Table of Contents

In this Manual

Serial Protocol

Introduction

This manual describes the software protocol used to interface an external device to the Accom DVEOUS digital effects system. Since DVEOUS can connect to editors, switchers and other devices, a variety of protocols and interface methods are offered. When interfacing a remote device to DVEOUS, use this manual to find the protocol and appropriate serial port for your device. In addition, refer to the DVEOUS Technical Guide, which details the various connectors and associated pinouts.

Interfacing methods fall into these categories:

• The DVEOUS appears as a VTR under SMPTE or Sony protocol to the remote device. Use this method for connection to an edit controller. Since DVEOUS can compose keyframe sequences into Effects, the Effects can be controlled as if they were a VTR. DVEOUS can interface to external editors as a VTR by using either Sony or SMPTE VTR protocols.

•The DVEOUS responds to external commands using serial protocols such as A53 serial protocol (limited command set), Peripheral Bus I and II (provides DVEOUS effect save and recall from a switcher). Use this method to control DVEOUS with a switcher.

• When DVEOUS is used with a protocol converter module or an external controller, you can use GVG Control Point Language (CPL) to control 4 Aux Bus source selections and run timeline from both the DVEOUS and the connected switcher (GVG Model 3000).

Ports

Interfacing DVEOUS with an editor or switcher is possible using the three serial ports on the rear DVEOUS chassis. However, each of the three ports can only be used with certain types of interface protocols as summarized below:

Serial Port 1 (Editor)

•Sony (VTR)

•SMPTE (Transport commands supported, such as Speed and Goto)

•EMEM (Grass Valley Group Peripheral Bus I and II protocols)

•Switcher Aux bus crosspoint selection (Grass Valley Group Aux Bus protocol)

•Control Point Language (CPL)

Serial Port 2 (AUX)

•Sony (VTR)

•SMPTE (Transport commands supported, such as Speed and Goto)

•EMEM (Grass Valley Group Peripheral Bus I and II protocols)

•Switcher Aux bus crosspoint selection (Grass Valley Group Aux Bus protocol)

•Control Point Language (CPL)

Serial Port 3 (LINC)

•LINC protocol (slave only). External devices can control DVEOUS timelines by using LINC protocol.

SMPTE Protocol

The VTR SMPTE protocol has been extended to included EFFECT commands in the same serial port as the TRANSPORT commands for VTR control. In this way the DVEOUS can be controlled as a VTR and effects machine simultaneously.

This protocol is intended to follow the Ampex VPR implementation. See that protocol for the specifics of the data bit format. To avoid confusion with the SMPTE protocol timeline, the DVEOUS's timeline is referenced by “effect.”

The commands are listed on the following pages:

Status **01H**

The STATUS command returns the current status of the device.

The STATUS command returns the following information to the host:

byte1	HR of current time	
byte2	MIN of current time	
byte3	SECS of current time	
byte4	FRAMES of current time	
byte5	bit0	1 = NTSC, 0 = PAL
	bit1	1 = drop frame, 0 = non drop frame
	bit2	not used
	bit3	not used
	bit4	not used
	bit5	not used
	bit6	1 = data not available, 0 data available
	bit7	not used
byte6	VTR MODE	
	00	stopped
	01	stopping
	02	play
	03	tso
	04	shuttle
	05	fast forward
	06	rewind
	07	syncing
	08	source sync
	09	not used
	0A	not used
	0B	not used
	0C	cueing
	0D	cued
	0E	searching
	0F	search complete
	10	not used
	11	not used
	12	not used
	13	not used
	14	not used
	15	not used
	16	not used

Xstatus Commands

The XSTATUS command returns the following information to the host:

byte1	machine type	= 03H
byte2	msb of ID	= 00H
byte3	lsb of ID	= 36H (A57)
byte4	*not used*	

Xstatus **02H**

The XSTATUS command returns the machine ID of the device and error information.

Defer **03H**

The DEFER command is used to defer TRANSPORT, and EFFECT commands until a point on the timeline is reached. This command is implemented in the format of the Ampex VPR3.

Tcue **04H**

The TCUE command is used to cue the device to the first event on the timeline.

Tstop **05H**

The TSTOP command is used to stop the running timeline. This command also clears out the timeline buffers.

Tclear **06H**

The TCLEAR command is used to clear out the timeline buffers.

Trun **07H**

The TRUN command is used to load the timeline with its initial value and then run the timeline.

Transport 26H

The Transport commands are implemented much like the Ampex VPR3, VPR300 command set. Some commands, however, need special attention.

01H	READY	Turn on/off scanner
02H	EE	Tape/EE switch
03H	CF	Set color framer mode
06H	EDMODE	Set edit mode
07H	ENABLES	Set channel enables
08H	TCMODE	Control timecode generator
09H	SPEED	Control effect speed
0AH	TCG	Load timecode generator
0BH	LDUBG	Load user bit generator
0CH	STOP	Stop the effect
0DH	PLAY	Play the effect 1X
0EH	ROLL	Play and sync to external reference
0FH	TSO	Tape speed override
10H	VAR	Variable play speed
11H	SHUTTLE	Shuttle speed
12H	PREROLL	Set the preroll amount
13H	SYNC	Mark sync point for cue command
14H	CONTROL	Mark sync point for non play speed
15H	CUE	Cue to "park" position
16H	SEARCH	Search to exact position
17H	ENTRY	Channel record entry
18H	EXIT	Channel record exit
1AH	RDREADY	Read ready status
1BH	RDEE	Read EE status
1CH	RDCF	Read CF status
1FH	RDEDMODE	Read edit mode status
20H	RDENABLE	Read enable status
25H	FJOG	Jog forward 1 frame
26H	RJOG	Jog reverse 1 frame
27H	ACCURACY	Sync accuracy window
28H	TMLOAD	Load current time source
29H	NBASE	Read machine standard
2AH	CFSEL	Color framer source selection
2BH	RDCFSRC	Read color framer source
2CH	TMSEL	Select current time source
2DH	RDTMSEL	Read current time source
32H	EDFIELD	Select edit field
33H	RDEDFIELD	Read edit field
35H	SLEW	Slew forward or reverse N frames
36H	STILL	AST playback when stopped
42H	FFJOG	Jog forward 1 field
43H	RFJOG	Jog reverse 1 field

READY	**01H**, ready state (1 byte) This command has no impact on the effect but is acknowledged.
EE	**02H**, ee state (1 byte) This command has no impact on the effect but is acknowledged.
CF	**03H**, cf state (1 byte) This command has no impact on the effect but is acknowledged.
EDMODE	**06H**, edit mode (1 byte) This command has no impact on the effect but is acknowledged.
ENABLE	**07H**, enables (1 byte) This command has no impact on the effect but is acknowledged.
TCMODE	**08H**, mode (1 byte) This command has no impact on the effect but is acknowledged.
SPEED	**09H**, speed bytes (2 bytes) Control transport speed in current mode. byte1 & 2 = speed magnitude as defined in each mode. (TSO,VAR, etc.)
TCG	**0AH**, time bytes (4 bytes) This command has no impact on the effect but is acknowledged.
STOP	**0CH** Stop the effect.
PLAY	**0DH** Play the effect at Play 1X.
ROLL	**0EH** Play the effect at Play 1X. (same effect as above)
TSO	**0FH**, speed bytes (2 bytes) Control the effect in tape speed override mode. *byte1 & 2* = speed magnitude (Linear scale in decimal) Magnitude is:

500	= PLAY 1X + 25%
0	= PLAY 1X
-500	= PLAY 1X - 25%

VARPLAY **10H**, speed bytes (2 bytes)

Control the effect in variable play speed mode.
byte1 & 2 = speed magnitude (Linear scale in decimal)

Magnitude is:

1535	= PLAY 3X
511	= PLAY 1X
0	= STOP
-511	= - PLAY 1X

Speed data is internally = *magnitude* * 1000 / 511 (1000 = Play 1X)
Sending a data value of 511 runs effect at PLAY 1X.

SHUTTLE **11H**, speed bytes (2 bytes)

Control the effect in shuttle mode.
byte1 & 2 = speed magnitude (Linear scale in decimal)

Magnitude is:1

500	= PLAY 30X
50	= PLAY 1X
0	= STOP
-50	= - PLAY 1X

Speed data is internally = *magnitude* * 20 (1000 = Play 1X)
Sending a data value of 50 runs effect at PLAY 1X.
Sending a data value of 500 runs effect at PLAY 10X.
Sending a data value of 1500 runs effect at PLAY 30X.

PREROLL **12H**, preroll bytes (4 bytes)

Specify the effect's preroll duration.
byte1 - 4 = BCD format of preroll duration.

SYNC **13H**, sync point (4 bytes)

Mark a synchronize point for the CUE command. In most instances this is the edit inpoint.
byte1 - 4 = BCD format of sync point

CONTROL **14H**, sync point

This command has no impact on the effect but is acknowledged.

CUE **15H**

Cue to a park position determined by the sync point and the preroll.

SEARCH **16H**, search position (4 bytes)

Move the machine to the position specified.
byte1 - 4 = BCD format of search point.

ENTRY **17H**, entry byte (1 byte)

This command has no impact on the effect but is acknowledged.

EXIT **18H**, exit byte (1 byte)
This command has no impact on the effect but is acknowledged.

RDREADY **1AH**
Read ready status.
Always returns ready ON.

RDEE **1BH**
Read EE status.
Always returns EE OFF.

RDCF **1CH**
Read color framer status.
Always returns OFF.

RDEDMODE
1FH
Read edit mode.
Always returns OFF.

RDENABLE **20H**
Read enable status.
Always returns 01H. (video only)

FJOG **25H**
Jog the effect forward 1 frame.

RJOG **26H**
Jog the effect reverse 1 frame.

ACCURACY **27H**, frames (1 byte)
Set the synchronize accuracy.
This command has no impact on the effect but is acknowledged.

TMLOAD **28H**, timer bytes (4 bytes)
Set the time code generator.
This command has no impact on the effect but is acknowledged.

NBASE **29H**
Read machine standard.
byte1 = FFH PAL, 00H NTSC

CFSEL **2AH**, source (1 byte)
Set color framer source.
This command has no impact on the effect but is acknowledged.

RDCFSRC **2BH**

Read color framer source.
Always returns 00H.

TMSEL **2CH**, source (1 byte)

Set the timer source.
This command has no impact on the effect but is acknowledged.

RDTMSEL **2DH**

Read timer source.
Always returns 00H.

EDFIELD **32H**, field (1 byte)

Set the edit field.
This command has no impact on the effect but is acknowledged.

RDEDFIELD **33H**

Read edit field.
Always returns 00H.

SLEW **35H**, frames (3 bytes)

Slew forward or reverse N frames.
byte1 = frames +/- 128.
byte2 & 3 not used.

STILL **36H**, mode (1 byte)

AST playback mode when in still.
byte1 0 = field mode
1 = frame mode

FFJOG **42H**

Jog forward 1 field.

RFJOG **43H**

Jog reverse 1 field.

Effect **30H**

The EFFECT commands are an Accom-only implementation. They are meant to enhance the TRANSPORT commands to give effects devices additional features.

Effect Commands

01H	CHNG_EFF	Change the current effect
02H	SPEED	Set the effect speed
03H	NAME	Set the effect name
04H	GOTO	Goto position
05H	FREEZE	Freeze video

CHNG_EFF **01H**, new effect (1 byte)
Set the effect to the specified value.
byte1 = 0 - 14H

SPEED **02H**, speed value (2 bytes)
Set the effect speed.
byte1 & 2 = speed magnitude (Linear scale in decimal)
Magnitude is:1535= PLAY 3X
511= PLAY 1X
0 = STOP
-511= PLAY 1X

NAME **03H**, name (9 bytes)
Not Yet Implemented.

GOTO **04H**, position (4 bytes)
Goto a position in an effect.
byte1 - 4 = BCD format of goto point.

FREEZE **05H**, state (1 byte)
Set the input video to the specified state.
This command has no impact on the effect but is acknowledged.

CONSOLE EMULATION 74H, address, data

The console emulation command is a specific command that allows certain hardkeys to be emulated from the remote protocol. For instance, the **RUN>** key can be pressed by sending the address for that key and a nonzero data value.
byte1 = console "HARDKEY"
byte2 = data

SET PARAM 80H
Set the internal parameter to a value.

byte1	= flags
byte2	= MSB of Param Number
byte3	= MDSB of Param Number
byte4	= MDSB of Param Number
byte5	= LSB of Param Number
byte6	= MSB of Param Value
byte7	= MDSB of Param Value
byte8	= MDSB of Param Value
byte9	= LSB of Param Value

Sony Protocol

The **SONY** protocol is implemented much like a Sony VTR. However, certain commands that have no bearing on this device are just acknowledged.

The device ID for the DVEOUS has not yet been assigned. However, the ID it uses is F006H, which is the ID for an Abekas A53.

Supported Transport Commands

Stop
Play
FFwd
Jog_Fwd
Var_Fwd
Sht_Fwd
Rew
Jog_Rev
Var_Rev
Sht_Rev
Prog_Speed_Pos
Prog_Speed_Neg
Preroll
Cue_Up

Beginning with software Release 6.1.1, there is a way to use the four data bytes of the Sony VTR Protocol "cue up" command to load effects from registers. This technique assumes you are generating the Sony VTR Protocol from a general purpose computer and custom software of your own design, because it uses an out-of-bounds time code to communicate the effect number to load, a value that might be difficult to generate using third party equipment.

The normal "cue up" command consists of 7 bytes. The first two signify the command (0x24,0x31), the next four are a binary-encoded decimal (bcd) timestamp (frm/sec/min/hr), and the final value is a checksum (the sum of all the previous bytes modulo 256). To indicate the command should load effects instead of cue up the current effect, an out-of-bounds value of 0xff is supplied for the hour. Then the least-significant digit of the effects number (the 1's place) is taken from the least-significant digit of the seconds value, and the most-significant digit of the effects number (the 10's place) is taken from the least-significant digit of the minutes value.

For example, to load an effect from register 0, transmit the following to the port enabled for Sony Protocol under the Remote Enable menu:

```
0x24 0x31 0x00 0x00 0x00 0xff 0x54
```

You will receive an acknowledge of "0x10 0x01 0x11" for any command successfully interpreted.

More examples:

Load Effect from Register #1: 0x24 0x31 0x00 0x01 0x00 0xff 0x55

Load Effect from Register #10: 0x24 0x31 0x00 0x00 0x01 0xff 0x55

Load Effect from Register #90: 0x24 0x31 0x00 0x00 0x09 0xff 0x5d

Load Effect from Register #99: 0x24 0x31 0x00 0x09 0x09 0xff 0x66

Be aware that an improperly formatted command or invalid checksum will cause a NACK to be returned (negative acknowledge -- this should be the three byte string 0x12 0x04 0x16, but you might see a 0x11 proceeding these). After a NACK you may have to reset the machine to restore proper functioning of the remote port.

Peripheral Bus I and II Interface Support

The DVEOUS supports limited Peripheral Bus I and II command protocols as listed below:

Peripheral Bus II Commands Supported

- Learn
- Recall
- Trigger
- Query

Please refer to your Grass Valley Group Protocol manual for information on how to use these GVG Peripheral Bus II Commands.

Peripheral Bus I Commands Supported

- Learn
- Recall

Please refer to your Grass Valley Group Protocol manual for information on how to use these GVG Peripheral Bus I Commands.

Crosspoint Aux Bus Command —ABEKAS SPECIAL

The CROSSPOINT AUX BUS command changes the crosspoint on an Aux Bus of a GVG switcher.

Function	Effect Address	Command Code	Message Byte
Abekas 8100/8150:			
Aux Bus	07H	C1H	Crosspoint #
GVG Model 200 Switcher:			
Aux Bus 1	07H	C1H	Crosspoint #
Aux Bus 2	07H	C2H	Crosspoint #
Aux Bus 3	07H	C3H	Crosspoint #
Aux Bus 4	07H	C4H	Crosspoint #

(continued on the next page)

Function	Effect Address	Command Code	Message Byte
GVG Model 300 Switcher:			
Aux Bus 1	04H	C1H	Crosspoint #
Aux Bus 2	04H	C2H	Crosspoint #
Aux Bus 3	04H	C3H	Crosspoint #
Aux Bus 4	04H	C4H	Crosspoint #
GVG Model 3000 Switcher:			
Aux Bus 1A	0CH	C1H	Crosspoint #
Aux Bus 1B	0DH	C1H	Crosspoint #
Aux Bus 2A	0EH	C1H	Crosspoint #
Aux Bus 2B	0FH	C1H	Crosspoint #
Aux Bus 3A	10H	C1H	Crosspoint #
Aux Bus 3B	11H	C1H	Crosspoint #
Aux Bus 4A	12H	C1H	Crosspoint #
Aux Bus 4B	13H	C1H	Crosspoint #
Aux Bus 5A	14H	C1H	Crosspoint #
Aux Bus 5B	15H	C1H	Crosspoint #
Aux Bus 6A	16H	C1H	Crosspoint #
Aux Bus 6B	17H	C1H	Crosspoint #
Aux Bus 7A	18H	C1H	Crosspoint #
Aux Bus 7B	19H	C1H	Crosspoint #

A53 (Limited) RS-232 Control Command Set

DVEOUS can be controlled by a personal computer or other controller via the DVEOUS RS-232 port. Commands supported by this interface are:

Command	Parameter	Type	Size	Scaling	Min	Max	Normal
EFFECT `f`	effect number	absolute	int	—	0(workspace)	24	
TRESET 0x0b	learn new T-bar limits	switch	int	—	any value		
TFRACT 't'	T-bar position value	absolute	int	1	0	32000	
MAN 'n'	manual mode	switch	int	—	0=off	1=on	0
RUN `r`	effect run control	switch	1st byte		0=don't care 1=forward	2=reverse	
	effect step control		2nd byte		don't care		
BREAK 'q'	run break control	switch	1st byte		0=don't care 1=on	2=off	
	programmed break attribute		2nd byte		don't care		

Serial data sent to the RS-232 port consists of characters formatted as 8 bits plus one parity bit. (Parity is even.) The host sends RS-232 data to the port. The final carriage return (CR) completes the message packet. At the field interrupt after a completed message DVEOUS will send a status message back to the host. It is not necessary for the host to wait for completion of the status reply before sending another command.

Status Message format:

byte 1	low byte of character count
byte 2	high byte of character count (total bytes transmitted)
byte 3	command constant
byte 4	command value
byte 5	command value
byte n	command constant
byte n+1	command value
byte n+2	command value
byte n+3	command value
last byte	carriage return (0d Hex)

The command constant specifies the type of command and the command value bytes give a numerical parameter associated with the command. There can be from two to four command value bytes, depending on the particular command.

Control Point Language

This protocol allows a Grass Valley Group Model 3000 switcher to control a DVEOUS via a protocol converter board or external controller. In operation this makes DVEOUS look like a Krystal effects device to the switcher. For information on Control Point Language, refer to that manufacturer's protocol manual. The following material documents the protocol between DVEOUS and the protocol converter (which is not Control Point Language).

The RS-422 interface between the DVEOUS and protocol converter runs at 76.8K baud, with odd parity. Data block sizes vary with different messages.

Message Format

Byte Count	1byte
Message Token	1byte
Message Data	0-253 bytes
Checksum	1 byte

Byte Count	=	Message Token + Message Data
Checksum	=	Byte Count + Message Token + Message Data

Message Tokens

SPCL_SET	0x08
SPCL_ONLINE	0x03

Communications

Communication with the external controller must be established before message transactions can occur. This is initiated by the controller sending a SPCL_ONLINE message. If DVEOUS is connected properly it will respond with an SPCL_ONLINE message.

Host	SPCL_ONLINE	
DVEOUS	SPCL_ONLINE	

Timeline

After communication is established, the controller sends SPCL_ONLINE messages once each field. DVEOUS responds at the beginning of the next field with an SPCL_SET command that gives the Aux Bus source and current effect. (See the SPCL_SET definition later in the Command Message format discussion.) This action occurs within the first 3mS of the field and continues until the controller stops sending SPCL_ONLINE messages. The SPCL_ONLINE messages from the controller should occur after the first 3mS.

Host	SPCL_ONLINE	SPCL_ONLINE	....
DVEOUS	SPCL_ONLINE	SPCL_SET	SPCL_SET ....

Timeline

Once this communication has been established, any message token can be sent. These messages can be sent at anytime during the field with processing of the message beginning on the next field.

Command Message Format

This topic discusses the format of command messages sent from DVEOUS to the external converter and from the External converter to DVEOUS.

DVEOUS to External Host

SPCL_ONLINE

This message is used to acknowledge the Host when connection is being established.

Format: 0x01, SPCL_ONLINE, Checksum

SPCL_SET

This message is a per field message issued once connection is established between DVEOUS and the external converter.

Format: 0x10, SPCL_SET, MD0...MD14, Checksum

Message data has the following format:

Byte 0	Front Source for Aux Bus 1A
Byte 1	Back Source for Aux Bus 1A
Byte 2	Front Source for Aux Bus 1B
Byte 3	Back Source for Aux Bus 1B
Byte 4	Front Source for Aux Bus 2A
Byte 5	Back Source for Aux Bus 2A
Byte 6	Front Source for Aux Bus 2B
Byte 7	Back Source for Aux Bus 2B
Byte 8	0, reserved
Byte 9	0, reserved
Byte 10	On-Air Aux Bus 1A
Byte 11	On-Air Aux Bus 1B
Byte 12	On-Air Aux Bus 2A
Byte 13	On-Air Aux Bus 2B
Byte 14	Current Effect Loaded

Crosspoint selection for any Aux Bus occurs on changes to the Front Source selection. It is required that the Source should switch on the next field boundary after receiving this message. Valid sources are 0 to 64.

The Back Source information can be used for status information.

On-Air is a Boolean to indicate that the Aux Bus is contributing to the final output for the DVEOUS picture.

Current Effect loaded is the last effect recalled into the DVEOUS workspace buffer. Valid range of numbers:

-1	= No effect has been loaded
0...99	= effect number.

External Host to DVEOUS

SPCL_ONLINE This message is used to establish communication with DVEOUS. Once the link has been established this message is sent every field to keep it alive. The external controller should expect an SPCL_SET from DVEOUS on the next field boundary. If it isn't received, you can assume that communication from DVEOUS has dropped.

SPCL_SET This message is used to deliver commands from the external controller to DVEOUS. This command has sub-messages.

Format:

Byte Count	1 byte
Message Token	1 byte
0x1, Aux Bus 1	1 byte
0x2, Aux Bus 2	1byte
MSB Sub Message Token	1 byte
LSB Sub Message Token	1 byte
Sub Message Data	Byte Count - 5
Checksum	1byte

Sub Message Tokens

PID_EFFECT_POSITION	0x036b (875)
PID_A_SIDE_SOURCE	0x0A52 (2642)
PID_B_SIDE_SOURCE	0x0A53 (2643)
PID_LOAD_EFFECT	0x0376 (886)

PID_EFFECT_POSITION

This command sends an S15.16 frames value to set the current position of the effect. The effect position will be reached 7 fields after receiving the command. ***Note:*** DVEOUS rounds to the nearest field value.

Sub Message Format

0x00	reserved
0x00	reserved
0x00	reserved
0x00	reserved
0x00	reserved
0x00	reserved
0xMSB	
0xLSB	S15.16 frames

PID_A_SIDE_SOURCE

This command sets the source for the A side of either Aux Bus 1 or 2. The Aux Bus change will be sent back to the external controller via the SPCL_SET in the next field.

Sub Message Format

0x00	reserved
0x00	reserved
0x00	reserved
Aux Bus Number	0x01 = Aux Bus 1, 0x2 = Aux Bus 2
0x00	reserved
0x00	reserved
0x00	reserved
Source Number	0..64
0x00	reserved
0x00	reserved

PID_B_SIDE_SOURCE

This command sets the source for the B side of either Aux Bus 1 or 2. The Aux Bus change will be sent back to the external controller via the SPCL_SET sent in the next field.

Sub Message Format

0x00	reserved
0x00	reserved
0x00	reserved
Aux Bus Number	0x01 = Aux Bus 1, 0x2 = Aux Bus 2
0x00	reserved
0x00	reserved
0x00	reserved
Source Number	0..64
0x00	reserved
0x00	reserved

PID_LOAD_EFFECT

This command causes DVEOUS to load an effect from the specified register.

Sub Message Format

0x00	reserved
0x00	reserved
0x00	reserved
0x01	Enable Load
0xMSB	
0xLSB	Effect Number, -1....99

Abekas LINC (slave) Protocol

The A82 and A83 Digital Switchers support a control protocol known as LINC^{TM}. This protocol appears generically as ***MULTI-DROP*** in the A83/A82 Remote Setup menu. LINC can be used to control the DVEOUS effects device as a slave device. The following documentation is reprinted from the A82/A83 Protocol documentation for your convenience.

Overview

LINC allows you to control and/or program as many as 32 remote devices from the A83/A82 Control Panel. An environment for developing custom interfaces, LINC makes much of the A83/A82's user interface (EL displays with softkeys and knobs, **PERIPHERAL CONTROL** hardkeys, joystick, and trackball) available for controlling remote devices. Within certain restrictions, explained below, the programmer can use this framework for developing custom interfaces and user menus.

There are two reasons for connecting devices to the A83/A82. Field-based devices, such as digital disk/ram recorders and digital effects devices, may be controlled as slaves to the A83/A82. The general intent here is to lock them to the A83/A82's effects timeline. This also lets the operator play, stop, and jog these devices with buttons in the **PERIPHERAL CONTROL** area of the A83/A82's Control Panel. Another class of device that supports a subset of the field-based machine command set might be a character generator, routing switcher, or color corrector. Such devices cannot normally be “run”, but effects register saves and learns performed at the A83/A82 Control Panel can “learn” and recall their current status (setup).

The second purpose for connecting a device to LINC is to program the device itself from the A83/ A82. In this mode the A83/A82 acts as a slave to the remote device. This may simply be an operator convenience, but with menu simulation, it may make a control panel unnecessary, saving edit bay space and perhaps money (if the remote device may be purchased without a control panel). Programming is via a soft menu with 23 softkeys and 11 softpots that provide relative motion indications. The remote device may also use the A83/A82 joystick and trackball.

LINC Specifications

Electrical specifications for LINC call for RS-422 differential transmission and reception. This is a four wire and ground system. Remote device outputs are to be tri-stated when they are not talking on the network, and when they are powered down. The baud rate is fixed at 38.4kbd. The remote device must transmit with 8 data bits, 1 stop bit, and even parity. The A83/A82 transmits “control” bytes with odd parity and data bytes with even parity. The remote device must distinguish between reception of control and data bytes.

Data Format

A remote device must be polled before it may speak on the network. Polling involves receiving a control byte with data equal to its address. Remote devices on the network must all have unique addresses between zero and 31. It is convenient to assign this address with a dip switch or hex thumbwheel. When a device is polled, it may remain silent if it has no commands or status for the A83/A82. It may reply with an overall byte count (the number of bytes to follow) and a mix of commands and/or status replies. The device may send multiple commands in one go, but if a status reply is mixed with commands the status reply *must be* the first data transmitted. Many commands sent from the A83/A82 to the remote device solicit a reply. We do not guarantee that the status request will be sent first. Note, however, that failure to reply to a status request for a few fields has no dire consequences. The best approach is to reply to status requests only when there is nothing else to send. Do not queue status requests; just reply when possible.

Note that under most circumstances every field-based device is polled once per TV field. You can send a small amount of data to the A83/A82 without bad consequences; for example, enough data to update one softkey. However, sending volumes of data whenever polled isn't recommended: it prevents the A83/A82 from talking to other nodes, prevents other nodes from talking to the A83/A82, and generally slows down the network. Also, take care not to ignore pleas for status from the A83/A82—or it may declare you dead. Status replies are used to update your rolling timecode displays on the A83/A82, which become irrelevant if they are not updated every few frames or so.

The A83/A82 sends data destined for a remote device in one of two ways. The data may be multicast or sent to an individual device. Individual device data is all the data bytes following the poll byte for this device up to the next `control' byte. No byte count is provided, as the message is bracketed by control bytes. Multicast data is prefaced by the MULTICAST control byte (0x40) and a four byte multicast mask. "1" bit in the first byte of the multicast mask are enables for remote devices 24 (LSB) through 31 (MSB). "1" bit in the fourth byte of the multicast mask are enables for remote devices 0 (LSB) through 7 (MSB). If a remote device is included in the multicast group all the data bytes following the last multicast mask byte are destined for this device as well as others up to the next control byte.

Communication on LINC is synchronous to reference field at the A83/A82. It is possible for transmissions in either direction to proceed across field boundaries. However, the A83/A82 transmit state machine always begins at top of field. The sequence of events, beginning at the top of field, is as follows:

```
TOP OF FIELD:
SYSTEM CLOCK BYTE, a control byte, data in range 32 through 63.
optional multicast 1:
            MULTICAST BYTE, a control byte, data = 0x40
            multicast mask byte 0;
            multicast mask byte 1;
            multicast mask byte 2;
            multicast mask byte 3;
            multicast data 0;
            multicast data 1;
            multicast data n;
end optional multicast 1 message:
optional multicast n:
end optional multicast n message:
            POLL BYTE n,'control' byte    /*poll bytes for all live nodes,
                                            order unspecified*/
            POLL BYTE n,'control' byte
            POLL BYTE n,'control' byte
            POLL BYTE n,'control' byte
            ETX BYTE, a 'control' byte, data = 0x41
```

Note that you can disable the polling of certain devices in the A83/A82 Remote Setup menu. Also note that LINC slave devices should not depend on the order of the above occurrences; it is given for informational purposes. However, the SYSTEM CLOCK BYTE is guaranteed to occur with a fixed relationship to (just into) the field. This is a convenient time to freeze certain receive or transmit data and can also be used to synthesize a reference vertical time that software may use. As mentioned above, if the LINC master has data for a particular remote device it immediately follows the remote device's poll byte. If the polled remote device has data for the LINC master it transmits it upon receiving its poll byte. The LINC master senses the beginning of the startbit sent by the talkative remote device and aborts the poll byte for the next node (presumably now partially sent). The master will not proceed to poll the next node until the remote device has finished talking. If data must be exchanged in both directions, simultaneous transfers will occur, with the longest one determining when polling resumes. Remote devices with data to transmit MUST respond VERY quickly (less than or equal to 120uS) to reception of a poll byte. This guarantees that the poll byte for the next node is aborted soon enough to make it invalid.

The LINC master must know the device ID assignment (assigned by Abekas) and two phrases describing your device. The first phrase is a long description (up to 32 characters), the second is a short description (up to 8 characters). We need to know if your machine is field-based, programmable, or both.

All machines connected to LINC must respond to the REQ_STATUS command. For field-based machines we expect a four byte status reply as described later in this section. Machines that are both field-based AND programmable have their own position in the Remote Device Programming menu to show lexically more information about their status than the above. For example, an effects machine might want to include the current keyframe number. If you want to use this capability we need to know where the additional information is packed into the 24 bit status reply and how to display it.

The format of the two byte status reply for non-field-based machines appears later in this section.

LINC Commands

Field-based devices that want to slave to the LINC master's timeline should interpret five LINC commands:

SEEK_OFFSET:

Sets timewise offset for SEEK_W_OFFSET and RUN_W_OFFSET commands. The offset number is given in fields. This number is subsequently added to all the numbers given by SEEK_W_OFFSET commands. If a device is seeked past its physical limitations, the seek should be limited to this boundary. For example, if a device incapable of seeking before 0 is seeked to -90 fields, the device should park at zero. A following RUN_W_OFFSET command should cause the device to play 90 fields after receiving that command.

SEEK_FIELD:

Device seeks to field number given.

RUNCMD:

Device runs at speed given from current position.

SEEK_W_OFST:

RUN_W_OFST:

Same as SEEK_FIELD and RUNCMD, but takes offset (described above) into account.

Devices that want to use the A83/A82's learn-recall capability need to support the following three commands.

For machines that are to run in sync with the A83/A82 timeline we need to know your roll delay: the time from receipt of a run forward command to a response. We have yet to determine the maximum pipeline delay to allow for the slowest device on the network. When this is determined it will be up to you to delay run commands to match this slowest device. If you are slower than eight fields you are likely to be left in the dust.

PERIPH_LEARN:

Indicates the A83/A82 operator has saved effects into the effects register specified in this command. The LINC slave MUST respond to this command with a PERIPH_DATA reply. Note that the LINC slave may chose to store its current state internally for later recall, but it still must reply with the four byte PERIPH_DATA command.

PERIPH_RECALL:

Indicates the A83/A82 operator has recalled effects from the effects register specified in this command. The sixteen bits of PERIPH_DATA given in this command is that which was sent earlier during the learn for this device.

PERIPH_DATA:

Four byte command sent from slave to the A83/A82 in response to a PERIPH_LEARN command. This command contains 16 bits of data to be echoed to the device in a later PERIPH_RECALL command. The 16 bits of data MUST be non-zero.

Programmable devices that want to use the A83/A82's joystick or trackball may do so using the following two commands:

TBALL_LEGEND:

The slave sends this command to the A83/A82 to take control of the trackball. The eight character legend given in this command appears above the trackball. If the string is not null, trackball movement is reported to the slave device using the TBALL_MOVE command, which contains the XY deltas.

JOY_LEGEND:

The slave sends this command to the A83/A82 to take control of the joystick. The eight character legend given in this command appears below the joystick. If the string is not null, joystick movement is reported to the slave device using the JOY_MOVE command, which contains the XYZ deltas.

Programmable devices may use the A83/A82 Remote Device menu to provide operator displays and acquire operator input. Operator input for an LINC slave may use the following:

• 23 softkeys

• 11 softpots (See SOFTP_MOVE command)

• A selection of A83/A82 hardkeys that are assigned to the Remote Device menu when it is selected.

Button presses, button releases, and softpot moves are reported to the LINC slave via the following commands:

KEY_PRESS:

KEY_RELEASE:

SOFTP_MOVE:

A sample header file that defines the button codes appears in Appendix D.

Operator displays on the A83/A82 Remote Device menu for the LINC slave are created with the following two commands:

SOFTKEY:

This displays up to 23 softkeys legends. Softkeys may be alpha only or alphanumeric. Alpha only buttons permit up to 12 characters. Alphanumeric buttons left justify the alpha portion and right justify the number. Numbers may be whole, fractional, or time numbers. Whole numbers range from -9999 to 9999; fractional numbers range from -32.768 to 32.767; time numbers appear as XX:XX:XX and range from 00:00:00 to 18:20:??. Softkey legends may be white on black or black on white (inverted).

ABA_TTY:

This command writes text to the area below the top row of softkeys and above the bottom row softkeys is available for text display. The display area is divided into two sides; there are two EL displays, 11 rows, and 80 columns.

Note that when the A83/A82 descends into the Remote Device menu it sends a REQ_SFTKEYS command to the LINC slave. When it sends this command, both EL displays are blank except for a status line on the top left of the left EL display and a ***MORE*** softkey at the bottom right of the right display.

LINC Status Reply Format

All devices on the network are subject to periodic status requests from the LINC master. An attempt is made to request status of field-based devices as frequently as possible to update rolling timecode displays at a reasonable rate. (This may not always be possible.) All non-field-based and non-existent devices get status requests a couple of times a second to determine what they are and that they exist.

The format of the status reply for a non-field-based device is two bytes. This does not include the overall byte count, which would be 2 if no commands follow the status reply. The reply is:

<BYTE COUNT><0x80><DEVICE_IDENTITY>.

The format of the status reply for a field-based device is more complicated.

<BYTE COUNT><DEVICE_IDENTITY><PROPRIETARY DATA : LOC MSB><LOC LSB><MODE : PROPRIETARY DATA>

The number of unused bits in this 24 bit reply depends on the number of bits used in location msb. For example, only two bits would be used to express locations up to 1023 fields (17 seconds +); the six remaining bits could be used to convey other proprietary status. The two mode bits in the last byte are interpreted as follows:

BIT6: Run reverse.

BIT7: Run forward.

Neither: Stopped.

The remaining bits are available.

LINC Command Format

NOTE: all int16 values are transmitted MSByte first, as with a 68000.

LINC Commands from A83/A82 to Remote

```
typedef struct
{
    char command;
    char keynum;
    long num_val;
    char decpt;/*number of places from right of number for decimal point*/
} Key_press;

#define KEY_PRESS1

typedef struct
{
    char command;
    char keynum;
    long num_val;
    char decpt;/*number of places from right of number for decimal point*/
} Key_release;

#define KEY_RELEASE2
```

For programmable machines. These commands are directed at a programmable device only when the Remote Device menu is selected on the A83/A82. The 23 softkeys (UKEY_SK0 though UKEY_SK22) and selected `generic' editing buttons respond. The keynum defines are in a separate include file. The generic buttons are SAVE, RECALL, EFFECT, MODIFY, COPY, INSERT, DELETE, GOTO, START, TO, END, ALL, MASTER, TENS (tension), DUR (duration), PREV (previous), NEXT, and THIS.
If the operator typed a number in the numeric keypad area of the A83/A82 before striking the button, the number is passed on in NUM_VAL in the range -0x7fffffff to +0x7fffffff. The number entry is then cleared.
Otherwise this value is 0x80000000.

```
typedef struct
{
    char command;
    char offset[3];/*viewed as 24 bit signed num*/
} Seek_offset;

#define SEEK_OFFSET3
```

For field-based machines. This command sets a permanent offset from the physical locations specified by the SEEK_W_OFST and RUN_W_OFST commands below. For example, a seek to location 0 with current offset of -30 will seek to timecode 23:59:59:15. A RUN_W_OFST command at normal forward speed (PLAY1X) will not cause the machine to advance for 15 frames.

```
typedef struct
{
    char command;
    int16 field;/*BIT14 means relative seek*/
} Seek_field;

#define SEEK_FIELD4
```

For field-based machines. This command causes the machine to go to a new location.

If BIT14 of the field argument is set, the location is taken to be relative to the current location. For example, 0x4002 means move 2 fields forward. A relative seek of 0 should be interpreted as a stop command.

```
typedef struct
{
    char command;
    int16 field;/*BIT15 means SEEK CUE*/
} Seek_w_ofst;

#define SEEK_W_OFST13
```

For field based machines. This command causes the machine to go to the location specified by field PLUS the last defined SEEK_OFST. BIT15 of the field may usually be ignored; it is set TRUE when a machine is precued before a sync roll. For example, a RUN_W_OFST command is coming after a fixed internal delay and color framing delay.

```
typedef struct
{
    char command;
    int16 runspeed;
} Runcmd;

#define RUNCMD5
#define RUN_W_OFST14
```

For field-based devices. This command causes the device to run at the speed specified in runspeed. Normal forward runspeed is defined as 1000 base 10. For example, if runspeed is -12981 the device should run in reverse at 12.981 times normal speed.

```
#define PLAY1X1000

typedef struct
{
    char command;
    char delta_x;
    char delta_y;
    char delta_z;
} Joy_move;

#define JOY_MOVE7
```

For programmable devices that have requested use of the joystick via the JOY_LEGEND command. These are the absolute values representing deflection of joystick from its rest position.

```
typedef struct
{
    char command;
    char delta_x;
    char delta_y;
} Tball_move;

#define TBALL_MOVE8
```

For programmable devices that have requested use of the trackball via the TBALL_LEGEND command. These are the relative values representing movement of trackball from its rest position.

```
typedef struct
{
    char command;
    char softp;
    char delta_x;
} Softp_move;

define SOFTP_MOVE9
```

For programmable devices. This command results from twisting the softpot softp when the A83/A82 is in the Remote Device menu. Negative values of DELTA_X reflect counterclockwise movement.

```
typedef struct
{
    char command;
    char spare;/*commands must be 2 bytes minimum*/
} Req_status;

#define REQ_STATUS10
```

A83/A82 sends this command to ask for status from remote nodes. If there is no response for 8 seconds the A83/A82 assumes this node is dead. The A83/A82 issues this command about twice a second for dead or non-field-based devices. It issues the command much more frequently for field-based machines.

```
typedef struct
{
    char command;
    char spare;
} Req_sftkys;

#define REQ_SFTKYS11
```

The A83/A82 sends this command to programmable devices when the Remote Device menu is first entered. It is appropriate for the remote device to respond with a full set of softkeys and requests for the joystick or trackball as required.

```
typedef struct
{
    char command;
    char regnum;
} Periph_learn;

#define PERIPH_LEARN15
```

The A83/A82 multicasts this message to all field-based devices when a peripheral learn occurs. Peripherals that are capable and willing to later recall a given page, effect, state, setup, etc., should reply with PERIPH_DATA command.

```
typedef struct
{
    char command;
    char regnum;
    int16 periph_data;
} Periph_recall;

#define PERIPH_RECALL17
```

The A83/A82 multicasts this message to all field-based devices when a peripheral recall occurs. Remote devices may honor this command based on either the 16 bits of peripheral data supplied (an echo of what they sent during a learn) or on the register number (regnum) if the learn/recall information is kept internal to the device.

Commands/Responses from Slaves to the A83/A82

```
typedef struct
{
    char command;
    char dev_addr;
    char legend[8];
} Tball_legend;

#define TBALL_LEGEND3
```

A remote, programmable device sends this command to gain use of the A83/A82 trackball. The legend appears in the 8 character display above the trackball.

```
typedef struct
{
    char command;
    char dev_addr;
    char legend[8];
} Joy_legend;

#define JOY_LEGEND4
```

A remote, programmable device sends this command to gain use of the A83/A82 joystick. The legend appears in the 8 character display above the joystick.

```
typedef struct
{
    char sknum;   /*0 through 22*/
    char sk_format;/*XXXY ZZZZ, X = NUM_FORMAT, Y = INVERT,
                              Z = STRLEN, strlen = 0 = Keyword*/
    char alpha[];   /*string of length strlen, NOT null terminated*/
    int16 number; /*Omitted if NUM_FORMAT is NONE*/
} SK_DESCRIP;
```

NOTE: if strlen is equal to zero the next byte is taken to be an integer that defines an Abekas specific keyword. This feature is used to reduce the bandwidth required to send words; it is not available to foreign manufacturers, EXCEPT note that keyword 0xff erases an existing softkey. For aesthetic reasons blank softkeys should be erased, but you can send empty softkeys by sending the one character string " ".

```
#define NUM_NONE0x00
#define NUM_WHOLE0x20/*-9999 to 9999*/
#define NUM_FRACT0x40/*WW.FFF if >= 10000, W.FFF if >= 1000,
                              0.FFF if < 1000*/
#define NUM_TIME0x60/*number is frames displayed as XX.XX.XX*/
#define SK_INVERT0x10/*reverse video softkey*/
```

```
typedef struct
{
    char command;
    unsigned char bcnt;/*True size of this structure, i.e., number of
                                bytes to follow + 2*/
    char dev_addr;
    char spare;
    SK_DESCRIP skdef[];
} Softkey;

#define SOFTKEY22
```

Programmable devices use this command to update softkeys in the A83/A82 Remote Device menu. There are 23 available softkeys, numbered 0 through 22. Any or all softkeys may be updated with this command.

```
typedef struct
{
    char command;
    unsigned char bcnt;/*real size of this structure as sent*/
    char dev_addr;
    char column;  /*0 - 79*/
    char row;     /*left display: rows 0 - 10; right display rows
                                64 - 74*/
    char string[];
} Aba_tty;

#define ABA_TTY23
```

Command used to write free form text and numbers on an EL display. If row is < 0 the last selected row is retained; if column is < 0 the last imaginary cursor position is used. Writing off the right edge of the screen produces undesirable results. Top bit set characters in the string are interpreted as escape sequences. Use the TTYNUM macro with the NUM_FRACT, NUM_WHOLE, or NUM_TIME argument to prefix a 16 bit number. Characters in the range 0xb0 through 0xff expand into spaces. 0xb9 expands to nine spaces.

```
#define TTYNUM(a) (0x80 + ((a) >> 5))

typedef struct
{
    char command;
    char dev_addr;
    int16 periph_data;/*must be non-zero*/
} Periph_data;

#define PERIPH_DATA6
```

Sent in response to a PERIPH_LEARN command, field-based devices only. The field-based device has the choice of storing `learn' data internally by register number (see PERIPH_LEARN command) or sending 16 bits of data to later be echoed to it with the PERIPH_RECALL command. If the device opts to store the data internally it should still respond with PERIPH_DATA not equal zero.

LINC Protocol for a Slave

```
"C"
"Z80"

$SEPARATE$
$RECURSIVE OFF$
$SHORT_ARITH ON$

#define uint8unsigned char
#define int16int

#include "/users/stu/A82v2/include/aba_com.h"

/* This is an example of an implementation of the LINC protocol from the slave side.
The use of a Zilog Z80 DART is presumed. Code is specific to an 8 bit microprocessor
with byte ordering like the Z80. This code is currently in use. */

extern int outport();/*send a byte out Z80 I/O port*/

typedef union
{
    char ch[2];      /* ch[0] = lsbyte */
    int i;
}  CHINT;

typedef union
{
    char ch[4];      /* ch[0] = low byte */
    int i[2];                          /* i[0] = low half */
    long l;
}  CHINTL;

typedef struct
{
    uint8 getptr;
    uint8 putptr;
    uint8 auxptr;
    uint8 spare;
    uint8 buffer[256];/*buffer size chosen to make wrap easier to
                                  handle*/
}  CIRC_BUFFER;

CIRC_BUFFER Receive_buffer;
CIRC_BUFFER Transmit_buffer;
uint8  Rx_state;
uint8  Aba_error;/*set on error in reception*/
CHINTL Aba_status;/*4 or 2 byte status reply*/
uint8 System_clock;/*A83/A82's field clock as recieved*/
uint8 Myaddress;/*0 - 31*/
uint8 Tx_lock;/*semaphore*/
uint8 Tx_pend;/*number of tx bytes pending as of last sysclock byte*/

#define QUIESCENT0
#define CTL_MCAST64
#define RX_IN_PROGRESS5

aba_rxint (rdata)
CHINT rdata;/*ch[0] is rxdata, ch[1] is TRUE if parity error*/
{
    static uint8 rxdata;
    static CHINTL multicast_bits;
    register uint8 nbytes;

    rxdata = rdata.ch[0];/*LSB is data*/
    if (rdata.ch[1]) /*this is a control byte*/
    {
```

/* If we have outstanding receive stuff this is the time to make it available to background process as receive messages are always bracketed by control bytes. Data is made available as a `chunk' by copying auxptr to putptr so that background process gets the full packet. A valid message must be 2 bytes or more as an invalid data byte could be generated by aborting the next node's poll. */

```
            if (Rx_state == (RX_IN_PROGRESS + 1))/*one byte message*/
                    Recieve_buffer.auxptr--; /*reject, minimum msg is 2 bytes*/
            else if (Rx_state > (RX_IN_PROGRESS + 1))
                    Recieve_buffer.putptr = Recieve_buffer.auxptr;

            if (rxdata == Myaddress)/*I have been polled*/
            {
                    if (Tx_pend)    /*I have data to tx, get it out quick*/
                    {                       /*Only have 100uS to do it, this may not be
qik enuf*/
                                    outport (DARTACTL, 5);/*enable buffer (RTS)*/
                                    outport (DARTACTL, 0xea);
                                    outport (TXADATA, Tx_pend);/*send bytecount
byte*/
                                    Tx_pend = 0;
                    }
                    Rx_state = RX_IN_PROGRESS;
                    return;
            }

            Rx_state = 0;/*reset state to quiescent*/
```

/* On sysclock byte (top of field) prepare to enable the tri-state buffer and send our first byte of transmission if we have anything to send. This works with aba_init_tx in assembly language, Z80 is not fast enough to respond in `C'. */

```
                if (rxdata >= MAX_NODES && rxdata < CTL_MCAST)/*sysclock byte*/
                {
                        System_clock = rxdata & 31;
                        if (Aba_status.ch[0] && Transmit_buffer.getptr ==
                                                Transmit_buffer.auxptr && !Tx_lock)
                        {
                                        Transmit_buffer.buffer[Transmit_buffer.auxptr++] =
                                                        Aba_status.ch[0];
                                        Transmit_buffer.buffer[Transmit_buffer.auxptr++]=
                                                        Aba_status.ch[1];
#if FIELD_BASED/*you should have four bytes to send*/
                                        Transmit_buffer.buffer[Transmit_buffer.auxptr++]=
                                                        Aba_status.ch[2];
                                        Transmit_buffer.buffer[Transmit_buffer.auxptr++]=
                                                        Aba_status.ch[3];
#endif
                                        Aba_status.ch[0] = 0;
                        }
                        Transmit_buffer.putptr = Transmit_buffer.auxptr;
                        Tx_pend = 256 - (Transmit_buffer.getptr - Transmit_buffer.putptr);
                        if (Aba_error)
                                        Recieve_buffer.auxptr = Aba_error =
Recieve_buffer.getptr =
                                                        Recieve_buffer.putptr = 0;
                }
                else if (rxdata == CTL_MCAST)/*prepare for four addr bytes*/
                        Rx_state = 1;
        }
        else            /*parity OK, this is a data byte*/
        {
                if (Rx_state >= RX_IN_PROGRESS)/*just keep on putting in the data*/
                {
                        Recieve_buffer.buffer[Recieve_buffer.auxptr++] = rxdata;
                        Rx_state++;
                }
                else if (Rx_state)
                {

/* My Z80 compiler has problems with longs, you can do something more sensible. */

                        multicast_bits.ch[3] = multicast_bits.ch[2];
                        multicast_bits.ch[2] = multicast_bits.ch[1];
                        multicast_bits.ch[1] = multicast_bits.ch[0];
                        multicast_bits.ch[0] = rxdata;

/* If the multicast bit field is fully assembled see if we are involved. If so things proceed
normally
(Rx_state now == RX_IN_PROGRESS) else go quiet.*/
```

```
                    if (++Rx_state == RX_IN_PROGRESS)
                    {
                                if (!(multicast_bits.ch[Myaddress >> 3] & (1 << My-
address & 7)))
                                        Rx_state = QUIESCENT;/*not destined for
me*/
                    }
            }
    }
}
aba_txint()
{
    /*Disable transmitter if last byte, RTS output tri-states buss*/

    if (Transmit_buffer.putptr == Transmit_buffer.getptr)
    {
            outport (DARTACTL, 0x28);   /*no furthur interrupts*/
            outport (DARTACTL, WR5);    /*select write reg 5*/
            outport (DARTACTL, WR5DAT ^ RTS);/*tri-state output driver, end
                                                        of this byte*/
    }
    else
            outport (TXADATA, Transmit_buffer.buffer[Transmit_buffer.getptr++]);
}

aba_tx (nbytes, data)/*que data in transmit buffer*/
int nbytes, *data;
{
    register int to_wrap;
    register int bcnt;

    bcnt = nbytes;
    Tx_lock = 1;
    to_wrap = Transmit_buffer.auxptr;
    to_wrap = 256 - to_wrap;

    if (bcnt <= to_wrap)
            cmove (data, &Transmit_buffer.buffer[Transmit_buffer.auxptr], bcnt);
    else
    {
            cmove (data, &Transmit_buffer.buffer[Transmit_buffer.auxptr], to_wrap);
            cmove ((int)data + to_wrap, Transmit_buffer.buffer, bcnt - to_wrap);
    }
    Transmit_buffer.auxptr += bcnt;
    Tx_lock = 0;
}

/* A good time to call deque_aba_cmds is at top of field for field based devices. The
SYSCLOCK byte doesn't happen until about 2ms into the field so this is usually a quiet
period. Could also be called out of the receive state machine at ETX byte time or
SYSCLOCK byte time after reenabling interrupts. */
```

```
deque_aba_cmds()
{
    while (Recieve_buffer.getptr != Recieve_buffer.putptr)
    {
            switch (Recieve_buffer.buffer[Recieve_buffer.getptr++])
            {
                    case REQ_STATUS:
                                    Recieve_buffer.getptr += 1;/*discard 2nd byte*/
                                    Aba_status.l = ???;/*format status with current po-
sition*/
                                    break;

                    case KEY_PRESS:
                                    Key_code Recieve_buffer.buffer[Recieve_buffer.getptr++];
/*example*/
                                    Key_value.ch[3] = Recieve_buffer.buff-
er[Recieve_buffer.getptr++];
                                    Key_value.ch[2] = Recieve_buffer.buff-
er[Recieve_buffer.getptr++];
                                    Key_value.ch[1] = Recieve_buffer.buff-
er[Recieve_buffer.getptr++];
                                    Key_value.ch[0] = Recieve_buffer.buff-
er[Recieve_buffer.getptr++];
                                    Recieve_buffer.getptr++;/*we don't need decpt*/
                                    break;

                    case KEY_RELEASE:
                    case SEEK_FIELD:
                    case RUNCMD:
                    case TBALL_MOVE:
                    case SOFTP_MOVE:
                                    Recieve_buffer.getptr += 2;/*discard*/
                                    break;

                    case PERIPH_LEARN:
                                    Recieve_buffer.getptr += 1;/*discard*/
                                    break;

                    case SEEK_OFFSET:
                    case JOY_MOVE:
                    case PERIPH_RECALL:
                                    Recieve_buffer.getptr += 3;/*discard*/
                                    break;

                    default:
                                    Aba_error = TRUE;
                                    break;
            }
    }
}
```

LINC Key Numbers

```
#define KEY_SAVE       0
#define KEY_RCLL       1
#define KEY_EFFECT     2
#define KEY_MODIFY     10
#define KEY_COPY       11
#define KEY_INST       12
#define KEY_DELETE     13
#define KEY_GOTO       17
#define KEY_STRT       18
#define KEY_TO         19
#define KEY_END        20
#define KEY_ALL        21
#define KEY_LASTKF     26
#define KEY_THIS       27
#define KEY_NEXTKF     28
#define KEY_NORM       53
#define KEY_CNTR       54
#define KEY_CLR        55
#define UKEY_SK0       83        /*these are the softkeys*/
#define UKEY_SK 1      84
#define UKEY_SK2       85
#define UKEY_SK3       86
#define UKEY_SK4       87
#define UKEY_SK5       88
#define UKEY_SK6       89
#define UKEY_SK7       90
#define UKEY_SK8       91
#define UKEY_SK9       92
#define UKEY_SK10      93
#define UKEY_SK11      94
#define UKEY_SK12      95
#define UKEY_SK13      96
#define UKEY_SK14      97
#define UKEY_SK15      98
#define UKEY_SK16      99
#define UKEY_SK17      100
#define UKEY_SK18      101
#define UKEY_SK19      102
#define UKEY_SK20      103
#define UKEY_SK21      104
#define UKEY_SK22      105
 #define UKEY_SK23     106
```