceptor routine (NULL to remove)

xc=(*rouadr)(chan,c);	the character interceptor routine
char xc;	new input character (0=ignore)
int chan;	channel number
int c;	the untranslated character received from the channel (or -1, see btuche(), page 42)

chiinp(chan,c);	character input utility
int chan;	channel number
char c;	character to simulate input

chiout(chan,c);	character output (via echo buffer)
int chan;	channel number
char c;	character to output

chious(chan,str);	string output (via echo buffer)
int chan;	channel number
char *str;	string to output

chiinj(chan,s);	status inject utility
int chan;	channel number
int s;	status to inject

DESCRIPTION

The btuchi() function provides a simple but powerful facility for customized handling of received characters. You can install your own C-language or assembly language routine so that it is invoked upon every byte received from a channel. This character interceptor of yours will only operate in the ASCII input mode. In Binary input mode, it is suspended.

This is one of the most complex and potentially dangerous routines in the Software Breakthrough Library. You should not attempt to use it if concepts like interrupts and reentrant code are unfamiliar to you.

We will describe the C-language method here. For the assembly language method, refer to your C compiler documentation sections on linking to assembly language routines.

To make your own input character interceptor, you must first code a C-language (or C-language compatible) function that accepts as parameters a channel number and a character. The function should return a single character value. See the synopsis of rouadr above. The syntax of that synopsis: (*rouadr)(...) simply means that the symbol rouadr represents a pointer to a function (which returns a value of type char, because xc is type char).

Next, you must call btuchi() and pass a channel number and the address of this custom handler routine of yours. If you want your custom handler to remain in effect constantly, you may call btuchi() right after the channel is defined (see btudef(), page 80).

Our character handler routine replaces the character translation function (refer to btuxlt() on page 184). However, the effects of all functions associated with ASCII input mode are still in effect. Again, your interceptor routine is not in effect when the channel is in Binary mode (page 12).

The input of a single character goes through the following sequence of steps during the ASCII input mode:

	Input Processing	Associated Library Routine
1.	Formatting, Overrun, Parity error checking	`btuerp()`
2.	Input lockout	`btulok()`
3.	ASCII mode or Binary mode?	`btuict()`
4.	XON/XOFF handling	`btuxnf()`
5.	Output abort character	`btutru()`
6.	Translate Table (or `btuchi()` routine)	`btuxlt()/btuchi()`
7.	Backspace	`btubse()`
8.	Line terminator	`btutrm()`
9.	Line length limit or input word wrap	`btumil()`
10.	Input buffer capacity	`btusiz()/btulsz()`
11.	Echo	`btuech()`

This table is provided for you to judge the interdependencies of the above library routines. For example, if your btuchi() routine returns a character code 8 (ASCII BS, backspace) then the backspace handler will swing into effect (it may echo a space-backspace-space sequence, unless you have adjusted btubse()).

To de-install your custom character interceptor routine, simply call btuchi() with a rouadr parameter value of NULL. bturst(), in resetting a channel, also de-installs any interceptor that may have been in effect.

EXAMPLE

Here is an example character handler that strips the high-order bit 7 off of each byte received:

```
char strip7(chan,c)
int chan;
char c;
{
     return(c&127);
}
```

Then the following call to btuchi() installs this routine on channel chan:

```
btuchi(chan,strip7);
```

The following routine de-installs it (restores default character handling) on channel chan:

```
btuchi(chan,NULL);
```

ADVANCED USAGE

Four routines in the Galacticomm Software Breakthrough Library are provided only for use by your input character interceptor (btuchi()) or real-time interrupt (bturti(), page 142) routines. These are

```
        chiinp()
        chiout()
        chious()
and     chiinj()
```

See synopses above. You may use these to cause all kinds of things to happen upon receipt of characters.

For example, in a “battleship” game, you may want the CTRL-E-D-X-S diamond on your user’s keyboard to control his screen cursor using ANSI-standard cursor control codes. Your character handler would echo (using chiout() or chious()) the appropriate cursor control codes upon receipt of the cursor commands from the user’s keyboard.

Also, keyboard macros could be implemented using chiinp(). You could translate special control codes from a user’s keyboard into a whole stream of characters. For example, you may want a user to be able to press CTRL+G when he means *go forward* ENTER.

Yet a third example would be inter-channel-chat. Let’s say you make the following character interceptor routine:

```
uchat(chan,c)
int chan;
char c;
{
     chiout(chana,c);
     chiout(chanb,c);
     return(0);
}
```

Then you could install this routine as the input character interceptor for *both* channels A and B:

```
btuchi(chana,&uchat);
btuchi(chanb,&uchat);
```

Now whatever character is typed by *either* the user on channel A *or* the user on channel B is immediately echoed to the screens of *both* users.

CAUTIONS

Your character handler routine (the rouadr parameter passed to btuchi(), above) must be coded to execute very fast, in order not to hinder operation of the Software Breakthrough. The Software Breakthrough has been coded in assembly language and honed into a lightning fast computing machine. That's why we can claim 256-user capability on a single PC. You have the opportunity to change all that by intercepting input characters with a slow clunky handler routine. In particular, your routine should only interact with variables and data structures in memory (you don't have enough time to even *think* about disk I/O!).

Since these routines may be called at the interrupt level, take care that any data structures used by both the rouadr character handler, and by any other of your routines, are not "skewed". This happens when your mainline (not interrupt generated) routine changes a data structure, and halfway through such a change, the interrupt-generated handler tries to use or change the same data structure. Remember that your character handler can be invoked *at any time*.

The routine identified as a character interceptor (by passing its address to btuchi()) should only be used for that purpose, and called under no other circumstances.

chiinp(), chiout(), chious(), and chiinj() must be called only by the routines that are used as input character interceptors, or real-time interrupt handlers... that is, whose address is passed as a parameter to btuchi() or bturti() (page 142).

The chiout() and chious() routines are limited by the size of the echo buffer: 255 bytes.

X.25

Note to experts about interaction between echo modes and the btuchi() routine on X.25 channels:

Conditions:

1) X.25 channel

2) In echo-plex mode

3) You are using btuchi(chan,custom) with a custom routine that returns values (intending them to be input *and* echoed) and those values are *not* the same as the values input to the custom routine.

Action:

call btuech(chan,2) when you install btuchi(chan,custom), and recover btuech(chan,whatever) when you uninstall btuchi(chan,NULL). See page 84 on the values for the btuech() echo parameter.

Reasoning: when you install the custom routine, you want the user's PAD to stop blindly echoing everything the user types, and start having the GSBL decide what gets echoed (via return value from your custom routine). That means that you do *not* want echo-plex... you want the GSBL to echo.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	all is well

btuclc

SUBROUTINE NAME

btuclc — clear command output buffer

SYNOPSIS

```
err=btuclc(chan);
int err;                    zero means OK
int chan;                   channel number
```

DESCRIPTION

btuclc() aborts any commands that may be in progress (refer to btucmd(), page 54), and clears the command buffer for the specified channel. A status 65 (refer to page 161) may be generated if a command was in progress.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	all is well

btucli

SUBROUTINE NAME

btucli — clear data input buffer

SYNOPSIS

```
err=btucli(chan);
int err;                      zero means OK
int chan;                     channel number
```

DESCRIPTION

Clears the data input buffer for the specified channel. Any queued input, even partial input strings or blocks, are completely cleared as though they were never received.

CAUTIONS

Calling this routine can cause inconsistencies between the status buffer contents and the input buffer contents. For example, if a CR-terminated string has been received but not yet processed at the time btucli() is called, the status of 3 (CR-TERMINATED INPUT DATA STRING AVAILABLE) will remain queued, but there will be no corresponding input data string available.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	all is well

btuclo

SUBROUTINE NAME

btuclo — clear data output buffer

SYNOPSIS

```
err=btuclo(chan);
int err;                  zero means OK
int chan;                 channel number
```

DESCRIPTION

btuclo() aborts any data output operation underway on the specified channel, clears its data output buffer, and eliminates the effects of XOFFs received, if any (see btuxnf(), page 193).

CAUTIONS

Any output data pending for the specified channel will be lost when this routine is called.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	all is well

btucls

SUBROUTINE NAME

btucls — clear status input buffer

SYNOPSIS

```
err=btucls(chan);
int err;                    zero means OK
int chan;                   channel number
```

DESCRIPTION

Provided only for completeness, this routine clears the highly sensitive status-input buffer, the key area upon which the sensing of modem condition depends. You should almost never need to use btucls().

CAUTIONS

Do not call this routine unless you are totally confident that this is really what you want to do, since it is potentially dangerous to arbitrarily clear out what might be critical modem status information, such as “lost carrier”.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	all is well

btucmd

SUBROUTINE NAME

btucmd — command channel

SYNOPSIS

```
err=btucmd(chan,cmdstg);
int err;                  zero means OK
int chan;                 channel number
char *cmdstg;             command string (ASCIIZ)
```

DESCRIPTION

This routine controls functions of the UART and (if used) the modem on a channel. For example, you could dial up Galacticomm's Demonstration System with the following C-language statement:

```
btucmd(chan,"WT13055837808M");
```

If the call goes through as expected, btusts(chan) will eventually return a value of 12 (for Hayes hardware), or 2 (for Xecom hardware), meaning that command execution has completed successfully (see page 5 for a discussion of hardware categories). Meanwhile your program can be off doing other things. Note that a status of 2 or 12 is created only when the end of the command string is reached, not on each command byte, and only when no other status from the modem is present after the last command byte in the string has been executed.

The syntax of the command string closely follows the modem command protocol developed by Xecom. However, many of these command codes can also be used on Hayes or UART category hardware. We have also added some command codes of our own which are hardware independent. The services accessed by btucmd() fall into six categories as shown in figure 4-1.

This table shows which commands are supported on which hardware:

	HAYES	XECOM	UART
Modem dial commands	all	all	none
Modem mode commands	most	all	none
Baud rate commands	all	most	all
Framing commands	all	all	all
Pause commands	all	all	all
ANSI commands	all	all	all
Diagnostic commands	none	all	none

Unsupported commands generate a status 13 on Hayes and UART category hardware, and a status 63 (a question mark) on Xecom category hardware (refer to btusts(), page 154).

In figure 4-1, Commands with an asterisk (*) in the Status column return information in the receive buffer. We recommend that you use btutrg() to select binary receive mode so that a status 4 is generated when these bytes are available.

Note 1: A status code of 13 from Hayes category hardware, or a status code of 63 (a question mark) from Xecom category hardware, means that the command is not supported on this hardware.

Note 2: On Hayes category hardware, commands which generate a status 3 return information in the receive buffer. The channel should be in the ASCII receive mode so that a status 3 indicates the availability of these bytes. This is the default mode, but if you should select binary receive mode using btutrg(), be sure to restore ASCII receive mode, also using btutrg(). In these cases, btusts() will return the status 3 *after* it returns the status 12.

Note 3: The baud rate, framing, pause, and ANSI commands are *soft* commands processed by the PC rather than by the modem hardware. If you use btucmd() with a string of commands, all of the soft commands will be executed first, and then all of the *hard* commands.

These soft commands are the exact set of commands that may be issued on UART category hardware.

Figure 4-1: Summary of Hayes and Xecom Commands

			HAYES	XECOM
	Command	Description	statuses	statuses
Modem Dial Commands	R	Rotary (pulse) dial mode	12	2
	T	Touch-tone dial mode	12	2
	1 2 3	\		
	4 5 6	\ Dial a digit	12	2
	7 8 9	/ (touch tone or rotary)		
	0	/		
	a	\		
	b	\ Dial a special digit	12	2
	c	/ (touch tone only)		
	* # d	/		
Modem Mode Commands	W	Wait for dial tone	12 3	2 BMIRT V
	A	Answer-carrier with no frills	12 3	2ITF
	^A	Answer-carrier with detections	12 3	2ITF1v
	O	Originate-carrier with no frills	12 3	2ITF
	M	Originate-carrier with detections	12 3	2ITFBRD V
	D	DTMF receive mode (touch tone input)	13	4*
	H	Hold (off-hook, modem disconnected)	13	2
Baud Rate Commands	^T	Set speed to 300 bps	12	2I
	^H	Set speed to 1200 bps	12	2I
	^F	Set speed to 2400 bps	12	?
Framing Commands	>	Set parity to "even"	12	2
	<	Set parity to "odd"	12	2
	=	Set parity to "none" (no parity bit)	12	2
	^S	Set character length to 7 bits	12	2
	^E	Set character length to 8 bits	12	2
	^N	Set stop bits to one	12	2
	^W	Set stop bits to two	12	2
Pause Commands	P	Pause 5 seconds	12	2
	p	Pause 2 seconds	12	2
	t	Pause 1/10 second	12	?
ANSI Commands	[	Enable ANSI graphics (default)	12	2
	]	Disable ANSI graphics	12	2
Diagnostic Commands	I	Identify modem version	13	4*
	i	Identify revision number	13	4*
	^M	Continuous monitor, return line freq	13	4*
	L	Line analyze	13	4*
	l (ell)	Return 1200-bps error statistics	13	4* I
	^X	Analog loopback in originate mode	13	
	^Y	Analog loopback in answer mode	13	
Invalid Command	(Anything else)		13	?

HAYES

On Hayes-protocol hardware, all modem dial commands, baud rate commands, framing commands, and pause commands are supported.

Most modem mode commands are supported. Some of these (^A and O) have reduced features over their Xecom-protocol counterparts. Others (D and H) are not supported on Hayes hardware.

None of the Diagnostic commands are supported on Hayes category hardware.

If you are exclusively using Hayes category hardware, you may want to use btuxmt() (the transmit routine, page 189) to handle the modem dial functions and modem mode functions, instead of btucmd().

Hayes-protocol modems have an AT command language for controlling the modem. See your modem manual for more details. The following table shows the modem mode commands that are translated into Hayes-protocol AT command codes by btucmd():

`btucmd()` command char	Hayes command char	Function
`^A` or `A`	`A`	Answer incoming call
`W`	`D`	Wait for dial-tone and dial a number
`M` or `O`	`O`	Go online (generate originate-carrier)

The btucmd() modem dial commands are translated directly into parameters of the Hayes-protocol D command. For this reason, when any of the following btucmd() commands are used on Hayes category hardware, they must be placed immediately following the btucmd() W command (wait for dial-tone), which is translated into the Hayes protocol D command (dial):

btucmd() command char	Hayes D command parameter	Function
R	P	Rotary (pulse) dial mode
T	T	Touch-tone dial mode
0 through 9	0 through 9	\
* #	* #	> Output DTMF tone
a b c d	a b c d	/

For example, the sample C-language statement mentioned above for dialing Galacticomm’s Demonstration System was:

```
btucmd(chan,"WT13055837808M");
```

For Hayes category hardware, this statement has almost the identical effect as:

```
btuxmt(chan,"ATDT13055837808O");
```

The major difference being that when the btucmd() directive has completed, a status code of 12 is generated (page 156), assuming nothing goes wrong with the call. Minor differences include the generation of a status 5 after btuxmt() (if btuoes() has enabled it, see page 133), and that btucmd() on Hayes category hardware uses the echo buffer for output and is subject to its limitation of 255 characters at a time.

On Hayes category hardware, some commands are handled by the Software Breakthrough, and others are passed directly to the modem hardware. We will call these *soft* and *hard* commands. The soft commands are the baud rate, framing, pause, and ANSI commands. The hard commands are modem mode and dial commands, and the diagnostic commands.

Soft commands are the only commands available on UART category hardware. On Hayes category hardware, if you have a btucmd() string with mixed soft and hard commands, then all of the soft commands are executed first, followed by all of the hard commands. For example, the command string:

```
btucmd(chan,"Ap");
```

will pause for 2 seconds (p to pause is a soft command), and then answer an incoming call (A to answer is a hard command).

However, if you use the following approach:

```
btucmd(chan,"A");
btucmd(chan,"p");
```

then the answer will precede the pause. See also page 191 for a description of the priority that transmitting takes over command execution.

Instead of the btucmd() baud rate commands, you may want to use btubrt() (page 34) to select the baud rate on Hayes and UART category hardware.

XECOM

All of the commands of btucmd() are supported on Xecom category hardware (Galacticomm Models 16 and 4), with the single exception of the ^F command (2400 baud).

The framing and pause commands provided by btucmd() apply to all categories of hardware. These functions can be obtained through no other Software Breakthrough library routine.

Be sure to review the **CAUTIONS** at the end of this section on btucmd() (page 78).

LAN

Here are the letter command codes that are valid for LAN channels:

		Page
L	Listen for connection (SPX only)	69
W	Prefix to dial-out command	73
0-9 A-F	24-digit network/node/socket address specified for dial-out command	73
M	Suffix to dial-out command	73
T	Terminate an SPX connection	72
G	Greet another GSBL and offer to handle its input preprocessing	67
P	5-second pause (uppercase P)	71
p	2-second pause (lowercase p)	71
V	Disable auto PAD echo programming	72
[	Enable ANSI-X3.64 output	77
]	Disable ANSI-X3.64 output	78

X.25

The following command codes may be used on X.25 channels:

		Page
A	Answer an incoming call	71
W	Prefix to dial-out command	75
0-9	Digits of address or user data field	75
A-F	Digits of user data field	75
,	Between caller and callee address	75
M	Suffix to an outgoing dial command	75
/	Prefix hexadecimal user data field	75
P	5-second pause (uppercase P)	71
p	2-second pause (lowercase p)	71
[	enable ANSI-X3.64 output	77
]	disable ANSI-X3.64 output	78

Astute readers will notice that A appears twice in the above list. A is a hexadecimal digit when, in a W command, it appears after a / and before the M. Otherwise, A is an answer command.

Command codes T and R are ignored on X.25 channels. All other codes will produce a status 23 (page 157), and the remainder of the command string will not be processed.

^A (CTRL+A)

ANSWER-CARRIER WITH DETECTIONS (TT,VOICE)

HAYES

This command acts just like command A (Answer-Carrier With No Frills) on Hayes hardware. See page 66 for details.

XECOM

This command puts "answer carrier" on the line, and will generate a status of 2 if "originate carrier" is heard within 17 seconds. It acts just like the regular A command, but adds extra monitoring capability — the answer sequence may be aborted by the caller in two ways. If the caller presses the DTMF "1" key, a status code of 49 (ASCII "1") is generated. If the caller speaks, a status code of 118 (ASCII "v") is generated. In either case the answer carrier tone is terminated.

^E (CTRL+E)

SET CHARACTER LENGTH TO 8 BITS

This mode can be used for binary data protocols such as XMODEM, etc. This command works on all hardware categories.

^F (CTRL+F)

SET SPEED TO 2400 BPS

HAYES UART

Sets the UART to 2400 baud. For Hayes category hardware, this command (or a call to btubrt()) must be issued after an answer command (command A, page 66) results in a CONNECT 2400 message.

^H (CTRL+H)

SET SPEED TO 1200 BPS

HAYES UART

Sets the UART to 1200 baud. For Hayes category hardware, this command (or a call to btubrt()) must be issued after an answer command (command A, page 66) results in a CONNECT 1200 message.

XECOM

For outgoing calls, sets the preferred speed of the modem to 1200 bits per second. For incoming calls, results in a status I (for Inappropriate) when a 300 baud connection is already established.

^M (CTRL+M)

CONTINUOUS MONITOR, RETURN LINE FREQ

XECOM

This command causes the modem to turn into a sort of spectrum analyzer, reporting back the frequency it hears 20 times a second. The frequency is reported as a data byte ranging in value from 0 to 255:

0	Quiet, no frequency at all
1 to 254	Frequency in tens of Hz (10 Hz to 2540 Hz)
255	Frequency greater than 2540 Hz

This function can only be terminated by a call to either bturst() (which resets the channel) or btuclc() (which clears the command buffer and aborts any command currently underway).

^N (CTRL+N)

ONE STOP BIT

UART

Sets the UART in this channel to use 1 stop bit. This mode is the default condition after bturst().

^S (CTRL+S)

SET CHARACTER LENGTH TO 7 BITS

This mode is the default upon channel reset. This command works on all hardware categories.

^T (CTRL+T)

SET SPEED TO 300 BPS

HAYES UART

Sets the UART to 300 baud. For Hayes category hardware, this command (or a call to btubrt()) must be issued after an answer command (command A, page 66) results in a CONNECT message.

XECOM

For outgoing calls, sets the “preferred speed” of the modem to 300 bits per second. For incoming calls, will result in a status code of I (Inappropriate) when a 1200 baud connection is already established. This is the default upon channel initialization or reset.

^W (CTRL+W)

TWO STOP BITS

UART

Sets the UART in this channel to use 2 stop bits.

^X (CTRL+X)

ANALOG LOOPBACK IN ORIGINATE MODE

XECOM

This command initiates a local loopback in the originate band. The filters for transmit and receive are set for the originate frequencies and are looped to each other. Transmitted data is routed through the full analog-digital path before being received back at the receiver port. May be used at any “preferred speed” to test all possible encode/decode components in a modem module. Once put in this mode, the channel must be reset (see bturst(), page 138) to get out of it.

^Y (CTRL+Y)

ANALOG LOOPBACK IN ANSWER MODE

XECOM

This command initiates a local loopback in the answer band. The filters for transmit and receive are set for the answer frequencies and are looped to each other. Transmitted data is routed through the full analog-digital path before being received back at the receiver port. May be used at any “preferred speed” to test all possible encode/decode components in a modem module. Once put in this mode, the channel must be reset (see bturst(), page 138) to get out of it.

0 to 9

DIAL 0 to 9 (ROTARY PULSE OR TOUCH TONE)

HAYES XECOM

The corresponding digit will be rotary dialed or touch-tone dialed depending on the current mode (previous R or T command). On Hayes category hardware, these codes must only be used after the W command (page 73).

*, #, a, b, c, d

DIAL SPECIAL TOUCH-TONES

The corresponding special touch-tone pair will be emitted (a through d are part of a “phantom column” in the Bell DTMF standard, but they don’t appear on any telephone we’ve ever seen).

HAYES

These codes must only be used after the W command (page 73).

XECOM

Any of these codes will generate a status of I (Inappropriate) if the current mode is rotary dial.

>

SET PARITY TO “EVEN”

Sets up to generate an even parity bit on every byte transmitted, and demand even parity on every byte received. If an odd parity byte is received, the results are controlled by btuerp() (the error pass-through routine, page 94), and by your input translation table, if in ASCII mode (btuxlt() controls the translation table, see page 185). Even parity is the default condition upon channel initialization or reset. This command works on all hardware categories.

<

SET PARITY TO “ODD”

Sets up to generate an odd parity bit on every byte transmitted, and demand odd parity on every byte received. If an even parity byte is received, the results are controlled by btuerp() (the error pass-through routine, page 88), and by your input translation table, if in ASCII mode (btuxlt() controls the translation table, see page 185). This command works on all hardware categories.

=

SET PARITY TO “NONE” (NO PARITY BIT)

Disables generation and checking of parity bits. This command works on all hardware categories.

A

ANSWER-CARRIER WITH NO FRILLS

HAYES

This command puts “answer carrier” on the line, and generates a status 3 if “originate carrier” is heard within 17 seconds. The status 3 indicates a received data string that you can get out of the receive buffer by using btuinp() (refer to page 108). Some examples of the string:

CONNECT	when 300 baud carrier has been detected
CONNECT 2400	when 2400 baud carrier has been detected
CONNECT <baud>	when other baud rates have been detected
CONNECT <baud><mode>	for special compression or correction modes

If no carrier is detected in 30 seconds, the following string usually appears in the receive buffer:

NO CARRIER	no originate carrier detected within 30 seconds

XECOM

This command puts answer carrier on the line, and will generate a status of 2 if originate carrier is heard within 17 seconds. Otherwise a status code is generated indicating: T (Timeout), or I (Inappropriate). This is the command you would normally use to pick up the phone when it rings (rather than the ^A command).

X.25

The A command will answer an incoming call on an X.25 line if issued immediately after receiving a status 3 with a RING message:

RING <caller> CALLING <callee>

Where <caller> is the decimal network address (if available) for the source of this call, and <callee> is your network address. The A command generates an immediate status 22.

If you set x25udt to 1, then btuinp() will return:

RING <caller> CALLING <callee> <user data field>

See page 32 for details on the limitations of this method.

D

DTMF RECEIVE MODE (ACCEPT TOUCH TONE INPUT)

XECOM

When placed in this mode, a modem will recognize incoming DTMF tone pairs as data input. The ASCII representations of 0-9, a-d, *, and # will be buffered as input data characters when they are detected. Note that DTMF information can be transmitted by issuing the normal digit dial commands. This mode is cancelled by a H, M, O, or A command.

G

GTC GREETING (LAN)

LAN

The Galacticomm Terminal Configuration protocol was designed for two computers that are connected via IPXV or SPX LAN channels. GTC allows one computer to preprocess the input of the other computer. The presumption is that both computers are using the GSBL to communicate.

For example, GTC can be used in a terminal emulation program talking to a Worldgroup server. The G command is executed in the terminal program once connection is established and says in effect: “Offer to preprocess the input of the server”. This includes buffering an input line with or without echo, input word wrap, etc. Executing the G command paves the way for status codes 40 - 44 to appear for the terminal program to handle (informing it of changing server input modes). For the Worldgroup server program all aspects of this exchange are handled by the GSBL and need no special handling by the server’s C program. The key point to remember here is that you only have to consider handling GTC if you execute the G command.

See page 224 for more on the GTC protocol.

H

HOLD (DISCONNECT MODEM, BUT REMAIN OFF-HOOK)

XECOM

Performs a “logical disconnect” of the modem from the phone line, but does not terminate the physical connection (i.e. the phone remains off-hook). This command can be used to quiet the line for voice or other use during a connection.

I

IDENTIFY MODEM VERSION

XECOM

Causes a single ASCII letter representing the version of the modem module to be generated as an input data character. Version codes are assigned consecutively, starting with A (65 decimal). Version number changes correspond to “significant unity upgrades”, whereas revision numbers (see i immediately following) represent incremental enhancements.

i

IDENTIFY REVISION NUMBER

XECOM

Causes a single ASCII digit representing the revision code of the modem module to be generated as an input data character. This will usually be of no concern to you, since only “version numbers” (see I immediately above) correspond to “significant unity upgrades”.

L

LINE ANALYZE

XECOM

This command acts the same as the M command but generates three special data input bytes if successful. The first byte represents the carrier frequency error, the second the S/N ratio, and the third the received carrier level. The calculations necessary for meaningful interpretation of this information are covered in Xecom data sheets.

L

LISTEN FOR SPX CONNECTION (LAN)

LAN

This command prepares the channel to accept a connection request from another party (presumably a terminal program like SPXTALK). The channel will remain in the listen-for-connection mode until a connection is established or until the channel is reset with bturst().

The listen command should only be issued on SPX channels that are in the idle state (see page 216). The L command will have no effect on an IPX channel.

l (LOWERCASE+L)

RETURN 1200-BPS ERROR STATISTICS

XECOM

This command may be used during a 1200 bps connection to check the phase demodulation statistics. An I (Inappropriate) status is generated if no 1200 bps connection exists. Otherwise two data bytes are queued: the first gives the average phase error of the signal, and the second gives the number of phase hits since the last request. The calculations necessary for extracting meaningful information from these values are covered in Xecom data sheets.

M

MONITOR LINE AND ORIGINATE WITH DETECTIONS

HAYES

The modem goes to the Online mode, waiting for answer carrier. If answer carrier is detected within 30 seconds, the following string appears in the receive buffer (refer to btuinp(), page 108):

CONNECT	when 300 baud carrier has been detected
CONNECT 1200	when 1200 baud carrier has been detected
CONNECT 2400	when 2400 baud carrier has been detected

If no answer carrier is detected within 30 seconds, the following string appears in the receive buffer:

NO CARRIER	no answer carrier detected within 30 seconds

XECOM

This command will generate a status of 2 if answer carrier is heard within 17 seconds. Otherwise a status code is generated indicating: B (Busy), R (Ringing), I (Inappropriate), T (Timeout), D (Dial Tone), or V (Voice). You should reissue the M command if you want to continue the attempt (e.g. for the first few rings).

LAN

See page 73 for using the M command on LAN channels.

O

ORIGINATE-CARRIER WITH NO FRILLS

HAYES

This command acts identically to the M command on Hayes category hardware. The modem goes to the Online mode, waiting for answer carrier. If answer carrier is detected within 30 seconds, the following string appears in the receive buffer (refer to btuinp(), page 108):

CONNECT	when 300 baud carrier has been detected
CONNECT 1200	when 1200 baud carrier has been detected
CONNECT 2400	when 2400 baud carrier has been detected

If no answer carrier is detected within 30 seconds, the following string appears in the receive buffer:

NO CARRIER	no answer carrier detected within 30 seconds

XECOM

This command will generate a status of 2 if answer carrier is heard within 17 seconds. Otherwise a status code is generated indicating: T (Timeout), or I (Inappropriate). This command is primarily useful when the physical connection is already established, and your program decides to begin using it for data communications. When making a new connection, use the M command (page 70).

P

PAUSE 5 SECONDS

The "long pause". Works on all hardware categories.

p

PAUSE 2 SECONDS

The "short pause". Works on all hardware categories.

R

ROTARY (PULSE) DIAL SUBSEQUENT DIGITS

HAYES

This is the default condition, upon initialization or reset. This command must follow a W command.

XECOM

Use this command to dial on a phone line which does not support touch-tone dialing.

T

TOUCH-TONE DIAL SUBSEQUENT DIGITS

HAYES

This command must follow a W command.

XECOM

This is the default, upon initialization or reset.

T

TERMINATE SPX CALL (LAN)

LAN

The terminate command will gracefully end an SPX session. A status 36 on the end that issued the T command indicates that the session was terminated properly. A status 31 on the other end (in the GSBL of a program to which your program is connected) indicates the session was terminated. Here's a diagram of how an SPX session is terminated with the T command:

```
Application 1 --> GSBL --/ /-- GSBL --> Application 2
                         \ \
         -- 'T' cmd -->  / /
                         \ \  -- status 31 -->
        <-- status 36 -- / /
```

The terminate command should only be issued on SPX channels that are in the connected state (see page 216). At any other time, the terminate command will have no effect. The T command will have no effect on an IPX channel.

t

WAIT 1/10 OF A SECOND

HAYES UART

Generates a status 12 after 0.1 seconds. Works only on 8250-type UARTs and modems or Model 2408 cards.

V

DISABLE AUTOMATIC PAD ECHO PROGRAMMING (X.25)

X.25

Normally, turning echo on and off with btuech(), or switching between binary and ASCII input modes with btutrg() would automatically cause programming of parameter 2 of the remote PAD. This command disables that feature until the next bturst().

W

WAIT FOR DIAL TONE (TO PLACE AN OUTGOING CALL)

HAYES

This command is usually accompanied by digit dial commands, such as:

```
btucmd(chan,"WT13055837808M");
```

(which would dial the Galacticomm Demo system from channel chan).

In this case, if connection is established within 30 seconds, a status 12 is generated, along with the following string in the receive buffer (refer to btuinp(), page 108):

CONNECT	when 300 baud carrier has been detected
CONNECT 1200	when 1200 baud carrier has been detected
CONNECT 2400	when 2400 baud carrier has been detected

If no answer carrier is detected within 30 seconds, the following string appears in the receive buffer:

NO CARRIER	no answer carrier detected within 30 seconds.

XECOM

This command will generate a status of 2 if a dial tone is sensed within 5 seconds. Otherwise a status code is generated indicating: B (Busy), M (Modem Carrier Sensed), I (Inappropriate), R (Ringing), T (Timeout), or V (Voice).

LAN Dial-out Command

W, digit, and M

LAN

This command is used to make an outgoing call on the network and establish connection with another address on the network. For SPX channels, a dial-out command means to establish a connection with a remote network/node/socket. You can have multiple SPX connections between the same pair of network/node/sockets. For IPX, a dial-out command specifies who we will talk to and who we will listen to. After an IPX dial-out command on a particular channel, all transmissions on the channel will go to the specified network address, and all packets received from that address (and directed to the same local socket number) will be routed to the input buffer of the channel.

On IPX Virtual circuits, the dial-out command may also be used to establish an incoming call. In the raw-packet mode, an IPX Virtual Circuit reports the complete IPX packet to you. Then you can look at the source network/node/socket of this packet to determine who is attempting to communicate with you (page 220). You can format the results of this report into a dial-out command to complete the connection.

You can issue dial-out commands on IPX channels whenever you want. When you do, all transmits and receives since the most recent call to btuscn() will correspond to the new network address. These transmits/receives will be processed/available after the next call to btuscn().

SPX channels are more complicated. The dial-out command should only be issued on SPX channels that are in the idle state. See page 216 for details.

If you use a dial-out command to specify a network that is not connected, then the GSBL may appear to hang for several seconds, and then issue a status 39 (page 158).

Unique Network Addresses for IPX Dial-outs

Among channels with the same local socket number, the network/node/socket of an IPX dial-out command should be unique. That is, once you specify a certain network/node/socket for the dial-out command on one channel, you should not specify the same network/node/socket for the dial-out command on any other channel that shares the same local socket.

For example, this is not a good idea because it uses the same local socket number, the same destination network/node/socket, twice:

```
btusdf(0,2,6,0x4007,2);
btucmd(0,"W000000010000C0A61018M");
btucmd(1,"W000000010000C0A61018M");
```

This is OK, however, because it uses different destination addresses:

```
btusdf(0,2,6,0x4007,2);
btucmd(0,"W000000010000C0A61018M");
btucmd(1,"W000000010000C0CE0018M");
```

This is also OK because it uses different local socket numbers:

```
btusdf(0,1,6,0x4007,2);
btusdf(0,1,6,0x4008,2);
btucmd(0,"W000000010000C0A61018M");
btucmd(1,"W000000010000C0A61018M");
```

The result of IPX dial-outs from two channels with the same local socket calling the same network address is that received packets may wind up in the input buffer of either channel, arbitrarily. Transmitted packets have no such ambiguity (although the other party may not be able to distinguish the originating channel).

Special Syntax of the Dial-out Command

An outgoing call may be specified by multiple btucmd() calls, as long as there are no other intervening calls to breakthrough routines. For example:

```
btucmd(chan,"W");
btucmd(chan,"00000001");
btucmd(chan,"0000C0A81018");
btucmd(chan,"4007");
btucmd(chan,"M");
```

will have the same results as:

```
btucmd(chan,"W000004010000C0A810184007M");
```

X.25 Dial-out Command:

W, digits, slash, hex digits, and M

X.25

To place an outgoing call on an X.25 channel, issue any of the following command string formats using btucmd():

```
W<caller>,<callee>/<user data field>M
W<callee>/<user data field>M
W<caller>,<callee>M
W<callee>M
```

You can use T (touch tone) or R (pulse dialing) in the above commands for compatibility with Xecom channels. For X.25 channels, they have no effect. For example:

```
btucmd(chan,"WT100657,399613M");
```

In this case, you are seeking network address 399613, while identifying yourself as network address 100657.

Note that an outgoing call may be specified in segments as long as there are no intervening calls to other GSBL routines. For example:

```
btucmd(chan,"W1234,");
btucmd(chan,"6789M");
```

will have the same results as:

```
btucmd(chan,"W1234,6789M");
```

The user data field of an outgoing call request packet is normally empty. To put data into the user data field, code it in hexadecimal after the / (slash) command. For example:

```
btucmd(chan,"W1234,6789/41423031M");
```

That example will put the 4-byte ASCII string AB01 into the user data field. This data need not be ASCII. To code three zero bytes and a byte with value 01 hex:

```
btucmd(chan,"W1234,6789/00000001M");
```

Important note: The outgoing call, if completed, may not actually be made over the same channel that you specify in btucmd(). The PC XNet card driver will decide the actual virtual circuit to use for the call.

Specifying the channel for btucmd() with a dial-out command does serve the purpose of identifying the card and line to use. The PC XNet card usually puts the call on the highest unused channel that has been configured (unless the line is configured as DCE, in which case outgoing calls go over the lowest available channel).

When an outgoing call has been confirmed, status code 77 (ASCII M) is generated on the channel on which the call has been established. If an outgoing call fails, either: status code 66 (ASCII B) is generated if there is some local transmission error encountered when transmitting the call request packet (like transmit window full) or status code 21 (lost carrier) is generated if the network refuses to connect your call, perhaps because the network address you specified is busy, or does not exist, or if the service you're dialing refuses your call for some reason (see page 94 about obtaining cause and diagnostic fields).

To terminate a call, bturst() disconnects the virtual circuit. When a call gets terminated, either by the network or by the remote user, status code 21 is generated on the channel.

[

ENABLE ANSI GRAPHICS (default)

This command affects the operation of btuxmt() when transmitting the following construct:

ESC [<for ANSI users> | <for non-ANSI users>]

If ANSI graphics have been enabled for a channel by this command, (or by default), then the string <for ANSI users> from the above construct *will* be transmitted and the string <for non-ANSI users> will *not* be transmitted.

ANSI graphics are automatically enabled by bturst(). See page 189 for more information on ANSI graphics directives.

]

DISABLE ANSI GRAPHICS

This command affects the operation of btuxmt() when transmitting the following construct:

ESC [<for ANSI users> | <for non-ANSI users>]

If ANSI graphics have been disabled for a channel by this command, then the string <for ANSI users> from the above construct will *not* be transmitted and the string <for non-ANSI users> *will* be transmitted. Also, ANSI graphics directives embedded in the text passed to btuxmt() will not be transmitted to that channel. See page 189 for more information on these directives. ANSI graphics are automatically enabled by bturst().

CAUTIONS

If data input or output is in progress when this call is made, the data operation will be allowed to complete before the command string is executed. See page 191 on the priority of transmission over commands.

The framing commands must be reissued each time you reset a channel (by bturst(), page 138).

The command buffer can hold 62 characters at a time.

HAYES

To use btucmd() on Hayes category hardware, the modem must be in command mode.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	all is well

btucpc

SUBROUTINE NAME

btucpc — sets the clear pause-counter character
(puts off screen pauses when in output stream)

SYNOPSIS

err=btucpc(chan,cpchar);	
int err;	zero means OK
int chan;	channel number
char cpchar;	clear pause-counter character (or 0 to disable)

DESCRIPTION

When the cpchar character is discovered in the output stream (btuxmt() page 189 or btuxmn()page 187), the internal line counter is reset to 0. This line counter is used to determine when to display the pause message set by btuhpk() (page 98). The character itself is never actually output. Use cpchar=0 to disable this feature.

In Worldgroup, this function is used to prevent screen pauses by inserting the CTRL+S character at strategic points in certain text blocks.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	all is well

btudef

SUBROUTINE NAME

btudef — define channels

SYNOPSIS

err=btudef(schan,sport,n);

int err;	zero means OK
int schan;	starting channel number
int sport;	starting port address
int n;	number of channels

DESCRIPTION

After you initialize the Software Breakthrough as a whole using btusiz() (or btulsz()) and btuitz(), you use btudef() to initialize individual channel groups. You will call btudef() once for each device that you have installed in your system. See also about btusdf() (page 147) for more alternative devices.

When you called btusiz() or btulsz(), you specified the total number of channels in the first parameter nchan. btuitz() initialized the data structures for that many channels, numbered 0 to nchan-1. Now you will use btudef() to associate those channels with actual hardware. The following example in C illustrates a typical initialization sequence for one each of the Galacticomm Breakthrough Models 16, 4, and 2408 and a Hayes-compatible modem on COM1:

```
btuitz(malloc(btulsz(40,128,1024)));
btudef(0,0x2F0,16);      /* Define Model   16 at 2F0 (hex)
*/
btudef(16,0x2F2,4);      /* Define Model    4 at 2F2 (hex)
*/
btudef(20,0x2F4,8);      /* Define Model 2408 at 2F4 (hex)
*/
btudef(28,0x2F8,1);      /* Define 1 Hayes modem
*/
                         /* on a COM2 port at 2F8 (hex)
*/
btudef(29,0x3F8,8);      /* Define 8 Hayes modems in a
*/
                         /* GalactiBox, each at the COM1
```

```
*/
                              /* port address of 3F8 (hex)
*/
```

For Galacticomm Breakthrough cards, the sport parameter identifies the I/O base address for the card, as set on the card’s DIP switches. Refer to the Installation Manual for your particular model.

For hardware on the standard serial ports COM1 and COM2, you should use the following values for the sport parameter:

COM1	0x3F8	COM3	0x3E8
COM2	0x2F8	COM4	0x2E8

btudef() can discriminate between the three categories of hardware: Hayes, UART, and Xecom (see page 5). If it fails to find a Galacticomm Breakthrough Model 16 or 4 at the base address sport, it will look for a Model 2408 card at the same address. Failing that, it will look for an 8250-type UART (which is used in both Hayes and UART category hardware).

HAYES UART

See the discussion of bturst() (page 138) for the proper sequence of initializing a Hayes or UART channel.

CAUTIONS

The return value from btudef() only indicates whether the mechanics of the calling sequence are correct, not whether or not the modem or UART hardware is working.

If you are changing the maximum data rate using btumxs() (see page 129), you must do that before calling btudef().

XECOM

If there is no hardware at the specified address, the channels specified will all have statuses of -10 (see btusts(), page 154). If an underpopulated card is installed (for example, a Model 16 with 8 modems), then calls to btudef() on the vacant modem slots will also generate a single status code -10 on those channels.

HAYES UART

If there is no hardware at the specified address, the channels specified will all have statuses of -10 (see btusts(), page 154). If an underpopulated card is installed (for example, a Model 2408 with 4 modems), then calls to btudef() on the vacant modem positions will *not* generate statuses of -10. Those channels will appear to the software to exist, but they will never be connected to anything. That's because the 2408 card has an 8-channel UART talking to up to 8 modems. Whether the modems are installed or not, the presence of the UART is all that is sought by the reset operation.

If you have purchased the N-channel version of the Software Breakthrough, then you may only use channels 0 through N-1 to talk to real hardware. If you attempt to define channels N or higher, then these channels will each have a single status code of -10, as if there were no hardware on these channels. You may still use these channels for local emulation (described on page 126).

LAN X.25

Don't use btudef() on LAN or X.25 channels. Use btusdf() instead (page 147).

RETURNS

-11 channel number(s) out of range:
the specified range of channels (schan to schan+n-1)
is not within the inclusive range 0 to nchan-1,
where nchan has been defined
in btusiz() (page 152) or btulsz() (page 117)

0 the specified channels were defined successfully

btueba

SUBROUTINE NAME

btueba — echo buffer space available, in bytes

SYNOPSIS

nbytes=btueba(chan);

int chan;	channel number
int nbytes;	room in the echo buffer for bytes to echo to the user's terminal

DESCRIPTION

This routine returns a value indicating the number of bytes that the echo buffer for this channel can handle before overflowing.

The echo buffer can hold up to 255 bytes. When the echo buffer overflows, a status 252 is generated (page 163).

RETURNS

0	echo buffer is full
1-254	echo buffer is between full and empty
255	echo buffer is empty

btuech

SUBROUTINE NAME

btuech — set echo on/off

SYNOPSIS

```
err=btuech(chan,mode);
int err;                   zero means OK
int chan;                  channel number
int mode;                  0 = disable echo
                           1 = enable echo (echo-plex on X.25 channels)
                           2 = enable echo (GSBL echo on X.25 channels)
```

DESCRIPTION

This routine allows input echo to be enabled or disabled while in the ASCII input mode (page 12). It defaults to the enabled state upon channel initialization or reset.

There are two typical uses for this routine:

The first is for hiding passwords: you can turn off the echo on a channel just as the user is prompted for his password, then turn it back on again once the password has been entered. In this way, prying eyes in the user's vicinity will not see his password displayed on his screen.

The second use of btuech() is for half-duplex communications, in which the user's terminal already echoes each keystroke that he types (therefore your system should not echo). In this case, you would simply turn off the echo as soon as a connection is established, and leave it off for the duration (you will need to disable the echo after each reset of the channel with bturst(), page 138).

X.25

Echo-plex means that the PAD (packet assembler/ disassembler) on the user's end of the X.25 network does the echoing of characters. The GSBL programs the PAD to do this using the "Q" bit, per recommendations X.29 and X.3 (parameter 2). GSBL echo means that the GSBL echoes each input character, just as it does with modems and serial ports.

On a non-X.25 channel, btuech(chan,1) has the same effect as btuech(chan,2). On an X.25 channel, the call to btuech() controls both the echo of input from the GSBL and the echo of characters by the user's PAD:

Value of `mode` parameter in call to `btuech()`	Local echo from the GSBL?	X.3 pad parameter number 2 setting	Echo mode
0	OFF	0 (echo off)	Echo off
1	OFF	1 (echo on)	Echo-plex
2	ON	0 (echo off)	GSBL echo

Echo-plex can be more economical if your packet-switching network charges for data traffic based upon the number of packets input and output. Echo-plex saves the extra packet per user keystroke that would result from the GSBL echoing each keystroke. However, there are a few minor drawbacks to echo-plex for X.25 channels:

- The input word-wrap feature enabled by btumil() will not work well on an X.25 channel in echo-plex mode. The GSBL will be splitting lines on word boundaries, but not displaying the effects on the user's screen as it would in GSBL echo mode, or as it would on a non-X.25 channels.
- The maximum line length set by btumil() will not automatically disable echo when the limit is reached. The user's PAD will continue to echo even after the GSBL stops accepting characters.
- The user BACKSPACE key erases input keystrokes fine, but does not stop when the beginning of the input line is reached, and may erase a preceding prompt.
- If a user types during output to his screen ("type-ahead"), then the echo is not suspended until the next prompt (as it is with GSBL echo mode or with non-X.25 channels), but is mixed in with the output.

The monitored channel, when in echo-plex mode, specially handles monitored input and output to most closely resemble the users screen, including the above mentioned drawbacks.

To achieve this:

- Each monitored keystroke (btumks() or btumks2()) is immediately echoed to the monitor screen buffer (for access by btumds() or btumds2()) and to the user's screen (via the GSBL's echo buffer).
- Each received character is immediately echoed to the monitor screen buffer (for btumds() or btumds2()).
- Carriage returns echoed to the monitor screen buffer are automatically appended with line feeds.

Note: Echo-plex is automatically suspended during binary input (when btutrg(chan,nonzero) is in effect) and re-enabled when ASCII input resumes (btutrg(chan,0)).

See also page 44 regarding echo considerations for your custom character interceptor routine, specified by btuchi().

CAUTIONS

Half-duplex communications modes are not recommended for reasons of user-friendliness, flexibility, and standardization.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	all is well

btuend

SUBROUTINE NAME

btuend — shut down the Software Breakthrough

SYNOPSIS

btuend();

DESCRIPTION

Prepare the PC for return to DOS. This routine must be called as part of your exit cleanup procedure.

CAUTIONS

If your program has called btuitz() (initialization, page 112), you must call btuend() before your program returns to DOS. The Software Breakthrough alters certain PC hardware settings for its own ends, and these must be restored, or else the operating system will become very confused.

Only call btuend() upon exiting your program altogether. The Software Breakthrough routines will no longer function once you have called btuend().

btuend() does not hang up your phone lines. You should call bturst() for every channel you have defined before calling btuend(). If you do not, your phone lines may remain off-hook with the carrier signal on, appearing to any users online at the time that your system has “locked up”. Better to let them think that you are rude than that your system is malfunctioning.

RETURNS

Nothing.

btuerp

SUBROUTINE NAME

btuerp — pass/block input bytes with errors

SYNOPSIS

err=btuerp(chan,onoff);

int err;	zero means OK
int chan;	channel number
int onoff;	1 = accept characters with PE/FE/OE errors, setting the high-order bit of each (default) 0 = ignore characters with PE/FE/OE errors

DESCRIPTION

This feature gives you control over the handling of input characters received with parity errors, framing errors, or overrun errors.

If error-passthru is enabled (onoff = 1), then any character received with one of these errors will have its high-order bit set (i.e. if not already in the range from 128 to 255 it will be brought there by adding 128 to it). Then this new value is used as an index into the global input-character translation table (see btuxlt(), page 184).

If error-passthru is disabled (onoff = 0), then any character received with one of these errors will be ignored.

By default, this feature is enabled (onoff = 1). With the default input character translation table (page 185), this means that 7-bit ASCII characters with parity, framing, or overrun errors are received as if they had nothing wrong with them: their high bits are set, but the translate table effectively strips them off.

CAUTIONS

When running protocols requiring 8-bit data, such as XMODEM, you should probably disable error-passthru because otherwise you will not be able to distinguish between valid received characters with their high-order bits set and invalidly received characters. Anyway, during XMODEM input, parity errors are not possible, framing errors are rare, and overrun errors are unlikely.

RETURNS

-10 channel is not defined (see btudef(), page 80)

-11 channel number is out of range
(see btusiz(), page 152 or btulsz(), page 117)

0 all is well

btuffo

SUBROUTINE NAME

btuffo — enable receiver FIFO on 16550 UART

SYNOPSIS

err=btuffo(chan,onoff);	
int err;	0 is ok
int chan;	channel number
int onoff;	1=enable 16-byte FIFO’s
	0=disable (for exact 16450 compatibility)

DESCRIPTION

HAYES UART

This routine enables the 16-byte FIFOs on the 16550 UART. This can be used to avoid losing received characters. There are many diverse reasons why incoming characters could be lost. If any code, such as disk-caching software, keeps interrupts disabled for too long, for example, then the resulting jitter in sampling can cause characters to “fall between the cracks”. If this happens, the btuerp() routine (page 88) can allow overrun errors to be reported as data bytes with the high bit set (and then possibly stripped by the translate table, page 185), resulting in double characters (for example frog would turn into frrg).

The 16550 UART is a pin-compatible replacement for the 16450 UART (itself compatible with the original 8250 UART). UART stands for Universal Asynchronous Receiver Transmitter. It is the heart of almost all serial devices on the IBM PC and compatibles, and is also used in most modems, internal and external.

The 16550 UART however, has 16-byte deep first-in first-out queues, one in the receiver circuit and one in the transmitter. That means that

(1) the GSBL is almost certain not to miss any incoming characters, and

(2) the port can be polled less often (see page 130).

To enable this FIFO feature, call btuffo with a 1 for the onoff parameter.

HAYES UART

btuffo() has no effect on 8250 or 16450 UART. It will return -18 to let you know they're not 16550's.

XECOM LAN X.25

btuffo() has no effect on Xecom, LAN, or X.25 category hardware (it always returns 0).

CAUTIONS

Calling btuffo() on a channel with a 16550 UART can cause you to lose any characters that are in the FIFO transmitter or receiver circuitry at the moment that you call btuffo().

On a non-hardware channel that was defined using btudef() (or btusdf() with chtype=0), btuffo() will always return 0. On a non-hardware channel that was defined using btusdf() with chtype=3 (8250), the btuffo() return value is undefined. See page 147 for more about btusdf(), or page 80 for btudef().

RETURNS

-10 channel is not defined (see btudef(), page 80)
-11 channel number is out of range
(see btusiz(), page 152 or btulsz(), page 117)
-18 channel has an 8250 or 16450 UART on it
0 either the channel does not have an 8250-type UART on it, or it has a 16550 type UART on it, and the FIFO mode has been set

btuhcr

SUBROUTINE NAME

btuhcr — set the hard-CR character (for output word wrap)

SYNOPSIS

err=btuhcr(chan,hardcr);	
int err;	zero means OK
int chan;	channel number
char hardcr;	hard-CR, translated to 0DH on output

DESCRIPTION

The *hard* carriage return is an output character that is unconditionally converted into an ASCII CR (carriage return). This occurs during ASCII output mode, when output word wrap is in effect. ASCII output mode, discussed on page 15, is performed using btuxmt() (page 189). Output word wrap is controlled by btutsw() (page 172).

The hard carriage return defaults to ASCII CR (13 decimal) when a channel is initialized or reset, which is to say that CRs are passed through unchanged.

By default, all CRs are also appended with LFs on output. This feature is controlled by btulfd() (page 114).

You can think of a hard CR as representing “end of paragraph” if you have output word wrap enabled. Usually the default value of 13 will be fine, but there may be cases in which you want to use some printable character, such as @ or < to indicate paragraph boundaries.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	all is well

btuhdr

SUBROUTINE NAME

btuhdr — capture information on X.25 or LAN channel

SYNOPSIS

```
err=btuhdr(chan,nbytes,buffer);
int err;                 zero means OK
int chan;                channel number
int nbytes;              number of bytes to capture
char *buffer;            where to put them
```

DESCRIPTION

This routine can be used to capture the contents of special internal data structures for the channel.

The nbytes parameter should be even (if odd, btuhdr() will only copy nbytes-1 bytes).

LAN

Here are some useful values of nbytes for LAN channels:

46	for most recent listen ECB
76	for most recent listen ECB + listen IPX header
88	for most recent listen ECB + listen SPX header
176	for most recent listen ECB + listen SPX header + send ECB + send SPX header

ECBs used by the GSBL

The ECBs (Event Control Blocks) used by the GSBL are 46 bytes long. This includes the 42-byte structure defined by Novell for an ECB with one fragment descriptor, plus 4 bytes used internally by the GSBL.

The contents of this ECB can be found in the ecbgsbl structure in IPX.H:

```
struct ecbgsbl {        /* Event Control Block (for use with GSBL btuhdr()) */
     void *link;
     void (*esr)();                                /* event service routine */
     char inuse;                       /* 0=not in use FE=listening FF=sending */
     char complt;               /* 0=good, nonzero=command-specific error code */
     int socket;                                   /* sending socket (hi-lo) */
     char ipxwsp[4];                                        /* IPX workspace */
     char drvwsp[12];                                    /* driver workspace */
     char immnod[6];                               /* immediate node address */
     int frgcnt;                                   /* fragment count (lo-hi) */
     void *frgadr;                              /* address of first fragment */
     int frgsiz;                           /* size of first fragment (lo-hi) */
     int prtseg;               /* protected mode segment (GSBL-specific usage) */
     int chanx2;                  /* channel number * 2 (GSBL-specific usage) */
};
```

Local Network/Node Address

After calling btusdf() for a channel group, btuhdr() can be used to find the local network/node address of the machine on which the GSBL is running. It will reside in the dstnet and dstnod fields of the listen IPX header buffer. This occurs even before any packets are received, and offers your program the first opportunity to determine your local node address (if your computer never logs into a file server, this information might not be available for the first minute or so of operation).

Note that the IPX/SPX header part of this information is redundant in raw packet mode – you always get the entire packet in the input buffer.

X.25

Here are some useful values for nbytes when calling btuhdr() on X.25 channels:

2 for cause and diagnostic of last clear packet (see about status 21 on page 156)

52 for cause and diagnostic, plus information about the last received X.29 string (see about status 24 on page 157)

Here is the full 52-byte structure for the data captured by btuhdr() on an X.25 channel:

```
struct x25hdr {        /* X.25 info from btuhdr()        */
     char cause;       /* clear packet cause             */
     char diag;        /* clear packet diagnostic        */
     char x29num;      /* number of parameters in x3list */
     char x29flg;      /* flags, see below               */
     struct x3list {   /* array of up to 24 X.3 pairs:   */
          char par;    /*      parameter number          */
          char val;    /*      value                     */
     } x3list[24];
};
                       /* Masks for x29flg flag bits:    */
#define X3SET  0x02    /* request to set X.3 values      */
#define X3READ 0x04    /* request to read X.3 values     */
#define X3NEW  0x80    /* new X.3 message has been rec'd */
#define X3OVF  0x40    /* more than 24 parameters rec'd  */
```

Each incoming clear packet generates a status 21 (page 156), and comes with a cause and a diagnostic field. As you can see above, you can obtain these values with btuhdr().

Calling btuhdr() clears the x29flg&X3NEW flag. Each received X.29 string sets X3NEW. So, if the X3NEW flag is set, then a new X.29 string has been received since you last called btuhdr(). Incoming X.29 strings are also heralded by a status 24 (page 157).

All other flag bits, the x29num field, and the x3list array are from the most recently received X.29 string. The X3OVF flag indicates that more than 24 parameter/value pairs were in the last incoming X.29 string.

An incoming X.29 string means that the other DTE on the X.25 network is trying to query (X3READ), set (X3SET), or both set and query (X3READ+X3SET) your X.3 parameters. Obviously this DTE thinks it is talking to a PAD. To respond to a query as if you were a PAD, you can call btux29() (page 179) with a message code of 0 (meaning query-reply).

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range

	(see btusiz(), page 152 or btulsz(), page 117)
0	all is well

btuhit

SUBROUTINE NAME

btuhit — hook into a COM port interrupt
and use it to invoke channel servicing

SYNOPSIS

err=btuhit(irqno);	
int err;	zero means OK
int irqno;	2-7 IRQ number to intercept

DESCRIPTION

The GSBL is normally based on timed polling of all ports. This function can be used to make the GSBL instead take its cue from the UART interrupts of a very limited number of serial ports (typically 1 or 2, but possibly up to 6).

The most common practical values for irqno will be 3 and/or 4. You can call btuhit() on as many of the IRQ numbers 2-7 as you wish, but only once per IRQ number:

```
btuhit(3);
btuhit(4);
```

Normally you would use this feature in a multitasking environment such as Windows or OS/2.

CAUTIONS

btuhit() can only be used when the GSBL has been initialized with btuitm() (page 111), not with btuitz() (page 112).

Only one port per IRQ line can be serviced using this scheme.

RETURNS

-20	irqno is neither 2, 3, 4, 5, 6, nor 7, or btuitm() was not used to initialize (see page 111)
0	all is well

btuhpk

SUBROUTINE NAME

btuhpk Handle keystrokes during screen-pause mode

SYNOPSIS

```
err=btuhpk(chan,hpkrou);
int err;                 zero means OK
int chan;                channel number
int (*hpkrou)();         pointer to function to be called

rc=(*hpkrou)(chan,c);
                         character interceptor routine during screen-pause mode
char rc;                 0=ignore
                         1=next page
                         2=continue nonstop
int chan;                channel number
char c;                  character received
```

DESCRIPTION

First, a channel goes into screen-pause mode when:

1. More than N lines of text have been output (see the cnt parameter of the btuxnf(chan,xon,-xoff,cnt,stg) form of btuxnf() on page 193).
2. A special pause-character defined by btupbc() has been transmitted (page 135) and printable output has been transmitted to the user since the last time he pressed ENTER.
3. A clear-screen code — either an ASCII formfeed (CTRL+L, 12 decimal) or an ANSI clear-screen sequence (ESC [2J) — can also trigger a pause because btuxmt() will insert the btupbc() pause character immediately before it.

After a channel goes into screen-pause mode, each character received triggers a call to the hpkrou routine, which may be coded in C. In other words: you will use btuhpk() to identify that the hpkrou routine will handle each character received on any channel during screen-pause mode.

There are several functions that can be implemented by the hpkrou routine, based on the keystroke(s) received during screen-pause mode.

You can compose a very short (e.g. 40-character) menu of these functions in the stg message of the -xoff form of btuxnf(). Within the hpkrou routine:

1. To handle a “continue” key, just return 1.
2. To handle a “go nonstop” key, return 2.
3. To handle an “abort” key, you must inform your mainline program. For example, inject a status 7 (this is what happens in Worldgroup) and have the mainline program handle the status by clearing output using btuclo(), and changing states (to stop stuffing the output buffer).

The channel remains in screen-pause mode when you return 0. Otherwise, screen-pause mode ends and output resumes.

The character handed off to the hpkrou routine is not automatically echoed, but you could use chiout() to echo it if you wish.

CAUTIONS

See also page 48 for cautions relating to interrupt-called functions. Design of the hpkrou routine is fraught with pitfalls. Don’t design lengthy complex code to run at interrupt level. Don’t invoke DOS (because it is not reentrant). You may use the special chixxx() routines (page 47), but not any other Breakthrough Library routines. Beware of data skewing, since a real-time interrupt may occur between any two machine-language instructions of your mainline code.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	all is well

btuhwh

SUBROUTINE NAME

btuhwh — enable hardware handshaking using RTS/CTS

SYNOPSIS

```
err=btuhwh(chan,inpcut);
int err;                    zero means OK
int chan;                   channel number
char inpcut;                input buffer byte count cutoff point
                            (or 0 to disable handshaking)
```

DESCRIPTION

By default, hardware handshaking is disabled to permit simple serial cables (ground, receive, transmit, carrier-detect, and DTR). With high-speed modems, though, hardware flow may be necessary. The btuhwh() function, when called with a nonzero inpcut parameter has two effects:

1. A high-to-low transition on the CTS (clear-to-send) input signal will inhibit transmission of output data until CTS goes high again.
2. The RTS (request-to-send) output signal will be asserted only when there are less than inpcut bytes waiting in the channel's input buffer. This use of RTS is not RS232-C standard, but complies with the de facto standard of hardware manufacturers. The meaning is essentially "ready to receive".

These features allow effortless (non-software) throttling of data flow in both directions. Calling btuhwh() with inpcut equal to zero restores the default condition: CTS is ignored, and RTS is always active.

CAUTIONS

The sensing of the falling edge of CTS means that if CTS is always inactive, then there is no throttling of the outgoing data.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range

(see btusiz(), page 152 or btulsz(), page 117)

0 all is well

btuibw

SUBROUTINE NAME

btuibw — Input Bytes Waiting: report the number of bytes received
and waiting in the input buffer

SYNOPSIS

```
inpbcw=btuibw(chan);
int inpbcw;                    input bytes waiting, or error code
int chan;                      channel number
```

DESCRIPTION

This routine simply returns the count of bytes currently present in the channel's input data buffer. This can be used to keep track of a block-oriented input process, or to detect user keystrokes without removing them from the input buffer. The total capacity of the input buffer is:

Total capacity = isiz - 1

where isiz is the parameter passed to btusiz() (page 152) or btulsz() (page 117). Therefore the current capacity of the input buffer is:

Current capacity = isiz - 1 - btuibw(chan)

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	the channel number is OK, but that channel's input buffer is empty
N>0	this is the number of bytes waiting in the channel's input buffer

btuica

SUBROUTINE NAME

btuica — input from a channel: reading in whatever bytes
are available, up to a limit

SYNOPSIS

```
nbytes=btuica(chan,inbuff,siz);
int nbytes;                    bytes transferred (negative if error)
int chan;                      channel number
char *inbuff;                  pointer to buffer for input bytes
int siz;                       maximum number of bytes to get
```

DESCRIPTION

Only use btuica() when you have already prepared for the binary input method by calling btutrg() with nbyt > 0 (see pages 167 and 12). Use btuinp() instead (page 108) for the ASCII input method.

This routine copies whatever bytes have been received so far into a location you specify with the inbuff and siz paramters. The siz parameter specifies the maximum number of bytes that btuica() will put at inbuff. If more are available, they'll be left in the input buffer.

The key difference between this routine and btuict() is that btuict() always takes a prearranged number of bytes from the input buffer (prearranged by btutrg(), page 167) or it will take none at all. btuica() always reads in as many as possible, up to the siz limit.

This is the preferred method for binary input. Your application should use it to process incoming data as a stream of bytes.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	the channel number is OK,

but that channel’s input buffer is empty

N>0 this is the number of bytes actually moved
from the channel’s input buffer to inbuff

btuict

SUBROUTINE NAME

btuict — input from a channel: by the byte count prearranged
with btutrg()

SYNOPSIS

nbytes=btuict(chan,inbuff);	
int nbytes;	bytes transferred (negative if error)
int chan;	channel number
char *inbuff;	pointer to buffer for input bytes

DESCRIPTION

Only use btuict() when you have already prepared for the binary input method by calling btutrg() with nbyt > 0 (see pages 167 and 12). Use btuinp() instead (page 108) for the ASCII input method.

btuict() checks to see if at least nbyt (the trigger number) bytes have been received in the input data buffer and, if so, transfers them to your buffer whose starting address you've supplied in inbuff. This data is flushed from the input buffer. If not enough bytes have been received, btuict() transfers as many as it can, and does not flush any of them from the input buffer. In this case, the global variable ictact can be used to find out how many bytes were transferred (see page 26).

The key differences between this routine and btuinp() are:

1. btuict() reads in a count of characters (prespecified by btutrg()), whereas btuinp() reads in a variable number of characters ending in CR.
2. The characters returned by btuict() are arbitrary binary data, so they may include embedded zeros, or any other 8-bit value.
3. btuict() does not terminate the returned buffer contents with a zero byte, as btuinp() does.
4. btuict() performs no translation or special handling of the bytes it receives. See page 12 for more on ASCII versus binary input methods.
5. btuict() sets global variable ictact, indicating number of bytes transferred.

CAUTIONS

Normally, you should call this routine only after btusts() has returned a status of 4 (BYTE-COUNT-TRIGGERED INPUT DATA AVAILABLE, page 155) for the channel.

Do not call btuict() when in the ASCII input mode.

RETURNS

-10 channel is not defined (see btudef(), page 80)

-11 channel number is out of range
(see btusiz(), page 152 or btulsz(), page 117)

-2 insufficient bytes available in input buffer

N>0 length of returned input block (will be the same
as the nbyt value passed to btutrg()
when nbyt > 0, page 167)

The sequence of bytes at address inbuff:

> the input block (the length of which you have already specified by calling btutrg()). This string is not 0-terminated (as is the output of btuinp()).

btuinj

SUBROUTINE NAME

btuinj — “inject” a status code into a channel

SYNOPSIS

```
err=btuinj(chan,status);
int err;                    zero means OK
int chan;                   channel number
char status;                status to be injected
```

DESCRIPTION

This routine simulates a status condition for a given channel. Each channel has a status queue that can accumulate multiple status codes if necessary. Those status codes are accessed by the main program one at a time using the btusts() routine (page 154). In some cases, it is handy to be able to put status codes directly into that queue. Then btusts() will return that code (after it returns any others that may be pending).

Status codes 200-249 are nominally reserved for you to define for special application-specific purposes and insert in the status stream using btuinj().

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	all is well

btuinp

SUBROUTINE NAME

btuinp — input from channel (ASCIIZ string)

SYNOPSIS

len=btuinp(chan,inbuff);

int len;	returned string length or error code
int chan;	channel number
char *inbuff;	pointer to buffer for input string (the line terminator is replaced with a 0-byte)

DESCRIPTION

btuinp() transfers a complete input data line into your buffer. This routine should normally be called only when btusts() returns status 3 (CR-TERMINATED INPUT DATA STRING AVAILABLE). When you call btuinp() after a status 3 has been detected, then the received line is copied into your buffer (inbuff) and removed from the channel's input buffer. The carriage return that terminated the input line is replaced with a single byte of 0 value in your buffer. In this case the len return of btuinp() will always be non-negative, giving the actual number of characters transferred (not counting the terminating zero byte).

If btuinp() is called when btusts() has *not* returned a status of 3 for this channel, then the len return code will probably equal -1, indicating that no CR-terminated data string exists in the buffer. If this happens, whatever characters were available are copied into your buffer anyway, and a zero is placed at the end of them. These characters are *not* flushed from the receive buffer in this case — you will see them again once a complete line has been received.

You may use btutrm() (page 169) to change the line terminator character from ASCII CR to some other character.

CAUTIONS

Unless you are doing something unusual, you should only call this routine after btusts() returns a status of 3.

Be sure that the memory area addressed by inbuff is big enough to contain the maximum size string anticipated (including the terminating zero). btumil() (page 122) will set a limit on the input length, but to be safe, you should probably expect up to the size of the input buffer (isiz, as specified in btusiz() or btulsz()).

RETURNS

The function’s return value (the variable len, above):

-10 channel is not defined (see btudef(), page 80)
-11 channel number is out of range
(see btusiz(), page 152 or btulsz(), page 117)
-1 a complete input line is not yet available (no CR yet)
N>=0 the length of the returned string
(not including line terminator, not including 0-terminator)

The character string (address inbuff in synopsis):

The input line entered by this user on this channel (not including the line terminator, but terminated instead by a 0-byte), is stored in the character array pointed to by the parameter inbuff.

btuirp

SUBROUTINE NAME

btuirp — define alternate GSBL timing source using COM1/2/3/4

SYNOPSIS

```
err=btuirp(comno);
int err;                    zero means OK
int comno;                  0=use real-time interrupt (8253)
                            1=use COM1:  03F8, interrupt 4
                            2=use COM2:  02F8, interrupt 3
                            3=use COM3:  03E8, interrupt 4
                            4=use COM4:  02E8, interrupt 3
```

DESCRIPTION

The GSBL is based on timed polling of all ports. Normally the system timer interrupt is intercepted and then its rate is multiplied by a factor sufficient to catch all characters at the highest baud rate.

The btuirp() routine specifies an MS-DOS asynchronous communication port, or COM port, to use as an alternate timing source. The port's baud rate and transmit interrupt are configured to provide a regular interrupt for polling all ports. This may be helpful in operating systems or MS-DOS-like environments in which the 8253 timing device is not available.

CAUTIONS

You must only call btuirp() once, immediately following the call to btuitz(), and coming before any other calls to GSBL routines that also follow btuitz(). The COM port interrupts 3 and 4 are probably at a lower priority in your system than the interrupt 0 generated by the 8253 timer. This may mean that btuirp(1-4) will increase jitter and increase the possibility of missed incoming characters.

RETURNS

-17	comno is not 0/1/2/3/4, or COM port specified is not available
0	all is well

btuitm

SUBROUTINE NAME

btuitm — initialize the Software Breakthrough for use
in a multitasking environment

SYNOPSIS

err=btuitm(region);	
int err;	0 is OK
char *region;	pointer to memory region (size indicated by btusiz() or btulsz())

DESCRIPTION

This alternative to btuitz() is used to initialize the GSBL in a multitasking environment such as Windows or OS/2. The main differences between btuitm() and btuitz() are:

1. btuitz() hooks the GSBL into the timer interrupt IRQ0 (CPU interrupt 08). At each timed interrupt, all channels will be serviced.
2. btuitm() does not hook into the timer interrupt. Instead, it's left up to you to call btuhit() (page 97) to explicitly designate which of several interupts to hook into, IRQ2 to IRQ7 (CPU interrupts 0A through 0F hex).

CAUTIONS

See starting on page 112 for the descriptions and cautions associated with btuitz().

RETURNS

0	all is well
-16	region pointer parameter is NULL (all zeros)
-15	memory allocation error
-19	protected mode memory tiling failed

btuitz

SUBROUTINE NAME

btuitz — initialize the Software Breakthrough

SYNOPSIS

```
err=btuitz(region);
int err;                 0 is OK
char *region;            pointer to memory region
                         (size indicated by btusiz() or btulsz())
```

DESCRIPTION

This routine initializes the Software Breakthrough package.

btuitz() must be called after either btusiz() or btulsz(), and before any other GSBL routine. See line 7 of the example program on page 197 for an example of using btuitz(). The region parameter must be a pointer to a memory region set aside for GSBL's exclusive use. The size of this region, in bytes, must be equal to the value returned by either btusiz() or btulsz(), whichever was previously called. Use btusiz() if the size of memory you will need is less than 64K. Use btulsz() if more, or if you are not sure.

If using protected mode, the region pointer passed to btuitz() must have an offset portion of 0 and be based upon a selector capable of being tiled by the GSBL using operating system calls.

CAUTIONS

btuitz() assumes that no errors had been reported by btusiz() or btulsz() (whichever you used, pages 152, 117) — in particular, that neither had returned an error code and that your dynamic memory allocator, if any, was able to allocate a block of the requested size.

This routine must be called exactly once at the beginning of execution, and btuend() must be called before btuitz() can be called again.

RETURNS

0	all is well
-16	region pointer parameter is NULL (all zeros)
-15	memory allocation error
-19	protected mode memory tiling failed

btulfd

SUBROUTINE NAME

btulfd — set linefeed character (what follows every carriage return)

SYNOPSIS

err=btulfd(chan,lfchar);		
int err;	zero means OK	
int chan;	channel number	
char lfchar;	> 0	character to output after each carriage return (defaults to 10, the ASCII linefeed character)
	= 0	disabled

DESCRIPTION

This routine enables automatic generation of linefeeds after carriage returns during the ASCII output mode (page 15). There is a lack of standardization in communications today regarding whether a "carriage return" also implies advancing to the next line or not. On some terminals, displaying a CR (ASCII code 13) causes the cursor to merely return to column 1 of the current line. On others, it also moves the cursor down the screen one line (or scrolls the screen, or advances the paper as the case may be).

The default upon channel initialization or reset is to assume the former: that is, that an explicit LF byte (ASCII code 10) is necessary following each CR in order to move on to the next line. However, this will cause lines to appear double-spaced under some conditions. You can eliminate the LF, or replace it with some other ASCII code, using the btulfd() routine. The btulfd()-defined linefeed character also has a minor effect in ASCII input mode: after a carriage return (defined by btutrm()) is echoed, the linefeed character is also echoed.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range

(see btusiz(), page 152 or btulsz(), page 117)

0 all is well

btulok

SUBROUTINE NAME

btulok — set input lockout on/off

SYNOPSIS

```
err=btulok(chan,onoff);
int err;                    zero means OK
int chan;                   channel number
int onoff;                  1 = lockout, 0 = remove lockout
```

DESCRIPTION

Allows a channel’s input to be locked out. You might use this in a “deaf good-bye” scheme. This is a log-off sequence in which you do not want the user to be able to throttle output (via XOFF, see btuxnf(), page 193), or to issue further commands, during the output of a final message.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	all is well

btulsz

SUBROUTINE NAME

btulsz — size of dynamic memory needed
(long version, used when more than 64K needed)

SYNOPSIS

lsiz=btulsz(nchan,isiz,osiz);	
long lsiz;	total size needed, in bytes
int nchan;	number of channels to allow for
int isiz;	input data buffer size per channel
int osiz;	output data buffer size per channel

DESCRIPTION

Calculates the size of the memory region needed by the Software Breakthrough package. Either this routine or btusiz() must be called prior to calling btuitz() to initialize the system. The input and output data buffer sizes specified must be integral powers of two (128, 256, 512, 1024, etc). The actual number of bytes that each buffer can hold will be 1 less than the size you specify. For example, if you specify an input data buffer size of 128, then blocks of up to 127 characters at a time may be input without overflowing. This routine specifies the total number of channels you will support, and the sizes of the input and output buffers associated with each channel. This can be done in either of two ways:

1. If the language you are using supports dynamic memory allocation (this is usually called a heap or a pool), you can simply pass the return value of btusiz() or btulsz() to your allocator, specifying the number of bytes you want to allocate. This is the preferred method.
2. If you have no capability for dynamic memory allocation, and you must allocate all your data structures before your program runs, this is what you can do: write a separate little test program to call btusiz() or btulsz() with the appropriate parameters and simply print out the result. This is the number of bytes that the Software Breakthrough will need. Then create a fixed-length array of this size in the main program (the one that will be actually doing the communications). If you do this, you *still* must call btusiz() or btulsz() in your main program just before you call btuitz().

Whether you use method 1 or method 2, the address of a memory region this many bytes long must be passed to btuitz().

CAUTIONS

Even when using method 2 above, either btusiz() or btulsz() must be called before calling btuitz(). There is no other way to inform the Software Breakthrough of the total number of channels and of their buffer sizes. Method 1 is preferred over method 2. Under method 2, the size test procedure must be redone every time you receive an update to the Software Breakthrough, while relinking is all that is required under method 1. You will use method 2 only when you require static memory (that is, when you cannot use dynamically allocated memory).

LAN

On LAN channels, input buffers and output buffers should usually be of sufficient size to accommodate full packet contents (546 bytes for IPX channels, 534 bytes for SPX channels). If either buffer is smaller than a full packet, then even SPX channels cannot necessarily be guaranteed against data loss (unless you're sure the main program won't require it).

Be sure to declare btulsz() as a function returning a long (32-bit) integer in one of the variable declaration sections of your C-language calling program. For example:

```
main()
{
     int i,j,k;
     char *farmalloc();
     long btulsz();

     btuitz(farmalloc(btulsz(64,1024,2048)));
     :
     :
     btuend();
}
```

Note: farmalloc() is a library routine of Borland's C++ compiler. You will need a memory allocation routine, like this one, that accepts a long parameter instead of merely integer.

RETURNS

N >= 0L This is the size, in bytes, of the memory region
needed by the Software Breakthrough

-1L Error: one of the buffer size parameters is not
an integral power of two

btumds

SUBROUTINE NAME

btumds — get next displayed character from the monitored channel
(as specified by btumon(), see page 126)

SYNOPSIS

dspchr=btumds();	
int dspchr;	character that was output or echoed (0 if no characters)

DESCRIPTION

Returns the next buffered output character from the monitored channel, or returns 0 if the display monitor buffer is empty. Use btumon() (page 126) to select the monitored channel.

CAUTIONS

When you are monitoring a channel, be sure to call this routine often enough that the monitor-output buffer will not overflow (it can hold up to 2047 characters). Also, when you do get around to calling this routine, we recommend that you call it repeatedly until it returns 0, to flush out its buffer.

RETURNS

0	the monitor-output buffer is empty
dspchr > 0	the next character from the monitor-output buffer

btumds2

SUBROUTINE NAME

btumds2 — get next displayed character from the monitored channel (as specified by btumon2(), see page 128)

This function is a clone of btumds(), for emulating a second channel.

SYNOPSIS

dspchr=btumds2();	
int dspchr;	character that was output or echoed (0 if no characters)

DESCRIPTION

Returns the next buffered output character from the monitored channel, or returns 0 if the display monitor buffer is empty. Use btumon2() (page 128) to select the monitored channel.

CAUTIONS

When you are monitoring a channel, be sure to call this routine often enough that the monitor-output buffer will not overflow (it can hold up to 2047 characters). Also, when you do get around to calling this routine, we recommend that you call it repeatedly until it returns 0, to flush out its buffer.

RETURNS

0	the monitor-output buffer is empty
dspchr > 0	the next character from the monitor-output buffer

btumil

SUBROUTINE NAME

btumil — set maximum input line length, input word wrap on/off

SYNOPSIS

```
err=btumil(chan,maxinl);
int err;                  zero means OK
int chan;                 channel number
int maxinl;               if > 0    then input word wrap is disabled and
                                    maxinl is the maximum input line length
                          if = 0    then input word wrap is disabled and there is
                                    no limit to the length of an input line
                                    (see status 251, page 163)
                          if < 0    then input word wrap is enabled and
                                    abs(maxinl) is maximum input line length
```

DESCRIPTION

This routine may be used to restrict each line of input data to a specified field width. If a user should attempt to type more than the limit of characters on a single line, characters after the limit will neither be stored in the input buffer nor echoed back to the user (but status 251's will be generated).

In interactive applications, you often want to restrict the length of a user's input line — maybe you only have 15 characters in which to store his name. Rather than accept data that is too lengthy, and truncate it after the user has pressed ENTER, it is nicer for the user to "feel" the restriction while he is typing. He can then backspace and retype some kind of abbreviation.

This routine also selects the input word wrap feature. To turn it on, make maxinl the negative of the screen width. For example, if maxinl is -79, then a user with an 80-column wide screen may type electronic mail without worrying about the right margin. Note: the last column should never actually be used, to prevent the display device from auto-scrolling. For example, if maxinl = -79, then word wrapping kicks in upon typing the

80th character of the line — so that each line can be no longer than 79 characters.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	all is well

btumks

SUBROUTINE NAME

btumks — simulate a keystroke on the monitored channel
(as specified by btumon(), see page 126)

SYNOPSIS

btumks(kyschr);	
char kyschr;	simulated-input character

DESCRIPTION

Simulates a keystroke on the user's terminal. The kyschr character is placed into the receive buffer just as if it had been received from the channel. This can be used to parallel the system operator's keyboard with that of the monitored user.

CAUTIONS

Use of key macros or other means of rapidly issuing large blocks of keystrokes should be avoided with this routine, since the input-keystroke buffer can hold only 15 entries.

This routine has no effect if btumon() (page 126) has not been called, or if it was last called with a parameter of -1.

RETURNS

Nothing.

btumks2

SUBROUTINE NAME

btumks2 — simulate a keystroke on the monitored channel
(as specified by btumon2(), see page 128)

This function is a clone of btumks(), for emulating a second channel.

SYNOPSIS

```
btumks2(kyschr);
char kyschr;                    simulated-input character
```

DESCRIPTION

Simulates a keystroke on the user’s terminal. The kyschr character is placed into the receive buffer just as if it had been received from the channel. This can be used to parallel the system operator’s keyboard with that of the monitored user.

CAUTIONS

Use of key macros or other means of rapidly issuing large blocks of keystrokes should be avoided with this routine, since the input-keystroke buffer can hold only 15 entries.

This routine has no effect if btumon2() (page 128) has not been called, or if it was last called with a parameter of -1.

RETURNS

Nothing.

btumon

SUBROUTINE NAME

btumon — start/stop monitoring a channel

SYNOPSIS

```
err=btumon(chan);
int err;                    zero means OK
int chan;                   channel number (-1 to disable monitoring)
```

DESCRIPTION

btumon() sets up a specific channel for monitoring, which means that each character transmitted to the channel can be obtained by calling btumds(), and you can simulate input from the channel by calling btumks(). Monitoring can be turned off by passing -1 to btumon().

The three routines btumon(), btumds(), and btumks() are designed for tuning in to any desired online channel from a master console. When monitoring a channel, the system operator can see everything the user of that channel is seeing: each keystroke echoed, and each block of text output. Also, the system operator's keyboard can be placed in parallel with that user's keyboard, in the sense that anything typed on the system operator's keyboard is processed exactly as though it had come from the monitored user's keyboard — it even echoes to both displays. There are two uses for the channel monitoring feature:

1. When a user is on the channel: to "look over a user's shoulder" and see what he is doing, maybe even help him along a little, by typing for him.
2. When no user is on the channel: for the system console operator to act as a user himself, as if he had dialed up the system with a modem and a terminal. This is called local emulation. This method can be used only on a channel with no actual modem hardware.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range

(see btusiz(), page 152 or btulsz(), page 117)

0 all is well

btumon2

SUBROUTINE NAME

btumon2 — start/stop monitoring a channel

This function is a clone of btumon() for emulating a second channel.

SYNOPSIS

err=btumon2(chan);	
int err;	zero means OK
int chan;	channel number (must be a non-hardware channel), or -1 to disable monitoring

DESCRIPTION

tumon2() sets up a specific channel for monitoring, which means that each character transmitted to the channel can be obtained by calling btumds2(), and you can simulate input from the channel by calling btumks2(). Monitoring can be turned off by passing -1 to btumon2().

The three routines btumon2(), btumds2(), and btumks2() are designed for tuning in to a non-hardware channel from a master console.

The only use for channel monitoring with btumon2() is when the channel is a non-hardware channel, for the console operator to act as a user himself, as if he had dialed up the system with a modem and a terminal. This is called local emulation. This method can be used only on a channel with no actual modem hardware.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	all is well

btumxs

SUBROUTINE NAME

btumxs — set maximum data speed

SYNOPSIS

```
err=btumxs(bdrate);
int err;                  error return: 0=OK, -3=rate no good
unsigned bdrate;          baud rate (bits per second)
                          Default=2400  min=300  max=38400
```

DESCRIPTION

This routine specifies the maximum data rate (in bits per second) of all channels on your system.

This directly affects the rate at which all channels are serviced. The byte rate is derived from the baud rate (bdrate) that you specify with btumxs():

bytes per second = bdrate / 10

A 21% margin is added to this rate to get the channel service rate:

service rate = bytes per second * 1.21

For example, with the default bdrate of 2400 baud, data is expected for receiving and transmitting at 240 bytes per second. Therefore each channel is serviced about 290 times a second.

This parameter also affects the service rate of channels that are monitored using btumds() and btumds2() (pages 120 and 121), even if no hardware is connected to this channel (see use #2 on page 126). For example, using Worldgroup, when you emulate a non-hardware channel (one that appears as "–" in the user matrix on the console), the display rate is controlled by the bdrate parameter of btumxs(). For this reason, you may want to set the service rate higher than that required by your hardware, so that your emulated screen is updated faster.

Using 16550 FIFOs

When you use 16550 UARTS, you may be able to set the btumxs() rate even slower than the maximum baud rate. That's because the 16550 UARTS have hardware FIFOs in them and don't need to be polled as often. This can save your computer a lot of processing overhead, because it doesn't have to poll your channels as rapidly.

To take advantage of this, you need to distinguish ports with 16550 UARTs from those with standard 8250 UARTs. The btuffo() routine (see page 90) returns 0 for 16550 ports, and -18 for 8250 or 16450 ports (it also returns 0 for Xecom or any other non-8250 based port).

So, to figure the bdrate parameter for btumxs(), compute the maximum of all of the polling rates that are required for all the channels. The polling rate required for each channel is:

- 16550 channel: the baud rate / 4
- Non-16550 channel: the baud rate

Here are some examples:

```
 8250 port at   2400 bps          btuffo(chan,1) = 0
16550 port at   4800 bps          btuffo(chan,1) = 1
btumxs(2400)

 8250 port at   2400 bps          btuffo(chan,1) = 0
16550 port at   9600 bps          btuffo(chan,1) = 1
16550 port at 19200 bps           btuffo(chan,1) = 1
btumxs(4800)
```

CAUTIONS

If you make the bdrate parameter smaller than the baud rate of the fastest channel on your system, you may lose received characters and you may be transmitting at less than capacity. If you make the bdrate parameter larger than the baud rate of the fastest channel on your system, you will be wasting CPU time servicing channels more often than they require.

RETURNS

0	the maximum data speed has been set
-3	bad baud rate: bdrate was not between 300 and 38400

btuoba

SUBROUTINE NAME

btuoba — Output Bytes Available: report the amount of space
(number of bytes) available in the output buffer

SYNOPSIS

outbca=btuoba(chan);	
int outbca;	output buffer size, or error code
int chan;	channel number

DESCRIPTION

This routine simply returns the size of the vacant portion of the output data buffer for the specified channel. It is intended for use in two ways (although you may find others):

1. You can find out whether or not an output message will fit in the space available, thereby avoiding a status of 253 (DATA OUTPUT CIRCULAR BUFFER OVERFLOW), and allowing your program to take remedial action (such as outputting a beep string to let the user know an output block has been lost).
2. You can find out when the buffer is empty, if that makes a difference to you in some situation (such as for performance monitoring, in which you might want to get some feel for the fraction of online time spent outputting). The output buffer is empty when btuoba() returns a value of osiz-1, where osiz is the size of the output buffer specified in the original call to btusiz() (page 152) or btulsz() (page 117).

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	the channel number is OK but that channel's output buffer is full
N>0	the number of bytes available in the channel's output buffer

btuoes

SUBROUTINE NAME

btuoes — enable/disable Output-Empty status codes

SYNOPSIS

err=btuoes(chan,onoff);

int err;	zero means OK	
int chan;	channel number	
int onoff;	= 1	generate a single status 5 when the output buffer becomes empty
	= 0	don’t generate status 5’s

DESCRIPTION

You may wish to be notified when the data output buffer goes empty on a channel, such as with certain block-oriented protocols like XMODEM. By default, no status 5 (see page 156) is generated. This routine turns the generation of this status on or off.

CAUTIONS

If the output buffer on the channel is already empty when this routine is called, no status 5 is generated. The test for generation of status 5 only occurs as the channel transitions from the not-empty state to the empty state.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	all is well

btuolk

SUBROUTINE NAME

btuolk — set output pausing on/off

SYNOPSIS

err=btuolk(chan,onoff);	
int err;	zero means OK
int chan;	channel number
int onoff;	1 = pause, 0 = resume, pause off

DESCRIPTION

This routine pauses a channel's output until the pause is turned off by btuolk(). You might use this in some scheme to throttle output. For example, to implement a file transfer protocol that uses XON/XOFF in binary mode, your mainline program will need to process these characters and pause or resume output.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	all is well

btupbc

SUBROUTINE NAME

btupbc — set screen-pause character (pauses screen when in output stream)

SYNOPSIS

err=btupbc(chan,pausch);

int err;	zero means OK
int chan;	channel number
char pausch;	pause character (0 to disable)

DESCRIPTION

When the pausch character is transmitted to the user in ASCII output mode, output pauses and the channel goes into the screen-pause mode (page 98). Clear screen characters (ASCII formfeed and ANSI ESC [2J) are automatically preceded by the pause character when btuxmt() stuffs them into the output buffer. Worldgroup uses CTRL+T for the pause character.

Characters received during the screen-pause mode are handled by a routine identified by btuhpk(). That routine specifies how to get out of screen-pause mode.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	all is well

btupmt

SUBROUTINE NAME

btupmt — set prompt character

SYNOPSIS

err=btupmt(chan,pmchar);

int err;	zero means OK
int chan;	channel number
char pmchar;	prompt character (0 to disable)

DESCRIPTION

This routine selects automatic prompting of the user with a single character. In this example, **>** is the prompt character, and the user's keystrokes are shown in boldface:

```
System ready
>What time is it?
Oh, about half past eight
>Download the marketing report.
OK, the marketing report will be downloaded when you log
off.
(Type "NOW" to do it now.)
>
```

Even when enabled, prompting is only active during the ASCII output mode (page 15) — that is, when using btuxmt() to transmit (page 189).

The prompt character is automatically sent each time a transition is made from an outputting state to an inputting state. To disable this feature (the default condition), call btupmt() with a pmchar value of 0.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	all is well

bturep

SUBROUTINE NAME

bturep — report channel statistics

SYNOPSIS

value=bturep(chan,statid);

long value;	value of the statistic
int chan;	channel number
char statid;	0 = count of characters (LAN, X.25)
	1 = count of packets (LAN, X.25)
	2 = count of overruns (8250)
	3 = count of parity errors (8250)
	4 = count of framing errors (8250)

DESCRIPTION

These statistical values are set to zero by btursT() (page 138), or to any value by btuset() (page 150).

LAN X.25

The output of this function is only defined for X.25 and LAN channels when statid is 0 or 1. A count of characters and packets that have been transferred in both direction since channel reset is reported.

UART

The output of this function is only defined for 8250-type UART channels when statid is 2-4. This function reports a running count of overrun, framing, and parity errors in the UART of the channel.

Overrun errors in non-16550-FIFO mode can mean one or more characters lost per error. In 16550 mode (page 90), up to 16 characters or more can be lost per overrun error.

Framing errors usually mean baud-rate incompatibility or line noise. Technically they result from the lack of a high (mark) state when a stop bit was expected.

RETURNS

bturep() returns the current long integer value of the statistic.

bturst

SUBROUTINE NAME

bturst — reset a channel

SYNOPSIS

err=bturst(chan);

int err;	0 = OK (Xecom hardware)
	1 = OK (Hayes or UART hardware)
	2 = OK (X.25 hardware)
	3 = OK (LAN hardware)
	less than zero means error (see page 140)
int chan;	channel number

DESCRIPTION

bturst() completely resets a channel, in both hardware and software, to its initial default conditions. This is also the recommended method for "hanging up", since the default condition of the switch hook is on-hook (disconnected).

Note: when btudef() (page 80) is used to define a channel or group of channels, the bturst() reset operation is automatically performed on these channels.

HAYES

We recommend that as part of resetting any channel with Hayes category hardware, you issue the following sequence of instructions, selecting certain non-default options on which our interface scheme depends:

```
bturst(chan);
btulok(chan,1);
btuoes(chan,1);
btuech(chan,0);
btubrt(chan,2400);
btuxmt(chan,"ATE0S0=1S2=1&C1&D2\r");
btucli(chan);
```

. . . wait for status 5 from this channel . . .

```
btulok(chan,0);
btuoes(chan,0);
```

This procedure is used in the TELCONH.C example program on page 197.

This should also be done to every channel after it is defined (see btudef(), page 80). This is the rationale of each of these statements:

bturst(chan);

Reset the channel.

btulok(chan,1);

Lock out input from the channel. This inhibits trash incoming characters from inhibiting transmission of the upcoming command string.

btuoes(chan,1);

This enables status code 5 generation when the transmit buffer goes empty. This will be used to detect when the command string completes transmission to the modem.

btuech(chan,0);

This turns off the echo in the Software Breakthrough (a different echo than that of the Hayes-compatible modem). For interactive applications, you will turn your echo on again after in incoming call has been answered.

btubrt(chan,2400);

This sets the baud rate of the UART that communicates with the modem. 2400 is the default baud rate, and if that's what you want, you can omit this statement. But if you want some other rate during communication with the modem in command mode, then set that rate here.

btuxmt(chan,"ATE0S0=1S2=1&C1&D2\r");

This command string selects various modes of the modem. AT is the standard prefix to all Hayes protocol commands.

E0 turns output echo off. Without this, the Hayes-compatible modem would echo back to us every character that we sent it, certainly not what we want.

S0=1 enables auto-answer mode, so that incoming calls are automatically answered and presented with answer-carrier. You may wish to omit this setting if you have phone lines that you don't wish to answer.

S2=1 selects '\1' (ASCII 01, CTRL+A) as the escape character. Per the Hayes modem control standard, transmitting the escape code three times (between 1 second pauses) results in switching from the online to the command mode. The default escape character is 43 (+). Since we do not want users to be able to issue the escape sequence, we make the escape character something that cannot be echoed back. Note that the default translate table (page 185) has ASCII 01's translated to 00 (ignored).

&C1 makes the modem DCD output indicate the presence of data carrier.

&D2 makes the modem DTR input reset the modem.

\r this is the C-language notation for a carriage return.

The &C1 and &D2 commands may not be desirable or necessary on certain hardware. See your modem manual for details.

btucli(chan);

This instruction discards any trash input characters that may have been received up to this point.

. . . wait for status 5 from this channel . . .

To continue the reset procedure, we must wait for the status code 5 that indicates that the UART on this channel has completely transmitted the command sequence to the modem. In a multi-user program, remember that other channels may require servicing in this interval. You must structure your code such that all other channels will continue to be serviced, and when this status 5 does come in, the reset procedure will continue. The program on page 197 does this.

btulok(chan,0);

Remove input lockout.

btuoes(chan,0);

Turn off status code 5 generation.

CAUTIONS

LAN

bturst() should always be called for each LAN channel just before you call btuend(). This ensures that SPX connections are completely aborted.

RETURNS

-10 channel is not defined (see btudef(), page 80), or no UART has been detected at the I/O address where it was defined. If you have purchased the N-channel version of the Software Breakthrough, then you may only use channels 0 through N-1 to talk to real hardware. If you attempt to reset channel N or higher, then you will get this error return code. You may still use these channels for local emulation (page 126), however.

-11 channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)

XECOM

0 means that a functional Xecom category modem has been detected on this channel

HAYES UART

1 means that functional Hayes or UART category hardware has been detected on this channel

X.25

	2	means that functional X.25 driver and PC XNet hardware have been detected on this channel
LAN	3	means that a functional LAN interface has been detected on this channel

bturti

SUBROUTINE NAME

bturti — define routine to be called in real-time

SYNOPSIS

```
err=bturti(hertz,rtirou);
int err;                    zero means OK
int hertz;                  number of rti's per second
int (*rtirou)();            pointer to function to be called
```

DESCRIPTION

This sets a specified routine to be called a specified number of times per second. The call takes place at *interrupt* level: once your program calls bturti(), no maintenance is necessary. Call timing is not affected by other activity such as disk I/O or lengthy mainline computation loops.

CAUTIONS

The hertz parameter may not exceed 0.12 times the maximum baud rate specified by btumxs() (page 129), or 288 if btumxs() has not been called. The time between rti's is quantized into discrete units equal to 0.12 times the btumxs()-specified maximum baud rate. So, as the specified rti rate approaches this maximum, the variation in intervals between calls to the rtirou routine can be as much as 100%. The average rate per second will be much more accurate, however (closer than 0.1%).

See also page 48 for cautions relating to interrupt-called functions. Design of the rtirou routine is fraught with pitfalls. Don't design lengthy complex code to run at interrupt level. Don't invoke DOS (because it is not reentrant). You may use the special chixxx() routines (page 47), but not any other Breakthrough Library routines. Beware of data “skewing”, since a real-time interrupt may occur between any two machine-language instructions of your mainline code.

RETURNS

-6	too many rti calls per second (hertz is too large)
0	all is well

btuscn

SUBROUTINE NAME

btuscn — scan for channels in need of service (with nonzero status)

SYNOPSIS

chan=btuscn();	
int chan;	channel number (0 to nchan-1) or -1, if no channels require service

DESCRIPTION

Scans all defined channels, searching for one with a nonzero status. If it finds one, the number of that channel is returned. This number will be between 0 and nchan-1, where nchan is the total number of channels allocated in btusiz() page 152 or btulsz() page 117. If no channels require service, -1 is returned, indicating that the status queues of all channels are empty. When btuscn() reports that a channel requires service, subsequent calls to btuscn() resume scanning with the channel immediately following, so that all channels have the same priority. This routine will most likely be the focal point of the main program. The main loop will repeatedly call btuscn() to find out what to do next — that is, which channel needs servicing. You could do the same thing with repeated calls to btusts(), but btuscn() is far more time-efficient. Also, btuscn() gives equal priority to all channels.

LAN X.25 The btuscn() function has added purpose on LAN and X.25 channels. It scans all channels for incoming packets, and processes them. It scans all channels with output data for the opportunity to transmit packets. The standard function of btuscn() applies to all channels: to look for the next channel with a status code to report.

CAUTIONS

LAN X.25 On LAN and X.25 channels, you must call btuscn() regularly. If you do not, you won’t have any input or output on those channels.

RETURNS

-1 no channels require service: they all have a status of 0

N>=0 channel number of the next channel with a nonzero status

btuscr

SUBROUTINE NAME

btuscr — set the soft-CR character (for output word wrap)

SYNOPSIS

```
err=btuscr(chan,softcr);
int err;                 zero means OK
int chan;                channel number
char softcr;             soft-CR, becomes space once wrapped
                         (0 to disable soft-CR translation)
```

DESCRIPTION

The *soft* carriage return (see the discussion of word wrap under btutsw(), page 172) is an output character that gets converted into a CR or CR LF sequence (CR LF if you have never called btulfd()) as long as no output word wrap has taken place yet. Otherwise, it is converted into a SPACE (ASCII code 32). It defaults to the disabled condition (softcr=0) when a channel is initialized or reset. Soft carriage returns are processed by btuxmt() during transmission — that is, they are only active during the ASCII output mode (page 15).

Background: when output word wrap is enabled, btuxmt() keeps track of whether or not a forced wrap has occurred within the current paragraph. After it has, appearances of the soft CR character are treated exactly like spaces. Before this point, appearances of the soft CR character are treated exactly like carriage returns. See the example on page 174, where the soft carriage return is set to 10 (\n in C language).

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	all is well

btusdf

SUBROUTINE NAME

btusdf — super-define channel groups

SYNOPSIS

err=btusdf(schan,n,chtype,...);

int err;	zero means OK (see below)
int schan;	starting channel number
int n;	number of channels
int chtype;	channel interface type:
	0 wildcard (auto-determine whether type 1, 2 or 3) (this is the same as btudef())
	1 Xecom 120x (as on Breakthrough Model 16 or 4)
	2 Xecom 2400 (as on Breakthrough Model 2408)
	3 8250-type UART/modem
	4 X.25 packet switching network channels
	5 Novell Netware IPX Direct circuit channels
	6 Novell Netware IPX Virtual circuit channels
	7 Novell Netware SPX channels

... the remaining parameters depend on the value of chtype:

Format of call to `btusdf()`	Purpose
`err=btusdf(schan,n,0,ioaddr);`	same as `btudef()`
`err=btusdf(schan,n,1,ioaddr);`	Model 16/4
`err=btusdf(schan,n,2,ioaddr);`	Model 2408
`err=btusdf(schan,n,3,ioaddr);`	8250-type UART
`err=btusdf(schan,n,4,card,line,slcn);`	X.25 network
`err=btusdf(schan,n,5,socket,necbs);`	LAN IPX Direct
`err=btusdf(schan,n,6,socket,necbs);`	LAN IPX Virtual
`err=btusdf(schan,n,7,socket,necbs);`	LAN SPX

int ioaddr;	starting port address
int card;	PC XNet card number (0-7)
int line;	PC XNet line (0=25-pin 1=15-pin)
int slcn	Starting logical channel number
int socket;	local socket number, or 0=define
int necbs;	number of listen ECB's to open

DESCRIPTION

You can use these symbols from BRKTHU.H for the chtype parameter:

```
#define SDFANY  0  /*GSBL btusdf() argument for XECOM/UART hdw */
#define SDFX25  4  /*GSBL btusdf() argument for X.25 hardware  */
#define SDFIPXD 5  /*GSBL btusdf() argument for IPX Direct hdw */
#define SDFIPXV 6  /*GSBL btusdf() argument for IPX Virtual hdw*/
#define SDFSPX  7  /*GSBL btusdf() argument for SPX hardware   */
```

Here is how the hardware categories compare with the chtype parameter for various types of hardware:

Communications hardware	Hardware category	chtype
Breakthrough Model 4	Xecom	1
Breakthrough Model 16	Xecom	1
Breakthrough Model 2408	Hayes	2
Hayes 2400B internal modem	Hayes	3
IBM Async adapter card	UART	3
OST PC XNet card	X.25	4
Novell LAN, IPX Direct	LAN	5
Novell LAN, IPX Virtual	LAN	6
Novell LAN, SPX	LAN	7

Calling btusdf(schan,n,0,ioaddr) is exactly the same as calling btudef(schan,ioaddr,n). The GSBL will automatically figure out the channel type among the Model 4/16, Model 2408, 8250 possibilities. See page 80.

Using btusdf() with the chtype parameter value of 1, 2 or 3 forces the GSBL to treat the channel as a Model 16/4, Model 2408, 8250 UART.

LAN

For LAN channel definition, the socket number is specified in the sensible, easy-to-use “lo-hi” order (it gets byte-swapped into “hi-lo” order before being handed off to NetWare). See page 213 for details on socket numbering.

The necbs parameter defines the number of ECB’s per channel to allocate for receiving packets. This parameter should probably be a minimum of 2.

Maximum possible value for necbs is 64. Memory overhead will be about 640 bytes per ECB per channel. Setting the necbs parameter to 0 will define a non-hardware LAN channel. Every channel should have at least 2 ECBs at its disposal. You may be able to set necbs to 1 if you are defining more than one channel in a group — since all ECBs for a given socket number are shared (among all the channels that use that socket number), then theoretically every channel will have at least 2 ECBs at its disposal.

CAUTIONS

LAN X.25

There is no btuudf() (un-define) for LAN or X.25 channels. Defining a LAN or X.25 channel (calling btusdf() with chtype from 4 to 7) is permanent and can occur only once for a given channel over the run-time life of your program.

The only way to re-define LAN or X.25 channels is by calling btuend() to shut down the entire Software Breakthrough, and starting over again with btuitz(). And since you can only run btuitz() and btuend() once per program load, this means that LAN and X.25 channels can only be defined once for the run-time life of your program.

LAN

Defining a LAN channel group of one channel with one ECB per channel will have undefined results.

RETURNS

	0	all is OK
	-11	channel number is out of range
	-12	sequence error — overlapping channel groups
X.25	-13	X.25 interface not available (only returned when chtype is 4 and you are using GSBL without the X.25 option)
LAN	-14	LAN interface not available (only returned when chtype is 6 or 7)
	-15	out of memory

btuset

SUBROUTINE NAME

btuset — set and report channel statistics

SYNOPSIS

value=btuset(chan,statid,newval);

long value;	value of the statistic
int chan;	channel number
char statid;	0 - count of characters (LAN, X.25) 1 - count of packets (LAN, X.25) 2 - count of overruns (8250) 3 - count of parity errors (8250) 4 - count of framing errors (8250)
long newval;	new value for the statistic

DESCRIPTION

These statistical values are set to zero by bturst() (page 138). The value returned by btuset() is the value of the statistic immediately before it gets set to the new value. A common practice is to call btuset() with newval=0L and use the result to keep a separate total.

LAN X.25

The operation of this function is only defined for X.25 and LAN channels when statid is 0 or 1. A count of characters and packets that have been transferred in both direction since channel reset is reported. For example, you could keep track of packet traffic with code like this:

```
long pdelta;
static long ptotal=0L;
long btuset();

pdelta=btuset(chan,1,0L);
ptotal+=pdelta;
printf("%ld packets input/output, %ld total!\n",pdelta,ptotal);
```

UART

The operation of this function is only defined for 8250-type UART channels when statid is 2-4.

This function reports a running count of overrun, framing, and parity errors in the UART of the channel.

Overrun errors in non-16550-FIFO mode can mean one or more characters lost per error. In 16550 mode (page 90), up to 16 characters or more can be lost per overrun error.

Framing errors usually mean baud-rate incompatibility or line noise. Technically they result from the lack of a high (mark) state when a stop bit was expected.

RETURNS

The value of the statistic immediately before it gets set to the new value.

btusiz

SUBROUTINE NAME

btusiz — size of dynamic memory needed (only if < 64K)

SYNOPSIS

size=btusiz(nchan,isiz,osiz);

unsigned size;	total size needed, in bytes, or 65535 if too much memory needed (meaning you should use btulsz())
int nchan;	number of channels to allow for
int isiz;	input data buffer size per channel
int osiz;	output data buffer size per channel

DESCRIPTION

btusiz() calculates the total size of the memory region required by the Galacticomm Software Breakthrough Library to manage the communication channels. The address of a region this big must be passed to btuitz() (page 112) to initialize the Software Breakthrough.

If the parameters are such that more than 65535 bytes are needed, this routine returns 65535. In this case, you should use btulsz() (page 117) instead of btusiz(), because btulsz() returns a long integer (32-bit) quantity.

The input and output data buffer sizes specified must be powers of two (128, 256, 512, 1024, etc). The actual number of bytes that each buffer can hold will be 1 less than the size you specify. For example, if you specify an input data buffer size of 128, then blocks of up to 127 characters at a time (plus terminator) may be input without overflowing.

This routine specifies the total number of channels you will support, and the sizes of the input and output buffers associated with each channel.

This can be done in two ways:

1. If the language you are using supports dynamic memory allocation (this is usually called a heap or a pool), you can simply pass the return value of btusiz() or btulsz()

to your allocator, specifying the number of bytes you want to allocate. This is the preferred method.

2. If you have no capability for dynamic memory allocation, and you must allocate all your data structures before your program runs, write a separate test program to call btusiz() or btulsz() with the appropriate parameters and simply print out the result. This is the number of bytes that the Software Breakthrough will need. Then create a fixed-length array of this size in the main program (the one that will be actually doing the communications). If you do this, you *still* must call btusiz() or btulsz() in the main program just before calling btuitz().

In either event, the address of a memory region this many bytes long must be passed to btuitz().

CAUTIONS

Even using method 2 above, either btusiz() or btulsz() must be called before calling btuitz(). There’s no other way to inform the Software Breakthrough of the total number of channels and their buffer sizes.

Method 1 is preferred over method 2. Under method 2, the size test procedure must be redone every time you receive an update to the Software Breakthrough, while relinking is all that is required under method 1. You should use method 2 only when you require static memory (that is, when you cannot dynamically allocate memory).

LAN

On LAN channels, input buffers and output buffers should usually be of sufficient size to accommodate full packet contents (546 bytes for IPX channels, 534 bytes for SPX channels). If either buffer is smaller than a full packet, then even SPX channels cannot necessarily be guaranteed against data loss (unless you are sure the main program won’t require it).

RETURNS

0 to 65534	This is the size, in bytes, of the memory region needed by the Software Breakthrough
65535	Error: more than 65534 bytes are needed, or either buffer size is not a power of two

btusts

SUBROUTINE NAME

btusts — status of a channel

SYNOPSIS

status=btusts(chan);	
int status;	channel status code
int chan;	channel number

DESCRIPTION

Several events can occur on your communication channels that need your attention: an incoming call can RING, a user can type in his name and press his ENTER key, etc. In other cases, a command that you issue (see btucmd(), page 54) can be completed. You will want to respond to these and other conditions, and that is what btusts() is all about.

This routine returns the next status code from the status buffer of the specified channel. The status buffer is a “first-in-first-out” structure, meaning that btusts() gives you the status codes in the same order in which they are generated.

Some of the status codes returned by btusts() are reporting conditions sensed by software running in the PC. Other conditions are sensed directly by the individual modem. Some status codes indicate normal, expected conditions; others indicate errors. Each status code is discussed in detail below.

Be sure to review the **CAUTIONS** at the end of this section on btusts() (page 165).

RETURNS

0 = Quiet, Nothing Special To Report

If you use btuscn() to scan for channels that need service, you will never see this status code. btuscn() (page 144) identifies channels that have a status code of anything *but* zero, and when none do, it returns -1.

1 = Xecom RING-Indicate Or Lost-Carrier (Break)

The Xecom modem module XE1201 uses the same bit to indicate both incoming ring and loss of carrier. Since incoming ring can only happen when the phone is on the hook, and loss of carrier can only happen when it is off the hook, it is simply matter of context to distinguish between the two. You can always reset the channel to be sure (bturst(), page 138).

2 = Xecom Command Execution Completed OK

If the modem has not generated any status codes by the time it finishes processing a command string (see btucmd(), page 54), then this status code is generated, indicating successful completion of the command string.

3 = CR-Terminated Input String Available

Indicates that a complete line of input data has been received over a channel that is in ASCII input mode (page 12). Use btuinp() (page 108) to retrieve this string.

HAYES

In command mode, Hayes category hardware can indicate several conditions by way of status 3. A subsequent call to btuinp() will retrieve a description of the condition like BUSY or NO ANSWER.

X.25

See page 228 about the RING xxx CALLING xxx status 3 message that represents an incoming call.

4 = Byte-Count-Triggered Input Data Available

Indicates that the trigger count specified for this channel (see btutrg(), page 167) has been reached, and that a logical block of transparent-mode data is ready for examination via btuict() (page 105).

5 = Output Buffer Empty

This status code is generated when (1) the btuoes() routine has been called to enable it (see page 133), and (2) the output data buffer makes a transition from being not empty to being empty. This might be used when your program needs to know when a block of output has been completely transmitted, before the program can proceed to some other task.

6 = Output Aborted By User

This status code may be generated when the user aborts output. The btutrs() routine enables/disables status 6 generation. The user aborts ASCII output by typing the truncate character defined by btutru().

11 = Hayes Lost Carrier

The carrier signal (a tone normally sounded continuously over the phone line) was interrupted, most often caused by the user hanging up.

12 = Hayes/UART Command Execution Completed OK

This status code is generated when btucmd() completes execution of the soft commands (and for Hayes hardware, before execution of the hard commands). See page 59 for a discussion of hard and soft commands.

13 = Hayes/UART Invalid btucmd() Command Byte

An invalid command byte was passed to btucmd() for Hayes or UART hardware. Look for 13s in the last column of figure 4-1 (page 57).

21 = X.25 Incoming Clear Packet (End Of Session)

An incoming clear packet was received on this X.25 channel. This could either be from the other party on the network, indicating that he ended the session, or from the network itself, indicating there was some error with the connection. See page 93 about how to obtain the cause and diagnostic fields of a clear packet.

22 = X.25 Command Or Pause Completed

A btucmd() command on this channel has completed successfully. Either a P or p pause or an A answer could generate this code.

23 = X.25 Invalid btucmd() Command Byte

An invalid command byte was passed to btucmd() for this X.25 channel.

24 = X.25 Incoming X.29 String

This status occurs when an X.29 string is recieved (a data packet with the "Q" bit set. You can use btuhdr() to retrieve the data (see page 93).

31 = LAN SPX Connection Terminated By Other Side

The other party issued a T command (page 72) to terminate the SPX session.

Note on statuses 31 and 39: If an SPX session terminates immediately after it receives data, then your program may find out about both at the same time: after calling btuscn(), there will be new data in the buffer, and a status 31 or 39 in the status buffer. You may want to be sure all input data is processed before using a status 31 to reset your channel.

32 = LAN Pause Command Completed

A pause command (P or p, see page 71) on this LAN channel has just finished the 5 or 2 second delay.

33 = LAN Invalid btucmd() Command Byte

Either you issued an invalid btucmd() command code, or your outgoing call command (W...M) did not have exactly 24 hex digits, or the channel transmitter is busy right now. The latter may happen while transmitting, dialing out, listening for a connection, or terminating a connection.

34 = LAN SPX Incoming Connection Established

An L command (page 69) was issued on this channel to listen for an incoming SPX call. This status indicates that the call has been received and the connection is established.

35 = LAN SPX Outgoing Connection Established

A W command (page 73) was issued on this channel to establish an outgoing SPX connection. This status indicates the connection is complete.

36 = LAN SPX Termination Complete By This Side

You issued a T command to terminate the SPX connection (page 72) on this channel. This status indicates that the termination of the connection is complete.

37 = LAN Receiver Error

This status indicates that the listen ECB completion code was not 00h or EDh. Some kind of low-level communications or interface error has occurred. Error status 37 would show up on any channel arbitrarily in a group with the same socket number.

38 = LAN Received Unknown Or Unexpected Packet

For IPX Direct channels, an unsolicited packet was received.
For IPX Virtual channels, all channels on this socket are full.
For SPX channels, a packet with an unknown connection ID was received.

Error status 38 would show up on any channel arbitrarily in a group with the same socket number. Status 38’s are likely if you define multiple channel groups on the same local socket number but don’t define them on consecutive channels (refer to page 214).

39 = LAN Connection Error

For IPX channels, the outdial command could not find the network path to the destination node (in this case, the status 39 may come after a perplexing pause of several seconds, during which the system seems to be hung), or some error occurred when transmitting (e.g. network failure).

For SPX channels, the connection may have terminated due to the watchdog, or to a transmitter error (e.g. remote abort), or some error was encountered trying to establish or terminate a connection (e.g. SPX connection table full, other partner disappeared, abort connection failed).

Note on statuses 31 and 39: If an SPX session terminates immediately after it receives data, then your program may find out about both at the same time: after calling btuscn(), there will be new data in the buffer, and a status 31 or 39 in the status buffer. You may want to be sure all input data is processed before using a status 31 to reset your channel.

40 = LAN GTC Input Mode: Locked Out

The terminal should accept no input. This mode is the result of btulok(chan,1) by the other party.

Note on statuses 40-44: These occur only on IPXV or SPX LAN channels when you've issued the G command (page 67), as for a terminal emulation application. The G command is a GTC (Galacticomm Terminal Control) Greeting. It means you volunteer to preprocess terminal input for the other party. Page 224 has more on GTC.

41 = LAN GTC Input Mode: Binary

Transparent input mode — the terminal should just forward characters verbatim. This mode is the result of btutrg(chan,nonzero), or of btuchi(chan,nonnull) by the other party.

See the note under status 40 about the G command.

42 = LAN GTC Input Mode: ASCII, No Echo

Buffer a string of characters, terminate with a CR, and send the whole string without echoing anything. This mode can be used to enter passwords. It is activated by btuech(chan,0) from the other party when he's in ASCII input mode.

If a control character other than ^H=BACKSPACE or ^M=ENTER comes in, you may want to immediately send the character directly to the other party, bypassing the line input buffer. For example, with Worldgroup this would preserve some of the effects of ^O=abort and ^S=pause.

See the note under status 40 about the G command.

43 = LAN GTC Input Mode: ASCII, With Echo

Same as status 42 except echo every character. After echoing the CR, also echo a LF. The line length specifies the maximum number of pre-CR characters to accept. Extra characters should be ignored by the terminal. This is the most common, plain vanilla, input mode.

Note on statuses 43 and 44: Immediately after a status 43 (or a status 44) is another single status code indicating the maximum length of the input line (when 1 to 255) or unlimited line input (when 0). This means that the routine that services a status 43 or 44 must call btusts() exactly once and treat its result differently from all other values returned by btusts().

See the note under status 40 about the G command.

44 = LAN GTC Input Mode: ASCII, W/Echo And Wrap

Same as 43 except that when the line length limit is reached, and another character is typed:

Case 1 if the character is white-space (SPACE or CR), then just terminate the line with CR and move on;

Case 2 if the character is printable and there are previous SPACEs on the line, then just forward the line up to and before the SPACE (with a CR on it), erase the rest from the line on the terminal’s display, then move this rest down to a new input line and resume as if it had just been typed;

Case 3 if a complete line of non-white-space has been typed, then terminate and forward the line up to before the new character and make a new entry line with the character.

The line length status code (see above) should be used by the main program to wrap input words when the line would exceed this length. This mode is used for multi-line text entry.

See the note under status 40 about the G command.

49 (ASCII 1) = Xecom DTMF 1 Sensed

This code can only happen in response to a ^A (controlled answer) command byte (see page 62). It indicates that the modem module has sensed the DTMF tones for 1 (the “1” on a touch-tone phone) while the modem was waiting for originate-carrier.

63 (ASCII ?) = Xecom Invalid btucmd() Command Byte

This happens if you pass an invalid command byte to btucmd() for Xecom hardware. Look for ?’s in the next to the last column of figure 4-1 (page 57).

65 (ASCII A) = Xecom Aborted Command Prematurely

Can only happen if a command is in progress, and your program calls btuclc() for that channel, which both clears the command output buffer and aborts the currently active command, if any.

66 (ASCII B) = Xecom/X.25 Busy Signal Sensed

With Xecom hardware, can only happen in response to an L, M, or W command byte (pages 69 to 73), indicating that the modem module has sensed a phone busy signal.

With X.25 hardware, after a dial-out command, this status indicates that the call failed for some internal reason, like transmit window full.

68 (ASCII D) = Xecom Dial Tone Sensed

Can only happen in response to an L or M command byte (see page 69). Indicates that the modem module has unexpectedly sensed dial tone on the line (if, for example, a touch-tone call were placed on a pulse-dial line).

70 (ASCII F) = Xecom Failed For Other Reasons

Indicates that carrier was heard but that the originate/answer handshake failed for some reason, probably due to noise on the phone line, or an intermittent or very weak signal.

73 (ASCII I) = Xecom Inappropriate Command

This status can be returned anytime you issue a command byte that is inappropriate in the current context. For example, passing an A to btucmd() when there is already a modem-to-modem connection underway is inappropriate.

This status provides a way to distinguish between 300 and 1200 baud connections on Xecom hardware. There is no other way to do this if the baud rate was automatically determined by a calling user's carrier signal (after you issue an A or ^A command). If you issue a ^H command (page 62) after a 300 baud connection has been established, you get this status code. If you issue a ^H command after a 1200 baud connection has been established, you'll get the OK status code 2 (assuming the ^H command is by itself in the command string).

77 (ASCII M) = Modem Carrier Sensed

With Xecom hardware, can only happen in response to a W command byte (see btucmd(), page 73). Indicates the modem module has sensed answer carrier on the phone line.

With X.25 hardware, after a dial-out command, this status indicates that the call succeeded and the connection has been established.

82 (ASCII R) = Xecom Ringing Sensed, But No Answer

Can only happen in response to an L, M, or W command byte (see btucmd(), page 69). Indicates that the modem module has sensed ringing on the phone line. This happens after you place a call, while you hear the telephone company's*ring back*sound, but before the other party picks up.

84 (ASCII T) = Xecom Timeout (Silence Sensed)

Can only happen in response to a ^A, ^X, ^Y, A, L, M, O, or W command (see pages 62 to 75). Indicates the modem module timed out after 17 seconds waiting for carrier signal (W waits only 5 seconds for dial tone).

86 (ASCII V) = Xecom Voice Sensed

Can only happen in response to an L, M, or W command byte (see pages 69 to 73). Indicates the modem module has sensed a rapidly varying spectrum of activity on the phone line. This can also indicate excessive line noise, or the cable may not be connected to the telephone company.

118 (ASCII v) = Xecom Voice Sensed (By ^A Command)

This status code can only occur in response to a ^A command (see btucmd() and page 62). Otherwise it is identical to status code 86.

250 = X.25 Transmission Error

This status indicates an internal transmission error on this X.25 channel. Some data may have been lost.

251 = Data Input Circular-Buffer Overflow

More data has been received than can fit in the input buffer for this channel. Either the capacity of the input buffer was reached before the termination condition was reached (the termination condition is: CR in ASCII mode, byte count in Binary mode), or the main program has not been servicing the channel often enough — using btuinp() (for ASCII mode) or btuict() (for Binary mode). Exceeding the maxinl line length specified by btumil() will also generate status 251.

In interactive applications, this status may merely mean that the user has typed an input line that is too long, and thus the condition can be safely ignored.

252 = Echo Output Circular-Buffer Overflow

An attempt was made to buffer more data for echo than the echo buffer could hold. This status can usually be safely ignored since it indicates merely that the user has not had all of his input echoed back to him. The echo buffer can hold up to 255 bytes, so this status can happen if both the input buffer size is specified larger than 128 (see btusiz() and btulsz(), pages 152 and 117), and if the user manages to type more than 255 characters during an extended period in which echoes are locked out — for example, during the transmission of a very long, continuous block of output data. The chiout() or chious() routines (page 47) of btuchi() or bturti() can also overflow the echo buffer.

253 = Data Output Circular-Buffer Overflow

An attempt was made to buffer more data for transmission than the output buffer can hold. This can happen if your program tries to output more, or larger, blocks of data than it has allowed for in the output buffer size parameter osiz passed to btusiz() or btulsz() (pages 152 and 117). This status can only occur as a result of a call to btuxmt() (page 189) or btuxct() (page 182). There is usually little your program can do but ignore this status if it arises, but you should consider increasing your data output buffer size if it arises very often. See also the caution on btuxmt() block terminator characters in the output buffer (page 191).

254 = Status Input Circular-Buffer Overflow

This is a serious error condition, and probably merits resetting the channel if it occurs. It indicates that status information has been lost, which means that you may have a false sense of the condition of the channel. For example, you may have missed a lost-carrier status (1 or 11). Fortunately, as the status buffer can contain up to 31 bytes, it is almost impossible for this status to crop up in peace-time conditions.

255 = Command Output Circular-Buffer Overflow

This is a serious error condition, and probably merits resetting the channel (if not the entire machine) if it occurs. You can buffer up to 63 bytes of commands before getting this status, which is far more than most programs should need. The most likely cause of this status is a limited program crash of some sort: some unrelated piece of software corrupted the channel data blocks.

-10 = Channel Not Defined

This status can only happen once after a call to btudef(). It indicates that the channel hardware did not respond properly to a reset attempt, which means that it is either malfunctioning or absent altogether. Yet another possibility is that you have purchased an N-channel version of the Software Breakthrough, and you have attempted to define channel N or higher. In this case, all channels except 0 through N-1 will always appear to have no hardware. These channels may still be used for local emulation however (refer to page 126).

-11 = Channel Number Out Of Range

Indicates that the chan parameter passed to btusts() itself is outside the range of valid channel numbers, as defined in the original call to btusiz() or btulsz() (pages 152 and 117).

CAUTIONS

Most probably, you will not have to deal with each and every one of these status codes whenever you call btusts(). In any given situation there are only a few that you should expect. See figure 4-1 (page 57) for a summary of what commands can be expected to directly generate which status codes. Other status codes are asynchronous, of course (ringing, lost carrier, input data received, etc). Any unexpected status codes should be handled by resetting the channel (bturst(), page 138).

Up to 31 bytes of status can be queued, per channel, before the status queue overflows. See the note on status 43 and 44 (page 160) about how btusts() follows these status return codes with a status that could be anywhere between 0 and 255.

Figure 4-2: Summary of btusts() Status Codes

Status Code	ASCII	Description
0		Quiet, Nothing Special To Report
1		RING-Indicate Or Lost-Carrier (Xecom)
2		Command Execution Completed OK (Xecom)
3		CR-Terminated Input String Available
4		Byte-Count-Triggered Input Data Available
5		Output Buffer Empty
6		Output Aborted By User
7		(Reserved For Screen-Pause/Quit Command)
11		Hayes Lost Carrier
12		Hayes/UART Command Execution Completed
13		Hayes/UART Invalid `btucmd()` Command Byte
21		X.25 Incoming Clear Packet (End Of Session)
22		X.25 Command Or Pause Completed
23		X.25 Invalid `btucmd()` Command Byte
24		X.25 Incoming X.29 String
31		LAN SPX Connection Terminated By Other Side
32		LAN Pause Command Completed
33		LAN Invalid `btucmd()` Command Byte
34		LAN SPX Incoming Connection Established
35		LAN SPX Outgoing Connection Established
36		LAN SPX Termination Complete By This Side
37		LAN Receiver Error
38		LAN Received Unknown Or Unexpected Packet
39		LAN Connection Error
40		LAN GTC Input Mode: Locked Out
41		LAN GTC Input Mode: Binary
42		LAN GTC Input Mode: ASCII, No Echo
43		LAN GTC Input Mode: ASCII, With Echo
44		LAN GTC Input Mode: ASCII, With Echo And Wrap
49	1	Xecom DTMF 1 Sensed
63	?	Xecom Invalid Command Byte
65	A	Xecom Aborted Command Prematurely
66	B	Xecom Busy Signal Sensed (Or X.25 Call Failed)
68	D	Xecom Dial Tone Sensed
70	F	Xecom Failed For Other Reasons
73	I	Xecom Inappropriate Command
77	M	Xecom Modem Carrier (Or X.25 Call Completed)
82	R	Xecom Ringing Sensed, But No Answer
84	T	Xecom Timeout (Silence Sensed)
86	V	Xecom Voice Sensed
118	v	Xecom Voice Sensed (by `^A` Command)
200-249		Reserved For Your Special Purposes Using `btuinj()` or `chiinj()`
250		Transmission Error (X.25)
251		Data Input Circular-Buffer Overflow
252		Echo Output Circular-Buffer Overflow
253		Data Output Circular-Buffer Overflow
254		Status Input Circular-Buffer Overflow
255		Command Output Circular-Buffer Overflow
-10		Channel Not Defined
-11		Channel Number Out Of Range

btutrg

SUBROUTINE NAME

btutrg — set input byte trigger quantity

SYNOPSIS

err=btutrg(chan,nbyt);		
int err;	zero means OK	
int chan;	channel number	
int nbyt;	input byte trigger quantity:	
	> 0	for Binary input mode, bytes per block (use btuict() or btuica() for input)
	= 0	for ASCII input mode (use btuinp() for input)

DESCRIPTION

If nbyt is nonzero, input characters will be processed in transparent Binary mode. There is no translation through the global input-character translation table, no BACKSPACE handling, and no CR-triggered end-of-line status handling (echoing is also automatically disabled when in transparent mode). Instead, characters are simply accepted as they come in and, when the trigger count of nbyt has been reached, a status of 4 (BYTE-COUNT-TRIGGERED INPUT DATA AVAILABLE) is generated for the channel. Each group of nbyt input characters generates this same status.

If nbyt is zero, then input is in ASCII mode (the default). See page 12 for more details on Binary versus ASCII input modes.

CAUTIONS

This routine should only be used in special situations such as machine-to-machine communications (XMODEM, etc.), or to capture individual user keystrokes in real time (nbyt=1). For the latter, consider a custom character interceptor routine. See btuchi() on page 44.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range

	(see btusiz(), page 152 or btulsz(), page 117)
0	all is well

btutrm

SUBROUTINE NAME

btutrm — set input line terminator character

SYNOPSIS

```
err=btutrm(chan,crchar);
int err;                        zero means OK
int chan;                       channel number
char crchar;                    input line terminator (0 to disable)
```

DESCRIPTION

The input line terminator character is the character which, when typed at an interactive user's console, indicates that all editing of the input line is complete, and that it is time for your program to take the line as a whole and perform whatever processing it may require. The default line-terminator is a carriage return (ASCII code 13), but there may be occasions when you need to set it to something else. This routine gives you that option.

CAUTIONS

You will want to use this routine only in very unusual circumstances.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	all is well

btutrs

SUBROUTINE NAME

btutrs — generate status 6 when output aborted by user?

SYNOPSIS

err=btutrs(chan,onoff);	
int err;	zero means OK
int chan;	channel number
int onoff;	1=generate 0=don’t generate (bturst() defaults to 0)

DESCRIPTION

In ASCII output mode (page 15), any data block in the process of being transmitted to a channel is aborted if the character specified in btutru() is received from the channel. To notify your mainline program, btutrs(chan,1) will trigger a status 6 whenever the user aborts output in this manner.

See page 171 for more on output abort.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	all is well

btutru

SUBROUTINE NAME

btutru — set output-abort character (truncates current output block)

SYNOPSIS

err=btutru(chan,trunch);	
int err;	zero means OK
int chan;	channel number
char trunch;	truncate character (0 to disable)

DESCRIPTION

In ASCII output mode (page 15), any data block in the process of being transmitted to a channel is aborted if the character you specify here is received from the channel. On DEC systems, the character that users are accustomed to for this purpose is CTRL+O. This is what we have used for Worldgroup. On other systems, CTRL+X is preferred.

Only the block currently being transmitted is truncated, or aborted — the rest of the output buffer is left alone. This way, it is less likely that users will inadvertently abort asynchronous messages such as Sysop alerts or messages from other users, or their next prompt. Each separate call of btuxmt() (see page 189) generates a different output block which is individually clobbered by the abort character.

This feature is disabled by default, since its use is potentially harmful. Be sure to educate your users about the proper use of output block truncation if you enable it in your particular system.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	all is well

btutsw

SUBROUTINE NAME

btutsw — set terminal screen width and select output word wrap

SYNOPSIS

err=btutsw(chan,width);			
int err;	zero means OK		
int chan;	channel number		
int width;	> 0	turn on output word wrap, where width is the screen width	
	= 0	turn off output word wrap (default condition)	

DESCRIPTION

This routine turns on the output word wrap feature, specifying the screen width to which text output should conform during ASCII output (page 15). Specifying a screen width of 0 turns off output word wrap.

First, we will describe output word wrap and the effects of related Software Breakthrough routines. Then we will describe the default situation (what happens when you call none of these routines). Finally, we will present an example.

How Output Word Wrap Works

The output word wrap feature prevents words (strings of non-SPACE characters) from being split across line boundaries. This means, for example, that 80-column wide messages will be readable both to users with 80-column wide terminals, and to users with 40-column wide terminals.

This is achieved by translating SPACEs into carriage returns in the output data string specified by btuxmt() (page 189). To prevent the user's terminal from doing its own line wrapping, each line is actually limited to width-1 characters.

btutsw() is the primary controller of output word wrap, but it operates in concert with other Software Breakthrough routines:

Routine	Involvement with output word wrap	Page
`btuhcr()`	specifies the *hard* carriage return character	92
`btuscr()`	specifies the *soft* carriage return character	146
`btulfd()`	specifies the linefeed character to append to every carriage return	114
`btuxmt()`	specifies the output string, and actually performs the output word wrap processing	189

The hard and soft carriage return characters are special characters in your output data string that affect the operation of output word wrap.

Hard carriage returns are always interpreted as the end of the line. They are translated into the ASCII CR character, which is appended with LF if btulfd() has so specified. Conceptually, hard carriage returns form paragraph boundaries.

Soft carriage returns may be interpreted as a single space, or as a carriage return, depending upon where they are positioned in the paragraph. The rule is this:

> When output word wrap has taken place in a paragraph, all subsequent soft carriage returns are converted into spaces.

Word wrap means that a line has been shortened to conform to the screen width by changing one of its spaces into a soft carriage return. Paragraphs are defined as strings of characters between hard carriage returns. Note that conversion of a soft carriage return into a space does not preclude its reconversion into a carriage return by the word wrap procedure. In this manner, wide paragraphs are completely reflowed to fit the available screen width. Note, however, that narrow paragraphs are not reflowed to fit wide screens.

Default Conditions

After a channel is reset by bturst(): the output word wrap feature is turned off; the hard carriage return is 13 (ASCII CR); and there is no soft carriage return. The linefeed character is 10 (ASCII LF). In this situation, when transmitting messages, the line terminator should be the ASCII CR character:

```
btuxmt(chan,"This is one line of text.\r");
```

The \r is the C-language representation for ASCII CR. Since this is the hard carriage return character, it is translated into ASCII CR (that is, it is not changed). It is appended with ASCII LF.

Example of Output Word Wrap

The following conditions are similar to those in Worldgroup (although screen width is user-selectable):

```
btutsw(chan,20);
btuhcr(chan,13);
btuscr(chan,10);
```

Now, hard carriage returns are ASCII CRs and ASCII LF is the soft carriage return. The terminal screen width is limited to 20 characters for simplicity. Now let’s suppose that the following statements were executed:

```
btuxmt(chan,"The blue form of the Engelmann Spruce\n");
btuxmt(chan,"is native to the mountains of western\n");
btuxmt(chan,"North America.\n");
btuxmt(chan,"\r");
btuxmt(chan,"Koyama's Spruce is native to central\n");
btuxmt(chan,"Japan (at altitudes of 1500 to 1800\n");
btuxmt(chan,"meters) and also to Korea.\n");
btuxmt(chan,"\r");
```

The result *with* word wrap on the terminal of channel chan would be:

```
The blue form of
the Engelmann
Spruce is native to
the mountains of
western North
America.

Koyama's Spruce is
native to central
Japan (at altitudes
of 1500 to 1800
meters) and also to
Korea.
```

With*out* turning on word wrap, this hypothetical 20-column wide screen would look something like:

```
The blue form of the
 Engelmann Spruce is
 native to the mount
ains of western Nort
h America.

Koyama's Spruce is n
ative to central Jap
an (at altitudes of
1500 to 1800 meters)
 and also to Korea.
```

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	all is well

btuudf

SUBROUTINE NAME

btuudf — un-define channels

SYNOPSIS

```
err=btuudf(schan,n);
int err;                  zero means OK
int schan;                starting channel number
int n;                    number of ports in this group
```

DESCRIPTION

This command undoes the effects of btudef() (page 80). This capability is provided in case you need to completely remove a channel, or group of channels, from service without affecting the operation of any other channels. In most cases this will not be necessary, but it may come in handy if you are doing something exotic. Many of the other Software Breakthrough routines will return a code of -10 on a channel that has been un-defined.

CAUTIONS

LAN X.25

There is no btuudf() (un-define) for LAN or X.25 channels. Defining a LAN or X.25 channel (calling btusdf() with chtype from 4 to 7) is permanent and can occur only once for a given channel over the run-time life of your program.

RETURNS

-11	channel number(s) out of range: the specified range of channels (schan to schan+n-1) is not within the inclusive range 0 to nchan-1, where nchan has been defined in btusiz() (page 152) or btulsz() (page 117)
0	the specified channels were un-defined successfully

btuusp

SUBROUTINE NAME

btuusp — special UART polling method

SYNOPSIS

```
err=btuusp(chan,onoff);
int err;                    zero means OK
int chan;                   channel number
int onoff;                  1=special method (IIR bit 0)
                            0=standard method (LSR bit 0)
```

DESCRIPTION

UART

This function only has an effect on 8250-type UART channels. It would most likely be used on 16550 UARTs (see btuffo() on page 90). The standard method (onoff=0) for polling UARTs is to disable interrupts and poll the Data Ready bit (bit 0) of the Line Status Register (offset 5). The special method (onoff=1) is to turn off OUT2 (MCR bit 3 is 0), enable received-data interrupts (IER bit 0 is 1), and to poll the Interrupt Pending bit (bit 0) of the Interrupt Identification Register (offset 2). No interrupts are actually received by the processor because OUT2 is turned off, which by IBM PC convention is used to gate the UART interrupt onto the bus. The special method is more reliable for some brands of UART, specially on fast computers, due to a sluggishness of the Data Ready bit to turn off after the last byte is read from the 16550's hardware FIFO. Compared with the Data Ready bit, the Interrupt Pending bit is the reverse sense: 0=interrupt pending=data available, 1=no interrupts=input FIFO empty.

CAUTIONS

UART

The special method may not work on some non-standard types of UART or non-standard serial hardware (though we haven't seen any yet).

HAYES XECOM
LAN X.25

Do not use this function on non-8250 ports. Such operation is undefined.

RETURNS

UART

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	OK, special mode turned off
1	OK, special mode turned on

btux29

SUBROUTINE NAME

btux29 — transmit an X.29 string across an X.25 network
to the remote user's PAD

SYNOPSIS

err=btux29(chan,nbyt,datstg);	
int err;	zero means OK
int chan;	channel number
int nbyt;	number of bytes
char *datstg;	data bytes, formatted X.29 string

DESCRIPTION

The source-code syntax of btux29() is identical to that of btuxct(), but there are two differences in function:

- btux29() transmits the message in a packet with the "Q" bit set, per X.29.
- btuxct() puts data into a circular buffer and transmits when the hardware says it's ready. btux29() attempts to transmit immediately, and if the hardware refuses, btux29() returns the 0x500 hex status code, in which case you will probably want to try and transmit again later.

The X.29 message programs one or more parameters in the user's PAD. The message is coded as follows:

byte 0	message code (2=set)
byte 1	number of first parameter
byte 2	value for first parameter
byte 3	number of second parameter
byte 4	value for second parameter
byte 5	...and so on...

It is possible to use other message codes (4=read, 6=read and set, etc.) that cause the PAD to respond back with messages that have the "Q" bit set. The GSBL does not process incoming X.29 messages. These messages will be received just like any other data from the user (the "Q" bit is ignored). If you want to receive them, use btuict() or btuica(), and be sure to select binary input mode (using btutrg()).

CCITT recommendation X.3 defines the numbering and meaning of PAD parameters. Many PAD manufacturers define their own extensions to X.3 to support specific features. Here is a sample of the parameters as used by OST's Europad III (unless otherwise mentioned, 0=off and 1=on):

1 Escape from data transfer (for user to enter X.28 commands) (0=disabled 1=enabled for CTRL+P)
2 Echo (0=off 1=on)
3 Transmit when (0=full packet, 2=after CR, 126=after a control character)
4 Transmit when (0=full packet, n=after n/20 seconds of no data from user)
5 Enable PAD to XON/XOFF throttle user transmissions (or CTS throttle if on-standard "FWC" PAD option is set to Yes)
6 Display PAD service signals (indicating call progress)
7 Handling of Break-detect (0=off, otherwise see PAD manual)
8 Discard data output
9 Padding after CR (0=none, n=append n NUL bytes)
10 Line folding (0=none, n=automatic CR after n characters on line)
11 Line speed, in bits per second:

0 = 110 bps	4 = 600 bps	10 = 50 bps	14 = 9600 bps
1 = 134 bps	5 = 75 bps	11 = 75/1200 bps	31 = 1200/75 bps
2 = 300 bps	6 = 150 bps	12 = 2400 bps	32 = 3600 bps
3 = 1200 bps	7 = 1800 bps	13 = 4800 bps	33 = 7200 bps

12 Enable PAD to respond to XON/XOFF throttling by the user
13 Linefeed insertion (0=none, 4=echo LF after echoing CR)
14 Padding after LF (0=none, n=append n NUL bytes)
15 Local user editing of each transmitted line
16 Character-Delete character (0=none, or ASCII value, e.g. 8=BACKSPACE)
17 Line-Delete character (0=none, or ASCII value, e.g. 24=CTRL+X)
18 Line-Display character (0=none, or ASCII value, e.g. 18=CTRL+R)

RETURNS

-10 channel is not defined (see btudef(), page 80)
-11 channel number is out of range
(see btusiz(), page 152 or btulsz(), page 117)
-13 X.25 interface not available (only returned when you are using GSBL without the X.25 Software Option)

0x500	(hexadecimal) transmit window full - try again later
0	channel number is good

all other return values indicate some error returned by the DATASD transmit function of the PC XNet driver. Refer to the documentation from OST (specifically, the PCXNET.H source file).

btuxct

SUBROUTINE NAME

btuxct — transmit to channel (by byte count)

SYNOPSIS

```
err=btuxct(chan,nbyt,datstg);
int err;                        zero means OK
int chan;                       channel number
int nbyt;                       number of bytes to send
char *datstg;                   data block to be sent
```

DESCRIPTION

This routine transmits a data block to the specified channel in Binary output mode (see btuxmt(), page 189 for ASCII output mode). There are no restrictions on the length of the block, as long as there is room for it in the output data buffer. The block may contain any data, including zeros. Each byte will be transmitted when its turn comes.

The btuxct() routine does not have the output-suspended-while-inputting feature of btuxmt() (page 189). If you need to use btuxmt() but want to disable this suspension feature, then after using btuxmt() you might code:

```
btuxct(chan,0,"");
```

CAUTIONS

If there is not enough room in the output buffer for the block, btuxct() still returns 0 but a status of 253 (DATA OUTPUT CIRCULAR-BUFFER OVERFLOW) is queued for btusts(), and *none* of the data block is output. If your program needs to know right away whether the data will fit, use btuoba() (page 132) ahead of time to find out how much space is available in the transmit buffer.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range

(see btusiz(), page 152 or btulsz(), page 117)

0 all is well

btuxlt

SUBROUTINE NAME

btuxlt — set input translation table

SYNOPSIS

```
btuxlt(oldchr,newchr);
char oldchr;                character to be translated
char newchr;                character to translate into (0 to ignore)
```

DESCRIPTION

The global input-character translation table applies to all channels in ASCII input mode (page 12) where you have not installed a custom character interceptor using btuchi() (page 44). As each character is received from the outside world, its ASCII value is used as an index into the translation table. If the resulting lookup value is zero, the character is ignored. Otherwise, the lookup value is passed on to the main program as if it had been received in place of the original (indexing) character.

By default, BACKSPACE (ASCII 8) and carriage return (ASCII 13, ENTER) are translated as themselves. All other control characters are ignored. All printable characters (ASCII 32 through 126) are translated as themselves. The RUBOUT character (ASCII 127) is translated to BACKSPACE since many older terminals do not have a backspace key. Characters above ASCII 127 are translated as if below 127 (the high-order 8th bit is always translated to 0). Parity, framing, and overrun errors are ignored.

The following chart shows the 256 table values in hexadecimal. To use the chart, start with the raw input character to be translated, expressed as a hexadecimal number. Use the first hex digit to select a row of the chart, then use the second to select a column. The number found at the intersection of the row and the column is what your original raw input character will translate to.

btuxlt() lets you change the translation table one character at a time.

Default Translation Table

		2nd digit															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	00	00	00	00	00	00	00	00	08	00	00	00	00	0D	00	00
	1	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	2	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
1	3	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F
s	4	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
t	5	50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F
	6	60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F
d	7	70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	08
i	8	00	00	00	00	00	00	00	00	08	00	00	00	00	0D	00	00
g	9	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
i	A	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
t	B	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F
	C	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
	D	50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F
	E	60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F
	F	70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	08

The translation table serves several purposes. Different people have different ideas as to what ASCII control characters they do or do not like. By default, only BACKSPACE (8) and carriage return (13, ENTER) are accepted and the translation table entries for all other characters in the range from 0 to 31 are zero. If you like BEL (ASCII 7), you could write:

```
btuxlt(7,7);
```

This changes the translation table to permit the bell character (CTRL+G), when typed at a user's keyboard, to be passed through to your main program normally just like any ordinary printable ASCII character.

Another purpose of the translation table is to permit you to translate UPPER case to lower case, or vice versa. This may improve the efficiency of the application software, in that you will not need to consider case when comparing strings, storing them in a database, etc. The following simple loop sets things up so that all channels in the ASCII mode automatically demote capitals to lower case:

```
for (i='A' ; i <= 'Z' ; i++){
     btuxlt(i,i-'A'+'a');
}
```

A third use of the translation table is to to deal with parity, overrun, and framing errors. With “error-passthru” (see btuerp(), page 88), enabled by default, bytes read in with these errors will be received and their high bits will be set. Thus they will form translation table indices in the range from 128 to 255, as opposed to proper ASCII characters, which will have values of 0 to 127. There are, then, four main ways you might elect to deal with input characters involving these errors, by setting the upper half of the global translate table:

1. Reject them: use btuxlt() to set all index values in the range 128-255 to zero, thereby causing erroneous input to be completely ignored.
2. Blot them out: use btuxlt() to translate all bytes in the range 128-255 to some nonzero “error” character, such as BEL (7), or X (88).
3. Tag them: use btuxlt() to cause values in the range 128-255 to be passed through unchanged. In this way, you defer the decision as to what to do about them to your main program. If it wants to accept or reject them in a context or user-dependent way, it may do so.
4. Accept them: the translation table defaults to this arrangement, whereby values in the range 128-255 effectively have 128 subtracted from them (in other words, their high bit is cleared). Accepting these characters means ignoring the error — this is the default because many interactive applications do not want to insist that the caller have the right parity.

CAUTIONS

Any changes made to the translation table apply instantly to the system as a whole. Unless you are doing something unusual, this means that your btuxlt() calls, if any, should all appear at the beginning of your program, just after your call to btuitz().

RETURNS

Nothing.

btuxmn

SUBROUTINE NAME

btuxmn — transmit ASCII string that btuclo() will not be able to clear. (btubsz() will clear, however.)

SYNOPSIS

err=btuxmn(chan,outstg);	
int err;	zero means OK
int chan;	channel number
char *outstg;	string to send (NUL-terminated)

DESCRIPTION

This transmits a character string to a channel in the ASCII output mode (page 15). There must be room for the string in the output buffer. The string may contain any number of line terminators (see below).

btuxmn() is almost identical to btuxmt() (page 189), but btuxmn() puts strings in the output buffer that btuclo() cannot clear. btuxmn() can transmit a message to a user that he won't be able to skip or abort.

CAUTIONS

If there is not enough room in the output buffer for the string, btuxmn() returns 0, but a status 253 (DATA OUTPUT CIRCULAR-BUFFER OVERFLOW) is queued for btusts() and the data string will not be output. If your program needs to know right away whether data will fit, use btuoba() (page 132) to find out first how much space is in the output data buffer. Like btuxmt(), each call to btuxmn() puts a new block into the output buffer, terminated by a hex 01 byte (CTRL+A), versus btuxmt()'s hex 00. btuxmn() temporarily writes a \x01 byte over the terminating \x00 NUL, then restores the NUL, so the outstg location must be writable.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range

(see btusiz(), page 152 or btulsz(), page 117)

0 channel number is good

btuxmt

SUBROUTINE NAME

btuxmt — transmit to channel (ASCIIZ string)

SYNOPSIS

err=btuxmt(chan,outstg);

int err;	zero means OK
int chan;	channel number
char *outstg;	string to send (NUL-terminated)

DESCRIPTION

This transmits a character string to a channel in ASCII output mode (page 15). There must be room for the string in the output buffer. The string may contain any number of line terminators (see below). btuxmt() is the primary way to transmit data to a user during an interactive session.

If the user is typing an input line when btuxmt() is called, output is postponed until he clears the line by pressing ENTER or by backspacing to the beginning of the line. Once output begins, anything the user types while output is still in progress is not echoed until output of the current block completes. This asynchronous input and output makes for a very readable user display, especially during teleconferencing and the like. If this feature is undesirable, use btuxct() (page 182). The following routines control operation of btuxmt() by selecting various ASCII output modes:

btutsw()	Output word wrap
btuhcr()	Hard carriage-returns
btuscr()	Soft carriage-returns
btulfd()	Linefeeds appended to carriage returns
btutru()	Output abort character
btuxnf()	XON/XOFF handshaking
btupmt()	Prompt characters

You will probably want to terminate each line in your data string with ASCII CR (\r in C notation), the *hard* carriage return. For transmitting a paragraph of text, you may want to use the *soft* carriage return, ASCII LF

(\n, see page 172). For prompting, you might have no terminator character at the end of your data string.

If your users have ANSI graphics capabilities, you may transmit these escape sequences to them using btuxmt() (use no spaces):

ESC [<row>;<column>H	Move cursor to <row>,<column>
ESC [<row>;<column>f	Move cursor to <row>,<column>
ESC [<nrows>A	Move up <nrows> rows
ESC [<nrows>B	Move down <nrows> rows
ESC [<ncols>C	Move forward <ncols> columns
ESC [<ncols>D	Move backward <ncols> columns
ESC [s	Save cursor position
ESC [u	Restore cursor position
ESC [2J	Erase display
ESC [K	Erase to end of current line

Display Attributes

ESC [0m	Normal
ESC [1m	Bold
ESC [4m	Underscore
ESC [5m	Blink
ESC [7m	Reverse
ESC [8m	Invisible

Note: m directives may be combined. For example, to select blinking black on blue, you could code: btuxmt(chan,"\x1B[5;30;44m");

Foreground Color	**Background Color**
ESC [30m Black	ESC [40m Black
ESC [31m Red	ESC [41m Red
ESC [32m Green	ESC [42m Green
ESC [33m Yellow	ESC [43m Yellow
ESC [34m Blue	ESC [44m Blue
ESC [35m Magenta	ESC [45m Magenta
ESC [36m Cyan	ESC [46m Cyan
ESC [37m White	ESC [47m White

The following special IF-ANSI construct interacts with the [and] commands of btucmd():

ESC [[<for-ANSI-users>¦<for-non-ANSI-users>]

Only one of these strings will be transmitted, depending on which of the [or] btucmd() commands was last issued for the channel (see page 77). There is more to these commands: the] command disables ANSI graphics and, if in effect, btuxmt() will not transmit any of the above directives.

The tilde (~) character is used as an escape character by btuxmt(). The following character pairs may be used to avoid conflicts with the IF-ANSI construct:

~¦ represents a single ¦ (vertical bar)
~] represents a single] (closed bracket)
~~ represents a single ~ (tilde)

CAUTIONS

If there is not enough room in the output buffer for the string, btuxmt() returns 0, but a status 253 (DATA OUTPUT CIRCULAR-BUFFER OVERFLOW) is queued for btusts() and the data string will not be output. If your program needs to know right away whether data will fit, use btuoba() (page 132) to find out first how much space is in the output data buffer.

Each call to btuxmt() puts a new block into the data output buffer. A block is the sequence of characters pointed to by the datstg parameter, and is terminated by a 0-byte. btuxmt() stores the 0-byte in the output buffer to determine block boundaries. This may become critical with regard to output buffer capacity if you are doing numerous small transmissions with btuxmt(). The alternative is to use btuxct() (page 182), which does not put termination characters in the output buffer.

There is a razor's edge situation in which the priority of transmission over command execution must be considered:

```
btucmd(chan,"p")
btuxmt(chan,"Before or after pause?\r");
```

In this example, you can't be sure the command will be executed before the string is transmitted because the interrupt-level code, when presented with commands and transmissions simultaneously, chooses transmissions. How can this be? The service rate (page 129) is low enough that, most probably, no service cycle will come between the above two instructions.

If this occurs, the effects of these two statements will appear to the interrupt-level code simultaneously, and the data transmission will come before the pause.

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	channel number is good

btuxnf

SUBROUTINE NAME

btuxnf — set XON/XOFF characters, select page mode

SYNOPSIS

```
err=btuxnf(chan,xon,xoff);              (for non-page mode)
                                  -or-
err=btuxnf(chan,xon,-xoff,cnt,stg);     (for     page mode)

int err;                          zero means OK
int chan;                         channel number
int xon;                          character taken as XON, or “resume”
                                  (0 to take anything)
int xoff;                         character taken as XOFF, or “suspend”
                                  (0 to disable, negative for page mode)
int cnt;                          number of lines on screen (page mode only)
char *stg;                        page-break pause message (page mode only)
```

DESCRIPTION

This feature is only in effect during the ASCII output mode (page 15). A user can suspend text output to his terminal by sending the xoff character that you specify here. The xon character will cause output to resume. Make the xon parameter 0 if you want any character to resume output. Make the xoff parameter 0 to disable XON/XOFF altogether.

The page mode feature of ASCII output mode automatically pauses output between each screen of text. This gives the user a chance to review one screen before typing a key to move on to the next screen. The user resumes output the same way as after an XOFF pause: he types the xon character (if the xon parameter is nonzero), or he types any character (if xon is 0).

To select page mode, simply substitute for the xoff parameter its negative value (for example -19 instead of 19). In this case, you must also pass the cnt and stg parameters to btuxnf() (see the second line of the synopsis above). cnt is the total number of lines on the user’s screen. stg is a pointer to the message to display between each screen (a 0-terminated character string).

The default values after initialization or reset operate just as if the following call had been made:

```
btuxnf(chan,0,19);
```

This means:

chan	For this channel number,
xon = 0	any character resumes after a pause
xoff = 19	CTRL+S pauses, page mode is *not* selected.

Here is an example of how you would enable page mode:

```
btuxnf(chan,0,-19,24,"Strike any key to continue...");
```

This means:

chan	For this channel number,
xon = 0	Any character resumes after a pause
xoff = -19	CTRL+S pauses, page mode *is* selected
cnt = 24	The user's screen has 24 lines
stg points to:	"Strike any key to continue..." is shown at the end of each block of 22 lines

CAUTIONS

The Software Breakthrough cannot count wrap-arounds and line-breaks enforced by the user's terminal, so be sure that the width of the user's screen has been properly set by btutsw() (page 172).

RETURNS

-10	channel is not defined (see btudef(), page 80)
-11	channel number is out of range (see btusiz(), page 152 or btulsz(), page 117)
0	all is well

Programming Examples

The two programs in this section were written in C. They illustrate the use of the most important Software Breakthrough routines. One program runs on Hayes category hardware (AT command set modems), and the other on Xecom category hardware like the Galacticomm Models 16 and 4 Breakthrough cards. See page 5 for a description of hardware categories. Each program is a complete multiuser teleconferencing system, albeit a simple one. Except for the hardware differences, the two programs function the same.

The Hayes program, page 197, with line numbers 1 to 96, are explained line-by-line starting on page 199. The lines of the Xecom program, page 207, with line numbers 101 to 155, are explained starting on page 208.

When a user calls up this teleconferencing system with a modem and a terminal, modem communication is established and the user is asked to enter his name. He is then placed online with the other users. While online, anything that one user types will appear on the displays of all others who are also online. All input and output is line-buffered, so that messages do not clobber one another.

For example, someone dialing up this teleconferencing system might see the following on his display (what the user types is shown in boldface):

```
To log on, please enter your name: Farkel
Okay, you're online.
>Hello out there
*** Message sent ***
>***
From Ferd: Hello, Farkel, how's the wife?
>why, Fanny's fine, Ferd.
*** Message sent ***
>***
From Ferd: Ah, good, good, and the children?
>Flora has the flu, but Simon and Gar Farkel are feeling
fine
*** Message sent ***
>
```

Teleconference for Hayes Hardware (TELCONH.C)

```
int chan,status,state[8],i;
char name[8][21],ibf[128],obf[163],*malloc();

main()
{
     printf("TELECONFERENCE DEMONSTRATION -- HIT ANY KEY TO STOP\n");
     btuitz(malloc(btusiz(8,128,1024)));
     btudef(0,0x2F0,8);
     for (chan=0 ; chan < 8 ; chan++) {
          rstchn(chan);
     }
     while (!kbhit()) {
          if ((chan=btuscn()) >= 0) {
               status=btusts(chan);
               switch (state[chan]) {
               case 0:   /*--- CHANNEL STATE 0:  Initialized condition ---*/
                    if (status == 5) {
                        btulok(chan,0);
                        btuoes(chan,0);
                    }
                    else if (status == 3) {
                        btuinp(chan,ibf);
                        if (strcmp(ibf,"CONNECT") == 0) {
                             btubrt(chan,300);
                        }
                        else if (strcmp(ibf,"CONNECT 1200") == 0) {
                             btubrt(chan,1200);
                        }
                        else if (strcmp(ibf,"CONNECT 2400") == 0) {
                             btubrt(chan,2400);
                        }
                        else {
                             break;
                        }
                        btucmd(chan,"p");
                        state[chan]=1;
                    }
                    break;
               case 1:   /*-- CHANNEL STATE 1:  Waiting for 2 sec pause --*/
                    if (status == 12) {
                        btuxmt(chan,"\rTo log on, please enter your name: ");
                        btucli(chan);
                        btuech(chan,1);
                        state[chan]=2;
                    }
                    else {
                        rstchn(chan);
                    }
                    break;
               case 2:   /*--- CHANNEL STATE 2:  Waiting for name --------*/
                    if (status == 3) {
                         btuinp(chan,ibf);
                         sprintf(name[chan],"%1.20s",ibf);
                         btuxmt(chan,"Okay, you're online!\r>");
                         state[chan]=3;
                    }
                    else if (status != 251) {
                         rstchn(chan);
                    }
                    break;
               case 3:   /*--- CHANNEL STATE 3:  Online ------------------*/
                    if (status == 3) {
                         btuinp(chan,ibf);
                         sprintf(obf,"***\rFrom %s: %s\r>",name[chan],ibf);
                         for (i=0 ; i < 8 ; i++) {
                              if (i != chan && state[i] == 3){
                                   btuxmt(i,obf);
                              }
                         }
                         btuxmt(chan,"*** Message sent ***\r>");
                    }
                    else if (status != 251) {
                         rstchn(chan);
                    }
                    break;
               }
          }
     }
     printf("TELECONFERENCE DEMONSTRATION OVER, RETURNING TO DOS\n");
     for (chan=0 ; chan < 8 ; chan++) {
          bturst(chan);
     }
     btuend();
}

rstchn(chan)        /* Reset Channel */
int chan;
{
     bturst(chan);
     btulok(chan,1);
     btuoes(chan,1);
     btuech(chan,0);
     btuxmt(chan,"ATE0S0=1S2=1&C1&D2\r");
     btucli(chan);
     state[chan]=0;   /* Now we wait for a status 5 on this channel */
}
```

Line-by-Line Synopsis

```
1    int chan,status,state[8],i;
```

Declare the integer variables:

chan	channel or user number
status	current status code
state[]	channel-specific state code: 0 waiting for incoming call 1 waiting for 2 seconds after carrier is detected 2 waiting for user to type in his name 3 online
i	scratch variable, used to count through the other channels

```
2    char name[8][21],ibf[128],obf[163],*malloc();
```

Declare several character variables:

name[][]	array of names for each user
ibf[]	input buffer (one user at a time)
obf[]	output buffer (one user at a time)
*malloc()	the standard memory allocation utility (it returns a pointer to character)

```
3
4    main()
```

Define the “main” function, the main part of the program.

```
5    {
6         printf("TELECONFERENCE DEMONSTRATION . . .\n");
```

Greeting for the system operator.

```
7    btuitz(malloc(btusiz(8,128,1024)));
```

btusiz() specifies the buffer sizes (input 128, output 1024), and the total number of channels (8). It also computes the total number of bytes required for the Software Breakthrough. Note that in this example, the total memory required will not be anywhere near 65535 bytes, so that we do not use btulsz(), nor check for an erroneous return value from btusiz().

malloc() The standard dynamic memory allocation utility.

btuitz() Initializes the Software Breakthrough and “formats” its data structures in the memory block allocated by malloc().

```
8    btudef(0,0x2F0,8);
```

Define 8 channels (numbered 0 to 7) with an I/O base address of 2F0 hex. This works for a Galacticomm Breakthrough Model 2408 card. For a single Hayes-compatible modem on the COM1 serial port, change this to btudef(0,0x3F8,1). Of course, you should also change the number of channels from 8 to 1, which is hard-coded throughout the program.

```
 9    for (chan=0 ; chan < 8 ; chan++) {
10         rstchn(chan);
11    }
```

All channels must be reset and subjected to the initialization sequence. This sequence is performed by function rstchn(), coded on lines 86 to 96.

```
12    while (!kbhit()) {
```

The main loop of the program continues until the system operator presses a key on the main console keyboard.

```
13    if ((chan=btuscn()) >= 0) {
```

The first channel number requiring service (if any) is put into chan.

```
14    status=btusts(chan);
```

The variable status now contains the status code of the channel that requires service.

```
15    switch (state[chan]) {
```

Now, let's treat each of the channel states separately. The state codes 0-3 are described above, with the explanation of line 1.

```
16    case 0:   /*--- CHANNEL STATE 0:  Initialized ---*/
```

Two possibilities: (A) status 5, meaning that the Hayes-protocol command in rstchn() has been sent to the modem and we need to complete the init sequence; or (B) status 3, indicating that the modem has sent us a message (RING, OK, NO CARRIER, CONNECT), etc. We are only interested in CONNECT.

```
17    if (status == 5) {
```

This is the status 5 that should eventually result from the btuxmt() call on line 93 in the rstchn() function, below. It means that the command string on that line has been completely transmitted to the modem. Now we will complete the initialization procedure that was started by the rstchn() function. We are following the procedure described on page 138 for resetting Hayes category hardware.

```
18    btulok(chan,0);
```

This turns input lockout off, undoing line 90 in function rstchn().

```
19   btuoes(chan,0);
```

This turns off output empty status. The output empty status option was enabled by line 91. We needed this to find out when the Hayes initialization command completed. Turning this option on is what caused the status 5 that got us into this code in the first place. We will no longer take special heed of the transition-to-empty event of the output data buffer.

```
20   }
21   else if (status == 3) {
```

A string was received. What could it be?

```
22   btuinp(chan,ibf);
```

Let’s get ahold of it and see...

```
23   if (strcmp(ibf,"CONNECT") == 0) {
```

Was it a message from the modem on this channel, indicating that a user has called up at 300 baud and that we have made connection with him?

```
24   btubrt(chan,300);
```

If yes, we must set the baud rate of the UART to 300 baud, because all subsequent communications with the modem will take place at 300 baud. We’ve just made the transition from command mode to online mode.

```
25   }
26   else if (strcmp(ibf,"CONNECT 1200") == 0) {
```

Or is the message that we’ve connected with a user’s modem at 1200 baud?

```
27   btubrt(chan,1200);
```

If yes, we must set our UART’s baud rate to 1200 baud.

```
28   }
29   else if (strcmp(ibf,"CONNECT 2400") == 0) {
```

Or is the message that we’ve connected with a user’s modem at 2400 baud?

```
30   btubrt(chan,2400);
```

Actually, this statement is unnecessary. We default the UART’s baud rate to 2400 baud after you reset the channel. Even this CONNECT 2400 message, like all of the CONNECT messages, has come to us at 2400 baud.

```
}
else {
     break;
}
```

If some other string is received from the channel, just ignore it.

```
btucmd(chan,"p");
```

If one of the above connect messages (for 300, 1200 or 2400 baud) has been received, we now must wait for the connection to settle. Our modem may have just changed baud rates, and so might the modem of the user that has just called up. To avoid losing any characters, we pause for 2 seconds.

```
state[chan]=1;
```

Channel state 1 means we are waiting for this 2 second period to pass.

```
}
break;
```

End of the channel state 0 case.

```
case 1:   /*--- CHANNEL STATE 1:  Wait for pause ---*/
```

We have been waiting for the 2 second pause to elapse...

```
if (status == 12) {
```

Do we have a “command-done” status condition?

```
btuxmt(chan,"\rTo log on, please enter your name: ");
```

If yes, say hello to this new caller.

```
btucli(chan);
```

Ignore anything he’s typed up to his point by clearing our data input buffer.

```
btuech(chan,1);
```

Now we will start echoing his keystrokes to his terminal.

```
state[chan]=2;
```

And he is online.

```
}
else {
```

If we have received any other status code from this channel while waiting for the 2 seconds to elapse, something bizarre is going on so...

```
rstchn(chan);
```

...we'll reset the channel.

```
}
break;
```

End of the channel state 1 case.

```
case 2:   /*--- CHANNEL STATE 2:  Wait for name ---*/
```

We have been waiting for the user to type in his name...

```
if (status == 3) {
```

Have we received something from him?

```
btuinp(chan,ibf);
```

Yes, let's get ahold of it.

```
sprintf(name[chan],"%1.20s",ibf);
```

And store it in the name[][] array. This will be his "handle" for the duration of his session. In case it is longer than 20 characters, we only store the first 20. This kind of precaution is necessary so that whatever a user does, he cannot crash the system. A good policy.

```
btuxmt(chan,"Okay, you're online!\r>");
```

Welcome him to the fray.

```
state[chan]=3;
```

Now he is online.

```
}
else if (status != 251) {
```

However, any other status (except input buffer overflow)...

```
rstchn(chan);
```

...means that we will hang up on him.

```
}
break;
```

End of the channel state 2 case.

```
case 3:   /*--- CHANNEL STATE 3:  Online ---*/
```

Now that he is online...

```
if (status == 3) {
```

...does he have anything to say?

```
btuinp(chan,ibf);
```

Let’s hear it...

```
sprintf(obf,"***\rFrom %s: %s\r>",name[chan],ibf);
```

...and format a message for everybody else to hear. The message will include his “handle” that he typed when he logged on.

```
for (i=0 ; i < 8 ; i++) {
```

For every channel...

```
if (i != chan && state[i] == 3) {
```

...not including chan, the one sending the message, and that is also online...

```
btuxmt(i,obf);
```

...transmit the message.

```
      }
}
btuxmt(chan,"*** Message sent ***\r>");
```

Tell the sender that the message was sent to everyone who is listening.

```
}
else if (status != 251) {
```

Otherwise, any other condition (except that he typed in too long of a line)...

```
rstchn(chan);
```

...means that we hang up on him.

```
}
break;
```

End of the channel state 3 case.

```
}
```

End of all possible values of the channel state.

```
}
```

End of check on btuscn() for channels needing service.

```
}
```

End of the main loop — system operator has pressed a key.

```
printf("TELECONFERENCE DEMONSTRATION OVER ...\n");
```

Tell him goodbye,

```
for (chan=0 ; chan < 8 ; chan++) {
     bturst(chan);
}
```

hang up all channels,

```
btuend();
```

shut down the Software Breakthrough,

```
}
```

and return to the operating system.

```

rstchn(chan)     /* Reset Channel */
int chan;
```

This function runs the initialization sequence recommended on page 138.

```
{
     bturst(chan);
```

First the channel is reset: data structures are cleared, and the UART is initialized.

```
btulok(chan,1);
```

Lock out input from this channel. This inhibits trash receive characters from inhibiting the upcoming transmission by btuxmt().

```
btuoes(chan,1);
```

Enable the generation of a status 5 when the transmit buffer goes from non-empty to empty. This will indicate that the Hayes-protocol command string has been fully transmitted to the modem.

```
btuech(chan,0);
```

Turn off the Software Breakthrough’s echo of every character it receives.

```
93   btuxmt(chan,"ATE0S0=1S2=1&C1&D2\r");
```

Send an init string to the modem:

AT	Begins almost all Hayes modem commands
E0	Modem echo off
S0=1	Auto answer mode on
S2=1	Escape character = CTRL+A
&C1	modem DCD indicates carrier detect
&D2	modem DTR input resets the modem

For more details on these commands, see your modem manual.

```
94   btucli(chan);
```

Clear any trash input from the channel.

```
95   state[chan]=0; /*Now we wait for a status 5 on this channel*/
```

We have done as much resetting of this channel as we can right now. The only work that remains is to turn off the input lockout and to turn off status 5 generation. For that we must wait for the status 5 that will come when the above Hayes-protocol command string has been fully transmitted to the modem. This is detected and serviced in lines 17-20 of this program.

```
96   }
```

End of the rstchn() function.

Teleconference for Xecom Hardware (TELCONX.C)

```
int chan,state[16],othchn;
char name[16][21],ibf[128],obf[163],*malloc();

main()
{
     printf("TELECONFERENCE DEMONSTRATION -- HIT ANY KEY TO STOP\n");
     btuitz(malloc(btusiz(16,128,1024)));
     btudef(0,0x2F0,16);
     while (!kbhit()) {
          if ((chan=btuscn()) >= 0) {
               switch (btusts(chan)) {
               case 1:   /*--- STATUS 1:  Phone ring or lost carrier ---*/
                    if (state[chan] == 0) {
                         btucmd(chan,"Ap");
                         state[chan]=1;
                    }
                    else {
                         bturst(chan);
                         state[chan]=0;
                    }
                    break;
               case 2:   /*--- STATUS 2:  Command complete ---*/
                    btuxmt(chan,"To log on, please enter your name: ");
                    break;
               case 3:   /*--- STATUS 3:  Input string available ---*/
                    if (btuinp(chan,ibf) > 0 && state[chan] == 1) {
                         sprintf(name[chan],"%1.20s",ibf);
                         btuxmt(chan,"Okay, you're online!\r>");
                         state[chan]=2;
                    }
                    else if (state[chan] == 2) {
                         sprintf(obf,"***\rFrom %s: %s\r>",name[chan],ibf);
                         for (othchn=0 ; othchn < 16 ; othchn++) {
                              if (othchn != chan && state[chan] == 2) {
                                   btuxmt(othchn,obf);
                              }
                         }
                         btuxmt(chan,"*** Message sent ***\r>");
                    }
                    break;
               case 251:   /*--- STATUS 251:  Input buffer overflow ---*/
                    break;
               default:    /*--- STATUS unknown ---*/
                    bturst(chan);
                    state[chan]=0;
                    break;
               }
          }
     }
     printf("TELECONFERENCE DEMONSTRATION OVER, RETURNING TO DOS\n");
     for (chan=0 ; chan < 16 ; chan++) {
          bturst(chan);
     }
     btuend();
}
```

Line-by-Line Synopsis

```
101  int chan,state[16],othchn;
```

Declare integer variables:

chan	channel or user number
state[]	user-specific state code: 0 waiting for ring 1 waiting for carrier or waiting for name 2 online
othchn	the other channel

```
102    char name[16][21],ibf[128],obf[163],*malloc();
```

Declare several character variables:

name[][]	array of names for each user
ibf[]	input buffer (one user at a time)
obf[]	output buffer (one user at a time)
*malloc()	the standard memory allocation utility (it returns a pointer to character)

```
103
104  main()
```

Define the “main” function, the main part of this program.

```
105  {
106       printf("TELECONFERENCE DEMONSTRATION . . .\n");
```

Greeting for the system operator.

```
107  btuitz(malloc(btusiz(16,128,1024)));
```

btusiz()	specifies the buffer sizes (input 128, output 1024), and the total number of channels (16). Computes the total number of bytes required for the Software Breakthrough
malloc()	The standard dynamic memory allocation utility
btuitz()	Initializes the Software Breakthrough and “formats” its data structures in the memory block allocated by malloc()

```
btudef(0,0x2F0,16);
```

Define 16 channels (numbered 0 to 15) with an I/O base address of 2F0 hex. This would also work for the 4 channels of a Model 4.

```
while (!kbhit()) {
```

The main loop of the program continues until the system operator presses a key.

```
if ((chan=btuscn()) >= 0) {
```

If any channel requires service, its channel number will be put in the variable chan.

```
switch (btusts(chan)) {
```

Now let's treat each possible status condition differently.

```
case 1:   /*--- STATUS 1: Phone ring/lost carrier ---*/
```

A status code 1 can mean two very different things. If we are on-hook, it means the phone is ringing. In this case we will want to answer it. If we are off-hook, it means that we have lost the carrier signal. In that case we will want to hang up that channel (go on-hook).

```
if (state[chan] == 0) {
```

If this channel state is zero, then the status 1 means that we detect the ringing of an incoming call.

```
btucmd(chan,"Ap");
```

We will answer the incoming call (A answer command), and give the channel time to settle (p pause command). Answering a channel consists of going off-hook and beginning to sound the answer carrier signal.

```
state[chan]=1;
```

We now expect originate carrier to be detected.

```
}
else {
```

If the channel state is nonzero, then the status 1 means that we are no longer getting the quiescent carrier signal from this channel — the user has probably hung up on us...

```
118  bturst(chan);
```

...so we hang up on him...

```
119  state[chan]=0;
```

...and wait for more calls on this channel.

```
120       }
121       break;
122  case 2:   /*--- STATUS 2:  Command complete ---*/
```

The status code of 2 means that the command string specified using btucmd() (in line 14) has completed successfully. This means we have detected originate carrier from this channel.

```
123  btuxmt(chan,"To log on, please enter your name: ");
```

Ask this caller to give us his name.

```
124       break;
125  case 3:   /*--- STATUS 3:  Input string available ---*/
```

This status code means that a carriage-return terminated string has been received from this channel. Either this is a new caller’s name (in response to the question in line 23), or a message that he wants sent to other users.

```
126  if (btuinp(chan,ibf) > 0 && state[chan] == 1) {
```

We have gotten a string of at least one character, and we are expecting a new caller’s name.

```
127  sprintf(name[chan],"%1.20s",ibf);
```

Record his name, but limit its length to 20 characters maximum.

```
128  btuxmt(chan,"Okay, you're online!\r>");
```

Let him in the door. Note the **>** prompt for his first message.

```
129  state[chan]=2;
```

We will treat all further information from him as messages to other users.

```
130  }
131  else if (state[chan] == 2) {
```

Now, here is the case where an input string came in and we were expecting a message from him.

```
132  sprintf(obf,"***\rFrom %s: %s\r>",name[chan],ibf);
```

Format the complete message to be sent to the other users who are online.

```
133  for (othchn=0 ; othchn < 16 ; othchn++) {
```

Check every channel...

```
134  if (othchn != chan && state[chan] == 2) {
```

...except for channel chan, the one that is sending the message. If there is a user on another channel, and he is online...

```
135  btuxmt(othchn,obf);
```

Transmit this user's message to the other user.

```
136       }
137  }
138  btuxmt(chan,"*** Message sent ***\r>");
```

Let the sender of the message know that he is being heard.

```
139       }
140       break;
141  case 251:   /*--- STATUS 251: Input buffer overflow ---
*/
```

This status indicates that the input from the user on this channel has exceeded the size of the input buffer (128 bytes, as specified in btusiz(), line 107). We will just ignore this condition. This will result in missing characters from the user's input string.

```
142       break;
143  default:    /*--- STATUS unknown ---*/
```

Several status codes indicate that the answer command (line 14) was not successful. These conditions are treated in the same way:

```
144  bturst(chan);
```

Hang up the channel...

```
145  state[chan]=0;
```

...and wait for the next call on it.

```
     break;
}
```

This is the end of the possible status codes.

```
}
```

This is the end of the case where btuscn() indicates that some channel needed servicing.

```
}
```

The system operator has pressed a key...

```
printf("TELECONFERENCE DEMONSTRATION OVER . . .");
```

Tell him goodbye . . .

```
for (chan=0 ; chan < 16 ; chan++) {
     bturst(chan);
}
```

And reset every channel, hanging up any calls that may have been in progress.

```
     btuend();
}
```

De-install the Software Breakthrough (a very important step).

LAN Programming

Socket Numbers from Novell

The sixteen socket numbers 80BB through 80C2 and 86D0 through 86D7 (hexadecimal) are registered with Novell by Galacticomm for use in Worldgroup software. Novell assigns "well-known" socket numbers for released software out of the range 8000 to FFFF. Alternatively, socket numbers 4000 through 7FFF are available for dynamic or experimental usage. Socket numbers 0000 through 3FFF are reserved by Novell, and may be used in accordance with Novell's instructions (e.g. such as for socket 0452, Service Advertising Protocol).

A value of zero for the socket parameter of btusdf() means pick an available dynamic socket number (4000 to 7FFF). In this case, the actual socket number assigned is available in the global integer lansop (page 29).

Byte Order

You will encounter a socket number with reversed byte order when you are dealing with the IPX headers yourself on IPX Virtual channels in raw-packet mode (page 220).

Warning: The length, dstsoc (destination socket), and srcsoc (source socket) fields of an IPX header contain values stored with the most-significant-byte in the lower address. This is foreign to Intel processors, so you need to remember to swap the bytes of these fields.

The following macro might be helpful for this (and is included in IPX.H):

```
#define hilo(i)  (((unsigned)(i)>>8)¦((unsigned char)(i)<<8))
```

This simply swaps the two bytes of a 16-bit int or unsigned variable.

Multiple Channels on the Same Local Socket

You can define several channels or several channel groups on the same local socket number if you follow a few rules.

Do not define IPX channels (Direct or Virtual) on the same local socket number as SPX channels.

If you define more than one channel group on the same local socket number, the groups must use consecutive channels.

You can define multiple channel groups on the same local socket number as long as the groups are defined over consecutive channel numbers, and groups defined on a given socket are either all IPX, or all SPX — don’t mix and match IPX and SPX on a single socket. You *can* however, mix IPX Direct Circuits with IPX Virtual Circuits on the same local socket.

Do not connect IPX channels to SPX channels. You *can*, however, connect an IPX Direct Circuit with an IPX Virtual Circuit.

Do not connect multiple IPX channels between two nodes (computers) using the same local socket number on both ends.

You can have several SPX connections on one network/node/socket talking one-to-one with several SPX connections on another network/node/socket. You can*not* do this with IPX, however. The only way IPX has of distinguishing what channel an incoming IPX packet is for is the local socket (of course) and the network/node/socket that the packet originated from. SPX has connection ID’s to distinguish who’s talking to who, and there can be several on one socket. To have multiple IPX connections between two nodes using GSBL, either the local end, the remote end, or both, must use separate sockets.

LAN Channel State-Machines

IPX Direct Circuit Call

Terminating:

bturst(chan);	(direct circuit address is subsequently undefined)

Incoming or outgoing:

btucmd(chan,"W000000010000C0A810184007M");

IPX direct channels are always in the “communicating” state.

IPX Virtual Circuit Call

Terminating:

bturst(chan);	(channel reverts to raw-packet mode)

Incoming:

btutrg(chan,30);	(prep for incoming 30-byte IPX header)
btusts(chan) == 4	
btuict(...);	returns a 30-byte IPX header (page 221), and inside it, the source network/node/socket contains the 12-byte node address in binary

btucmd(chan,"W<24-digit node address in ASCII hex>M");

Outgoing:

btucmd(chan,"W000000010000C0A810184007M");

States:

raw-packet mode

communicating

SPX Call

Terminating when in session:

```
    btucmd(chan,"T");
    wait for btusts(chan) == 36        (You should timeout this wait, in case a status
                                       36 never comes for some reason)
    bturst(chan);          (optional)
```

Terminating otherwise:

```
    bturst(chan);
```

Incoming:

```
    btucmd(chan,"L");
    wait for btusts(chan) == 34
```

Outgoing:

```
    btucmd(chan,"W000000010000C0A810184007M");
    wait for btusts(chan) == 35
```

States:

    idle
    waiting for incoming call
    waiting for outgoing call
    connection established
    terminating

The SPX Channel State-Machine

Each SPX channel is always in one of these five states:

    Idle
    Listening
    Outdialing
    Connected
    Terminating

Here’s a chronological diagram of how two applications based upon the GSBL might communicate over SPX channels:

```
Calling party                                Listening Party

                                             btucmd(chan,"L");
btucmd(chan,"W000000010000C0A810184007M");
status 35                                    status 34
 :                                            :
 :                                            :
 :                                            :
btucmd(chan,"T");
status 36                                    status 31
```

Note: once the link is established, it is symmetrical; either channel can terminate it. In the above diagram this means, for example, that the listening party might decide to terminate the connection with btucmd(chan,"T"). It would soon get a status 36, and the calling party would get a status 31.

If either party abruptly aborts the connection with bturst(), the other party will eventually get a status 39. The aborting party will get no special status code.

Idle SPX Channel

All SPX channels become idle immediately after any of the following:

- Definition by btusdf() (only allowed once per program load)
- Reset by bturst() (may be done at any time)
- Status 31: connection explicitly terminated by the remote party
- Status 39: watchdog termination, meaning the other party has just disappeared, or some other transmit or connection error (see page 158)

When a channel is idle, and only when a channel is idle, you can issue dial-out or listen commands.

Listening SPX Channel

After issuing the listen command (e.g. btucmd(chan,"L")), the channel is in the listening state, and ready to receive an incoming call from another network party (using SPX). Status 34 indicates when such a call has come in and been completed. In that case, the channel moves to the connected state.

The listening state has no timeout, and a channel may remain in the listening state indefinitely. To give up waiting for an incoming call, and return a channel to the idle state, call bturst().

Outdialing SPX Channel

After issuing the dial-out command (e.g. btucmd(chan,"W...M")), the channel is in the outdialing state, and trying to make connection with the network address specified in the dial-out command. Status 35 indicates when the call has gone through and the channel is in the connected state.

The dial-out command may timeout, in which case a status 39 indicates that the call is aborted and the channel is reverting to the idle state. To give up waiting for an outgoing call to complete, and return a channel to the idle state, call bturst().

Connected SPX Channel

After an SPX channel has established a connection with another party on the network, you can transmit to the party (btuxmt(), btuxct()) and receive from it (btuinp(), btuict(), btuica()). Now most of the hardware specifics vanish, and the channel acts much like any other channel on the GSBL.

SPX connections that have been completed (via status 34 or 35) can end in one of four ways:

- Explicitly, by issuing the **T** command and then waiting for status 36
- Remotely, by receiving a status 31 (other side terminated — see page 157)
- Abortively, by resetting the channel (bturst())
- Unexpectedly, by receiving a status 39 (watchdog abort)

Terminating SPX Channel

When you issue the terminate command (e.g. btucmd(chan,"T")), the channel moves to the terminating state. This is the polite way to end a connection, because the SPX system tries to make sure the other party knows the connection is ending. To end a connection quickly, with no muss and no fuss, just call bturst(), and the channel moves immediately to the idle state.

After issuing the terminate command on a channel that is in the connected state, the channel moves to the terminating state. Pretty soon, a status 36 indicates that the other party has acknowledged the termination and the connection has ended gracefully. The channel returns to the idle state.

On the other hand, a status 39 indicates that the termination notice was not acknowledged, and we can’t be sure what the other party thinks. After a status 39, the channel ends up in the idle state.

IPX Virtual Circuits in Raw Packet Mode

```
               ___________
/      /     0|  chksum   |  (set by IPX)
|      |      |___________|
|      |     2|  length   |  Total number of bytes, incl header
|      |      |___________|  Range 30-576, WARNING: MSB FIRST!
|      |     4|  trnspt   |  (set by IPX)
|      |     5|  paktyp   |  Packet type, 4=IPX packets
|      |     6|           |
|      |      |  dstnet   |  Destination network, 4 binary
|      |      |  bytes,   |  right-justified, and zero-
|      |      |___________|  filled
|      |    10|           |
|      |      |           |
|      |      |  dstnod   |  Destination node, 6 binary bytes,
|      |      |           |  right-justified, and zero-filled
|     30      |           |
|   bytes     |___________|
|      |    16|  dstsoc   |  Destination socket
|      |      |___________|  WARNING: MOST SIGNIFICANT BYTE FIRST!
|      |    18|           |
30     |      |  srcnet   |  Source network, 4 binary bytes,
to     |      |           |  right-justified, and zero-filled
576    |      |___________|
bytes  |    22|           |
|      |      |           |
|      |      |  srcnod   |  Source node, 6 binary bytes,
|      |      |           |  right-justified, and zero-filled
|      |      |           |
|      |      |___________|
|      |    28|  srcsoc   |  Source socket
|      \      |___________|  WARNING: MOST SIGNIFICANT BYTE FIRST!
|           30|           |
|             |           |
|             |           |
|             |  datbuf   |  Data buffer, containing 0 to 546
|             |           |  bytes
|             |           |
|             |           |
|             |           |
|             |           |
|             |           |
|             |           |
|             |           |
|             |           |
|             |           |
|             |           |
|             |           |
|             |           |
|             |           |
|             |           |
\             |___________|
```

Figure 6-1: IPX Virtual Raw Packet Format

Before transmitting a raw IPX packet, (that is, before transmitting a packet on an IPX Virtual channel in raw-packet mode), you must set the following fields of the packet:

length	Length of entire packet, including header, range 30 to 576 WARNING: most significant byte first
paktyp	Packet type (4 for IPX)
dstnet	Destination network
dstnod	Destination node
dstsoc	Destination socket WARNING: most significant byte first
datbuf	Data buffer, 0 to 546 bytes

The following fields are automatically set by IPX:

chksum	Checksum
trnspt	Transport control
srcnet	Source network
srcnod	Source node
srcsoc	Source socket

Incomplete Packets Under Raw-Packet Mode

WARNING: You can only transmit complete packets in raw-packet mode. Fill the length field (in hi-lo byte order) with the total number of bytes in the packet, including the IPX header. The GSBL will use this field to determine packet boundaries. If you return to btuscn() in your main loop with a partial packet in the output buffer, the packet may be lost.

Also, if you don’t make sure that the length byte contains the actual packet length, or if you otherwise mess up packet alignment, then total trash could be transmitted via IPX, possibly clobbering your file server.

You could use this property (that partial raw packets will be lost) to detect when a packet transmitted in raw-packet mode has actually been totally and completely transmitted.

First, let’s look at the problem: it is not enough just to poll at btuoba() (or to enable and then wait for status 5) to know when transmitting is done — the GSBL may be done with a packet, but the low level Netware IPX interface may not have completed the associated “Send Event Control Block”. So you could diligently make sure the output buffer was empty, and then innocently call bturst() thereby cancelling the data before it went out! You need to make sure IPX has finished transmitting the packet.

If after transmitting a raw packet, you “transmit” a single NUL (as in btuxct(chan,1,"\0")) and then wait for btuoba() to indicate an empty buffer (when it returns output buffer size minus 1), the raw packet preceding the NUL will have been completely transmitted, and the NUL will be discarded.

```
struct {
     struct ipx ipx;
     char data[NBYTES];
} ipxpkt;

/* Fill up ipxpkt somehow. */
btuxct(chan,sizeof(ipxpkt),&ipxpkt);
btuxct(chan,1,"\0");
while (btuoba(chan) < OSIZE-1) {
     btuscn();
}
bturst(chan);
```

This code is an excerpt only (btusiz(), btuitz(), btusdf(), btuend() calls are all essential but they are not shown above). And the while-loop should really have a timeout. And you would rarely call btuscn() only inside a loop like that. But this should give you an idea of what’s involved in this little trick to detect total completion of raw packet transmission.

IPX Raw-Packet Transmit Example

For example, let’s say you wanted to transmit the alphabet on an IPX Virtual channel in raw-packet mode to socket 4007 hex on node 2 of network 1. Assume you are using local socket 5008 hex. Here is code that would do this:

```
#define hilo(i) (((unsigned)(i)>>8)|((unsigned char)(i)<<8))

struct ipxhdr {                             /* an IPX header contains...   */
     int chksum;                                             /* checksum   */
     int length;           /* 30-576:  header PLUS rest of packet (hi-lo)  */
     char trnspt;                                       /* IPX sets to 0   */
     char paktyp;                                /* 0=unknown 4=IPX 5=SPX  */
     char *stnet[4];                                 /* destination network*/
     char dstnod[6];                                 /* destination node   */
     int dstsoc;                              /* destination socket (hi-lo)*/
     char srcnet[4];                                  /* source network    */
     char srcnod[6];                                        /* source node */
     int srcsoc;                                   /* source socket (hi-lo)*/
};

#define IPXMAX 546               /* max length of data field of IPX packet */

struct ipxpak {                                   /* an IPX packet contains*/
     struct ipxhdr hdr;                                      /* IPX header */
     char datbuf[IPXMAX];                          /* 0 to 546 data bytes */
}

struct ipxpak xmtpak;
char net[]={0,0,0,1};
char nod[]={0,0,0,0,0,2};
char alphabet[]="ABCDEFGHIJKLMNOPQRSTUVWXYZ";
int chan=0;
char *malloc();

main()
{
     btuitz(malloc(btusiz(1,256,2048)));
     btusdf(chan,1,6,0x5008,2);
     xmtpak.hdr.length=hilo(26);        /* xmtpak.hdr.length=0x1A00 */
     xmtpak.hdr.paktyp=4;
     movmem(net,xmtpak.hdr.dstnet,6);
     movmem(nod,xmtpak.hdr.dstnod,4);
     xmtpak.hdr.dstsoc=hilo(0x4007);    /* xmtpak.hdr.dstsoc=0x0740 */
     movmem(alphabet,xmtpak.datbuf,26);
     btuxct(chan,&xmtpak,sizeof(struct ipxhdr)+26);
     while(btuoba(chan) < 2047) {
          btuscn();
     }
     btuend();
}
```

Note: this program is shown as a coding example and is almost useless otherwise, especially in its limited versatility and in its failure to check for erroneous return codes.

The calls to btusiz(), malloc(), and btuitz() prepare for one channel with a 2048-byte output buffer and a 256-byte input buffer.

The call to btusdf() defines one IPX Virtual channel using socket 5008 hex, with 2 ECBs. Note that the specification of the socket number here uses natural (Intel) byte order.

Galacticomm Terminal Configuration (GTC) Protocol

GTC enables one GSBL-based program to tell another that it is willing to preprocess the other's ASCII input.

For example, a terminal program based on GSBL function calls can tell an online service program based on GSBL that the terminal will buffer line input before sending it to the online service.

In this section we'll talk about the "terminal" and the "server" as the GSBL program volunteering to preprocess input and the GSBL program that relinquishes input preprocessing, respectively. Other applications could make use of this feature, but the terminal/server situation is why GTC was developed, and makes for a clear example.

In this case, each of the programs (terminal and server) consists of a copy of the GSBL and an application program that uses it.

The usurpation of the server's input handling is detected and allowed by the server GSBL and is transparent to the server application program. GTC service starts with the terminal program sending a "greeting message" to the server GSBL. After that, the server GSBL will notify the terminal GSBL as to the current input mode. The terminal GSBL passes all such GTC information, in the form of status codes, to the terminal application program for processing.

For enhanced compatibility, a GTC-compatible server will not require a GTC-supporting terminal program. It is up to the terminal to request GTC before the server starts depending on it. At the same time, if a non-GTC server ignores the terminal's initial greeting message, the terminal GSBL will not generate GTC status codes.

If you don't want the main program to have to deal with GTC, then just don't ever send the "greeting message" (don't call btucmd(chan,"G")).

A new greeting must be issued after a channel reset from either the server or the terminal side.

Several varieties of input are supported, so that features like editor input word-wrap, teleconference chat, and password input operate transparently — that is, just as they would calling over a non-LAN GSBL channel. The special case of suppressed output during line input (as is used so well in the teleconference) can be handled at the option of the terminal program: (1) suppress the output while entering a line; or (2) go ahead and display the output, but "move" the input line down to the new prompt.

The GTC protocol for IPX packets is based on:

IPX header with packet type 0 (unknown packet)

For SPX packets:

SPX header with data stream type F0 hex

Data fields:

Greeting message:

(empty data buffer)

The terminal sends this to the server (the **G** command) to volunteer to preprocess the server's ASCII input. The server GSBL takes note that it is communicating with a GTC-compatible program.

Input mode message:

"GTCI"	('I' means input configuration)	
1 byte	inmode	input mode (see below)
1 byte	maxinl	maximum input line length (0=unlimited)

These messages are transmitted from the server to the terminal to tell the terminal how to treat the server's input (that the terminal is gathering, for example, from an operator at the keyboard, or from a disk file).

Input modes (decimal)

40	Locked out
41	Binary
42	ASCII line mode without echo
43	ASCII line mode with echo (limit line length)
44	ASCII line mode with echo and word-wrap (limit line length)

These input modes are also the status codes that will eventually appear to the terminal application program when it calls btusts(). When the status code is 43 or 44, another status byte follows immediately, which is the input line length, where 0 is unlimited. See starting on page 155 for more details on these status codes.

CTRL+S Pause with GTC

Also we recommend that a terminal program locally process the CTRL+S pause of server output. This function cannot be handled by the server due to packetization: by the time the server would get a CTRL+S, it may have already shipped out several huge packets.

X.25 Programming

The features described in this section are available only with the X.25 Software Option for the GSBL. See page 10 for an overview of how X.25 works.

The GSBL handles low-level communications with X.25 and LAN channels differently than with modem or serial channels. While modems or serial ports must be timer-interrupt driven at the byte level, X.25 and LAN channels are handled without interrupts, at the packet level.

The btuxmt() output routine still puts data into a circular buffer. And the btuinp() routine still gets data out of another circular buffer — these aspects of the GSBL have not changed. However, the interaction between these buffers and the X.25 or LAN hardware takes place transparently during your call to btuscn().

So remember that only after your call to btuscn():

- Will received data become available for:
  btuibw(), count of input bytes waiting
  btuinp(), ASCII input
  btuict(), binary input
  btuica(), binary input
- Does transmitted data, specified by:
  btuxmt(), ASCII output
  btuxct(), binary output

  actually get formed into packets and transmitted, and will:

  btuoba(), output bytes available

  indicate room in the output buffer
- Will monitored output become available:
  btumon(), start monitoring a channel
  btumds(), report latest output to that channel
  (similarly with btumon2() and btumds2())

Simulated keystrokes on the monitored channel (btumks() or btumks2()), however, immediately become available in the input buffer.

Handling an Incoming Call

When an incoming "call request" packet is received by the server on an X.25 virtual circuit:

btusts() returns status 3

btuinp() returns a string of the form:

RING <caller> CALLING <callee>

Where <caller> is the decimal network address (if available) for the source of this call, and <callee> is your network address. If you set x25udt to 1, then btuinp() will return:

RING <caller> CALLING <callee> <user data field...>

See page 32 for details on the limitations of this method.

To answer an incoming call:

```
btucmd("A");            (which will immediately generate a status 22, see also page
                                    157)
```

Making an Outgoing Call

To place an outgoing call on an X.25 channel, issue any of the following command string formats using btucmd(chan,cmdstg) (page 75):

```
W<caller>,<callee>/<user data field>M
W<callee>/<user data field>M
W<caller>,<callee>M
W<callee>M
```

The <caller> and <callee> fields are decimal addresses, up to 16 digits each. The <user data field> is specified in hexadecimal. See page 75 for more details about making an outgoing call on an X.25 virtual circuit.

Index

GSBL Quick Reference

	Routine	Description	Page
Initialization	`btusiz(nchan,isiz,osiz)`	Size (bytes) of dynamic memory needed	152
	`btulsz(nchan,isiz,osiz)`	Long version of `btusiz()`	117
	`btuitz(region)`	Initialize the Software Breakthrough	112
	`btuitm(region)`	Initialize for multitasking environment	111
	`btumxs(bdrate)`	Set maximum data speed	129
	`btudef(schan,sport,n)`	Define channels	80
	`btusdf(schan,n,chtype)`	Super-define channels	147
	`btuudf(schan,n)`	Un-define channels	176
	`bturti(n,rtirou)`	Define real-time interrupt routine	142
	`btuirp(comno)`	Use COM1/2/3/4 port as timing source	110
	`btuhit(irqno)`	Hook I/O interrupts from UART	97
Status	`btuscn()`	Scan channels for need of service	144
	`btusts(chan)`	Status of a channel	154
	`btuibw(chan)`	Input bytes waiting	102
	`btuoba(chan)`	Output buffer space (bytes) available	132
	`btueba(chan)`	Echo buffer space (bytes) available	83
	`btuinj(chan,status)`	Inject status-code into a channel	107
Command	`btucmd(chan,cmdstg)`	Command channel	54
Receiving	`btuinp(chan,inbuff)`	Input from channel (ASCIIZ string)	108
	`btuict(chan,inbuff)`	Input from channel (by trigger count)	105
	`btuica(chan,inbuff,siz)`	Input from channel (up to `siz` bytes)	103
	`btutrg(chan,nbyt)`	Set input byte count trigger	167
Transmitting	`btuxmt(chan,datstg)`	Transmit to channel (ASCIIZ string)	189
	`btuxct(chan,nbyt,datstg)`	Transmit to channel (by byte count)	182
	`btuxmn(chan,datstg)`	Non-clearable ASCII transmit	187
	`btux29(chan,nbyt,datstg)`	Transmit an X.29 string to remote PAD	179
Receiver Modes	`btuxlt(oldchr,newchr)`	Set input translation table	184
	`btumil(chan,maxinl)`	Set max input length / word-wrap on/off	122
	`btutrm(chan,crchar)`	Set input line terminator character	169
	`btulok(chan,onoff)`	Set input lockout on/off	116
	`btuerp(chan,onoff)`	Pass/block input bytes with errors	88
	`btuusp(chan,onoff)`	Special UART polling method	177
	`btuchi(chan,rouadr)`	Set character input interceptor	44
	`btuche(chan,onoff)`	Call `btuchi()` when echo buffer empties	42
Transmitter Modes	`btulfd(chan,lfchar)`	Set linefeed character (follows CR)	114
	`btutsw(chan,width)`	Set terminal screen width	172
	`btuscr(chan,softcr)`	Set soft-CR (for output word-wrap)	146
	`btuhcr(chan,hardcr)`	Set hard-CR (for output word-wrap)	92
	`btuoes(chan,onoff)`	Enable/disable output-empty status codes	133
Receiver/ Transmitter Modes	`btubrt(chan,bdrate)`	Set channel's baud rate	34
	`btuech(chan,onoff)`	Set echo on/off	84
	`btubse(chan,bschar)`	Set backspace echo character	37
	`btupmt(chan,pmchar)`	Set prompt character	136
	`btuffo(chan,onoff)`	Enable receive FIFO for 16550 UART	90
Flow Control and Output Throttling	`btuhwh(chan,inpcut)`	Enable RTS/CTS hardware handshaking	100
	`btuxnf(chan,xon,xoff)`	Set XON/XOFF characters	193
	`btutru(chan,trunch)`	Set output-abort character	171
	`btutrs(chan,onoff)`	Generate status 6 when so aborted	170
	`btupbc(chan,pausch)`	Character to initiate screen pause	135
	`btuhpk(chan,hpkrou)`	Handle screen-pause keystroke	98
	`btucpc(chan,cpchar)`	Character to clear pause-counter	79
	`btuolk(chan,onoff)`	Lock out or resume output	134
Monitoring and Diagnostics	`btumon(chan)`	Start/stop monitoring a channel	126,128
	`btumds()`	Get next displayed character	120,121
	`btumks(kyschr)`	Simulate a keystroke	124,125
	`btuhdr(chan,nbytes,buff)`	Capture information on channel	93
	`bturep(chan,statid)`	Report on channel statistics	137
	`btuset(chan,statid)`	Set and report channel statistics	150
Utilities	`bturst(chan)`	Reset a channel	138
	`btuclo(chan)`	Clear data output buffer	52
	`btucli(chan)`	Clear data input buffer	51
	`btuclc(chan)`	Clear command output buffer	50
	`btucls(chan)`	Clear status input buffer	53
	`btubsz(chan,isiz,osiz)`	Respecify input/output buffer sizes	39