THE BLIT DOT MAPPED DISPLAY TERMINAL
MANUFACTURING CODE: RD BLIT 1
PRODUCT CODE: RDAMMD

TABLE OF CONTENTS

BASIC FUNCTION	2
GENERAL TECHNICAL DATA	2
GENERAL OPERATION PROCEDURE	2
ASSOCIATED DOCUMENTS	3
COMPONENTS	3
DETAILED DESCRIPTION	4
ADJUSTMENTS	6
INSTALLATION INFORMATION	6
DISASSEMBLY-ASSEMBLY INFORMATION	7
SERVICING INSTRUCTIONS	9

BASIC FUNCTION

Blit is an alphanumeric and graphic dot mapped display terminal. The terminal can be used for desk-top design work when connected to a UNIX computer system which downloads the application program. Blit terminal applications consists of software run on Unix and software that is downloaded and run on the Blit terminal. C is the principal software development language.

GENERAL TECHNICAL DATA

The Blit terminal has a fifteen inch CRT display monitor that displays an 800X x 1024Y dot matrix at one hundred dots per inch. The terminal has an operator accessible ac switch and brightness control. The detached keyboard has 83 keys, eight lights, and an alarm, and interfaces to the display by a 300 baud TTL level serial input output port. A mouse is used for graphics and text manipulation by sending the display the displacement of the mouse held in the palm of the hand, and the selection of one of three mouse buttons. The mouse interfaces with the display by a nine wire TTL level input port.

See the attached pictures of the Blit terminal showing the display, keyboard and mouse. The size and weight are as follows: display 18"h x 13"w x 17"d and 59 pounds, keyboard 2.7"h x 19.7"w x 8"d and 5 pounds, mouse 1.8"h x 2.9"w x 3.5"d and 9 ounces.

The Blit communicates with a UNIX computer by means of an EIA RS232 interface. The baud for this interface is switch selectable on the circuit board for 300, 1200, 9600, or 19200 baud.

Circuit board strap options are provided for increasing the ROM capacity from 12k x 16 to 24k x 16 bits.

The blit operating requirements are as follows: 120 volts 60 Hz ac at 1.3 amperes nominal (110 watts at 0.524 pf), ambient temperature 10 C to 40 C.

GENERAL OPERATION PROCEDURE

All terminals come selected for 1200 baud rs232 operation. If you desire a different baud rate, open the back of the display and make the proper switch selection per the Detailed Description section of this document. Connect your terminal and dataphone with an EIA cable (not provided with the terminal). Call your computer, sign on and select the Blit application program you want downloaded. After downloading has completed the terminal is ready for operation. If you need application information or your computer needs the Blit software, see your account representative.

ASSOCIATED DOCUMENTS

"Blit Owner Information"

"Blit Hardware Description"

"A Programming Manual for Jeros"

"Overlapping Bitmap Graphics"

The circuit board schematic and assembly drawing.

Pictures of the Blit terminal.

Keyboard layout and code table.

COMPONENTS

The Blit consists of three electronic components: the display, the keyboard, and the mouse. The display is housed in a metal cabinet. The keyboard is housed in a structure foam cabinet with a metalized coating. And the mouse is housed in a plastic enclosure. All of the components are designed for use on a desk or table.

The Blit terminal components and their part numbers are as follows:

Component	Part number
AC cord, filter and cables	E196633
Cabinet	E196634
Circuit board	E196635
EIA cable (internal)	E196636
Keyboard	E196637
Keyboard cable (internal)	E196638
Monitor	E196639
Mouse	E196640
Mouse cable (internal)	E196641
Power supply and cables	E196642
Video cable (internal)	E196643

DETAILED DESCRIPTION

The Blit circuit board provides several options by means of switches and straps. On the back of the keyboard is a boot button that causes the dataphone to hangup when dip switch contact 2 is closed and causes the processor to reset when dip switch contact 1 is closed. The keyboard has a 300 baud serial interface. The rs232 serial line baud can be selected by closing dip switch contact 3,4,5 or 6 for 19200,9600,1200 or 300 baud respectively. Only one of the four switches should be closed. There are six 4k x 8 electrically programmable read only memories (EPROM) for program storage arranged as 12k x 16. The program storage can be increased to 24k x 16 by using 8k x 8 EPROM and by changing straps S3 and S5 to S4 and S6 respectively. The EPROM access time can be selected for 200 ns or 350 ns chips by moving strap S1 to S2. When removing or inserting EPROMs do not push or pull on the circuit board. The terminal is shipped optioned for resetting the processor with the boot button, 1200 baud and 12K x 16, 200 nsec EPROMs.

The circuit board logic controls the display monitor, the keyboard, the mouse, and the rs232 link to the host computer. The logic has two control circuits that share a 64k x 32 bit random access memory (RAM). The RAM is a two port memory with 256 k bytes used for storing data, programs and the screen image of the 800X x 1024Y x 1Z dot mapped display. Half the effective memory bandwidth is occupied with refreshing the display. The first control circuit has a MC68000 processor for executing programs that manage the RAM, handle the MC6850 communication interfaces for the input output data from the keyboard and rs232 line, and handles the input data from the mouse. The second control circuit reads the RAM and serializes the bits into a 32 MHZ video signal. Horizontal and vertical synchronizing signals cause the display monitor to sweep out an 800X x 1024Y matrix.

The circuit board power supply is a 50 watt switching power supply with +5 VDC @ 6A, +12 VDC @ 1A and -12 VDC @ 1A.

The mouse is an input device for graphics and text manipulation. The principle of the mouse is to signal it's displacement, speed and selection. Selection is signaled by three keys on the mouse. Displacement and speed are signaled by two TTL signals for each X and Y axis. The rate of the signals determine speed and the phase determines the direction. The signals are generated by precision optical wheels with phototransistors.

The 15" CRT display monitor will display a full typewritten page with either bright characters on a dark background or dark characters on a light background. The display area is 8" wide by 10.24" high. The horizontal rate is 32 kHz, the vertical rate is 60 Hz, and the video rate is 32 MHz. All of the signals to the

monitor are TTL levels. A P39 green phosphor provides long persistence for a 30 HZ interlaced refresh rate. The monitor has a special faceplate for anti-glare and contrast enhancement and an operator accessible brightness control. The operating temperature range is from 10 C to 40 C (50 F to 104 F).

The keyboard has 83 keys with a popular key layout. The keyboard is detached from the display and communicates with the display by ASCII coded TTL serial signals. The signal to the keyboard is used for actuating the eight LED indicators and an audible clicker. See the attached keyboard layout and ASCII code table.

The Blit display has 24K bytes of firmware for: downloading programs, basic terminal operations, bit manipulation operations for modifying rectangular areas of dots (bitmaps), and the routines for maintaining the bitmaps and the display windows (layers). Each process, even if hidden, has its own totally logically contiguous memory space. Layer consists of sets of bitmaps.

The Blit is a bitmap display used for interactive graphics using overlapping rectangles each containing a working environment, much like sheets of paper on a desk. Suitably programmed, the Blit display can become several displays at once. Different areas on the screen can be emulating printers, graphics terminals, drafting tables and even games.

Software to manipulate these overlapping areas , or layers, has been implemented on the BLIT terminal. A program drawing in a layer is fully isolated from other programs drawing on the same screen. even if one layer overlaps the other. Each program manipulates its layer just as it would manipulate the full screen if it were the only program running in the terminal.

The software to support asynchronous graphics in multiple, overlapping bitmap layers use several bitmap operations. A bitmap is a rectangular image stored off-screen. A routine bitblt copies images to and from the screen. The on and off screen memory is contiguous with the last word in a scan line followed by the first word of the next scan line.

The layer software isolates a program from other programs and guarantees that the image is always correct regardless of the layers on the screen. A partially obscured layer has an obscured rectangle list. To simplify rearranging layers all necessary subdivision is done when a layer is first made.

ADJUSTMENTS

The switching power supply used for the logic has a +5 VDC adjustment. Adjust the power supply with a screwdriver so the voltage on the circuit board is +5 VDC.

The display monitor has adjustments for DC focus, contrast, brightness, data horizontal centering, horizontal width, vertical centering, vertical height, B+ voltage, and dynamic focus. All of these adjustments are made at the factory and should not require adjustment in the field. If adjustment is required refer to the service manual for the HD Series display from Ball Electronic Display Division.

INSTALLATION INFORMATION

Unpack the display, the keyboard, and the mouse. Connect the ac line cord, the keyboard cable, and the mouse cable to their mating connectors on the back of the terminal. Plug the ac line cord into an ac outlet and toggle the power switch on. The first keyboard light should turn on and the cursor should appear in the home position after the monitor warms up. Connect the rs232 cable from your computer or dataset to the back of the terminal. Call your computer.

ASSEMBLY INFORMATION

Mount the ac filter in the pedestal chassis ground terminal down, with two screws, lock washers and nuts. Mount the ac switch from the outside by pushing retaining clip on the back of the switch. Push the spade lugs over the ac switch terminals. The ac terminals must be insulated to prevent a possible shocks hazard. Mount all D connectors in the pedestal wide part up. The keyboard D connector and the RS232 connector are fastened with four posts, lock washers and nuts. The mouse D connector is fastened with two screws, lock washers and nuts. Mount the brightness control with washer and nut provided. Push knob on the brightness control.

Mount the power supply, with the pc board insulator on the bottom, upside down and underneath the monitor mounting bracket with four screws. Connect ac to the power supply and adjust the dc output to five volts.

Assemble the pedestal, cabinet bottom cover and monitor mounting bracket together by routing the pedestal cables through the cabinet bottom cover hole and by routing all the cables between the monitor mounting bracket and the cabinet bottom cover on the left side of the cabinet. Turn the assemble upside down and secure together by inserting four screws through the pedestal and cabinet bottom into the monitor mounting bracket. Hold the cables to the cabinet bottom cover with the cable clamp.

Mount the monitor on the monitor mounting bracket with six screws. Do not tighten the screws. Plug the ac cable, the ac cable fused, and the brightness cable into the monitor.

Insert all of the parts in the board per the board assemble drawing and wave solder. Hand solder two 20 AWG bare wires with tubing on the solder side of the board as follows: from the dc voltage connector pins 1, 3, 5 to the board VCC plane and from the dc voltage connector pins 2, 4, 6 to the board GND plane.

With the circuit board on the left side of the monitor, plug in the diagnostic ROMs, the dc power supply cable, the mouse cable, the keyboard cable, and the RS232 cable. Plug the video cable (extra long for testing) into the monitor and the circuit board. Plug the mouse and the keyboard into the pedestal. Plug the ac line into the pedestal and ac outlet.

Turn power on and watch for the first light on the keyboard to light. If the light does not turn on, turn power off and determine the cause. After at least five minutes of operation, adjust the monitor for an 8" by 10.24" display per the Ball monitor manual. Adjust the picture using the following monitor adjustments: horizontal data center, horizontal width, vertical

height, horizontal center, and vertical center.

Turn set off. Remove all the circuit board cables. Secure the circuit board to the board mourting brackets using four screws and washers. Secure the pc board insulator to the board mounting brackets with two nuts to provides insulation between the circuit board and the CRT. Secure the board mounting brackets to the monitor using four screws and washers. Plug in the five circuit board cables. Make sure the short video cable is used. Remeasure the +5 VDC on the board.

Position the monitor as far back on the monitor mounting bracket as possible. Assemble the front, sides, and top of the cabinet with six nuts and install on the cabinet bottom cover. Push the monitor forward until it almost touchs the cabinet front. Remove the cabinet assembly and tighten the six monitor mounting screws. Reinstall the cabinet assembly and secure the back of the cabinet with eight chrome screws.

Run the diagnostic test in the continuous mode for one week of burn-in. Check every morning and repair if a failure occurred. After one week of reliable operation, remove the cabinet back and replace the diagnostic ROMs with terminal ROMs and insert the RS232 test cable in the pedestal. Run the terminal and repair if a failure occurs. Remove the RS232 test cable from the pedestal.

SERVICING INSTRUCTIONS

The diagnostic program is two EPROMs in rom sockets rom0 and rom1. The program runs several tests. Proceed to the next test by hitting the keyboard return button. When first turned on, the terminal should draw munching squares. If the image is a steady, slightly random pattern of almost horizontal bands, the processor is not running, so get the scope and analyzer out.

In the next two tests the screen should go completely green with 25 vertical black lines. All the black dots forming these lines visible at once are in the same RAM chip (1 of 32). The program will step through 32 times and the lines will slowly march from left to right across the screen. If any of the lines flicker, that RAM chip is slow and must be replaced. By counting 31 to 0 as the lines march across, you'll be able to determine which chip is bad. Pressing the Return key will cause the same thing for byte writes.

The next two test check out the keyboard. In these tests the keyboard lights will sequence and the display will copy a line of text from the keyboard.

The next two tests check out the mouse. The 'A' should track the mouse position, somewhat jumpily as you move it. Try pushing each of the three mouse buttons, individually. When a button is depressed, you will see which button you have depressed shown in the upper left hand corner of the screen. To go to the next test type another return. The 'B' on the screen is again tracking the mouse, this time using video interrupts. The tracking will be smoother than with the 'A'.

Typing another return will cause the display to continuously draw the bounce demonstration.

Typing another return will cause munching squares to repeat.

MOUSE CABLE

I/O CONN.	9 PIN SUB-D CONNECTOR FOR FLAT CABLE
CABLE	10 WIRE 28 AWG FLAT CABLE, 31"
BOARD CONN.	10 PIN FLAT CABLE CONNECTOR (J07)

	I/O CONN.	BOARD CONN. (J07)
+5V (VCC)	1	1
Y2 (AX1)	2	3
Y1 (AX0)	3	5
X2 (AY0)	4	7
X1 (AY1)	5	9
GND	6	2
K2- (BUT1)	7	4
K3- (BUT0)	8	6
K1- (BUT2)	9	8
KEY		10

RS232 CABLE

I/O CONN.	25 PIN SUB-D CONNECTOR FOR FLAT CABLE
CABLE	26 WIRE 28 AWG FLAT CABLE, 30"
BOARD CONN.	26 PIN FLAT CABLE CONNECTOR (J03)

		I/O CONN.	BOARD CONN. (J03)
PROTECT GND		1	1
TRANS DATA	(=TXD)	2	3
REC DATA	(=RXD)	3	5
DATA TERM. RDY.	(DTR)	20	14
CARRIER DET.	(DSR)	8	15
GND		7	13

KEYBOARD CABLE (EXTERNAL)

I/O CONN.	9 PIN SUB-D CONNECTOR
CABLE	8 WIRE 24 & 18 AWG CABLE, 4.5 FEET
KBD. CONN.	6 PIN SERIAL PORT CONNECTOR
BOOT BUTTON	3 TERMINAL SWITCH

	I/O CONN.	KBD. CONN.	BOOT BUTTON
+5V (VCC)	1	1	
GND	2	3	
NOT USED	3		
PANEL-	4		NO
GND	5		CEN
OUTPUT (RXDK)	6	2	
PROTECT GND	7	4	
INPUT (TXDK)	8	6	
PANEL	9		NC

KEYBOARD CABLE (INTERNAL)

I/O CONN.	9 PIN SUB-D CONNECTOR FOR FLAT CABLE
CABLE	10 WIRE 28 AWG FLAT CABLE, 27"
BOARD CONN.	10 PIN FLAT CABLE CONNECTOR (JO5)

	I/O CONN.	BOARD CONN. (JO5)
+5V (VCC)	1	1
GND	2	3
PANEL-	4	7
GND	5	9
OUTPUT (RXDK)	6	2
PROTECT GND	7	4
INPUT (TXDK)	8	6
PANEL	9	8
KEY		5

AC CABLE

FILTER	3 PIN AC LINE FILTER
CABLE	3 WIRE BLK,WHT,GRN 18 AWG, 19",24" BLK
SWITCH	2 PIN AC SWITCH (BLK WIRE ONLY)
CRT CONN.	4 PIN AC CONNECTOR TO MONITOR (P2)

		FILTER	SWITCH	CRT CONN.
AC LINE	BLK	L	IN OUT	4
AC NEUTRAL	WHT	N		1
PROTECT GND	GRN	CASE		2

AC CABLE FUSED

CABLE	2 WIRE BLK,WHT 24 AWG, 20"
DC POWER SUPPLY	2 PIN AC IN OF DC POWER SUPPLY
CRT P1 CONN.	4 PIN AC CONNECTOR ON MONITOR (P1 IS SUPPLIED WITH MONITOR)

		DC POWER	CRT P1 CONN.
AC LINE	BLK	AC IN 1	2 (WITH WHT/BLU WIRE)
AC NEUTRAL	WHT	AC IN 2	3 (WITH BLU WIRE)

BRIGHTNESS CABLE

CABLE	3 WIRES 24 AWG, 27"
P107 CONN.	4 PIN CONNECTOR TO MONITOR
CONTROL	3 PIN 60K POTENTIOMETER

	P107 CONN.	CONTROL
CW END OF CTL.	4	3
CENTER OF CTL.	3	2
CCW SIDE OF CTL.	1	1

DC POWER SUPPLY CABLE

CABLE	8 WIRE 22 AWG, 24"
DC POWER	5 PIN DC OUT OF DC POWER SUPPLY
DC BOARD CONN.	10 PIN BERG (J11)

	DC POWER	DC BOARD CONN. (J11)
+5VDC (VCC)	1	1
+5VDC (VCC)	1	3
+5VDC (VCC)	1	5
GND	2	2
GND	2	4
GND	3	6
+12VDC	4	10
-12VDC	5	9
KEY		8

VIDEO CABLE

CABLE	3 TWISTED PAIR WIRES, 5.5"
VIDEO BOARD CONN.	10 PIN BERG (J09)
VIDEO CRT CONN.	6 PIN VIDEO CONNECTOR TO MONITOR (J3)

	VIDEO BOARD CONN.(J09)	VIDEO CRT CONN.
HORIZONTAL SYNC	5	1
HORIZONTAL GND	6	3
VERTICAL SYNC	7	2
VERTICAL GND	8	3
VIDEO	1	4
VIDEO GND	2	6
KEY	4	

The following list locates the chips in the circuit board.

NUMBER	LOCATION(PIN 1)			PINS	WIDTH	CHIP	NAME
001	B	D	023	016	02	RAM64K	ram0
002	C	C	022	016	02	RAM64K	ram8
003	B	D	043	016	02	RAM64K	ram16
004	C	C	043	016	02	RAM64K	ram24
005	E	C	001	020	02	LS244	inbuf
006	D	B	012	016	02	DL240	delay
007	B	F	023	016	02	RAM64K	ram2
008	C	E	022	016	02	RAM64K	ram10
009	B	F	043	016	02	RAM64K	ram18
010	C	E	043	016	02	RAM64K	ram26
011	B	H	032	020	02	LS374	d_out0
012	F	H	044	014	02	LS02	nor0
013	B	H	023	016	02	RAM64K	ram4
014	C	G	022	016	02	RAM64K	ram12
015	B	H	043	016	02	RAM64K	ram20
016	C	G	043	016	02	RAM64K	ram28
017	D	F	032	020	02	LS374	d_out1
018	B	B	009	014	02	74C14	cinv
019	C	A	022	016	02	RAM64K	ram6
020	C	J	022	016	02	RAM64K	ram14
021	C	A	043	016	02	RAM64K	ram22
022	C	J	043	016	02	RAM64K	ram30
023	C	A	032	020	02	LS374	d_out2
024	E	C	012	016	02	B18983	r_res
025	C	J	012	016	02	B18983	w_res
026	C	C	032	020	02	2966	a_buf0
027	E	A	032	020	02	2966	a_buf1
028	D	H	032	020	02	LS374	d_out3
029	D	E	032	020	02	PAL.L8	mem/lce
030	D	B	022	016	02	RAM64K	ram1
031	E	A	022	016	02	RAM64K	ram9
032	D	B	043	016	02	RAM64K	ram17
033	E	A	043	016	02	RAM64K	ram25
034	C	E	032	020	02	LS244	d_buf0
035	A	E	026	016	02	LS166	s_sr0
036	D	D	022	016	02	RAM64K	ram3
037	E	C	022	016	02	RAM64K	ram11
038	D	D	043	016	02	RAM64K	ram19
039	E	C	043	016	02	RAM64K	ram27
040	C	G	032	020	02	LS244	d_buf1
041	A	G	026	016	02	LS166	s_sr1
042	D	F	022	016	02	RAM64K	ram5
043	E	E	022	016	02	RAM64K	ram13
044	D	F	043	016	02	RAM64K	ram21
045	E	E	043	016	02	RAM64K	ram29
046	C	J	032	020	02	LS244	d_buf2
047	A	J	026	016	02	LS166	s_sr2
048	D	H	022	016	02	RAM64K	ram7

049	E	G	022	016	02	RAM64K	ram15
050	D	H	043	016	02	RAM64K	ram23
051	E	G	043	016	02	RAM64K	ram31
052	D	D	032	020	02	LS244	d_buf3
053	B	B	026	016	02	LS166	s_sr3
054	A	G	009	016	02	K1100A	clock
055	A	G	017	016	02	S195	f_sr
056	A	E	017	016	02	S163	div0
057	A	G	001	014	02	S00	nand0
058	A	J	009	014	02	LS10	3nand0
059	B	D	014	016	02	LS163	rs0
060	B	F	014	016	02	LS163	rs1
061	B	H	014	016	02	LS163	rs2
062	E	J	001	016	02	DIP.RC	rc
063	B	B	001	014	02	75188	drv
064	A	J	001	014	02	75189	rcv
065	F	D	001	028	03	2764	rom2
066	F	D	015	028	03	2764	rom3
067	F	D	029	028	03	2764	rom5
068	F	G	001	028	03	2764	rom0
069	F	G	015	028	03	2764	rom1
070	F	G	029	028	03	2764	rom4
071	B	D	001	024	03	MC6850	kbd_acia
072	B	G	001	024	03	MC6850	acia
073	E	C	032	020	02	PAL.L8	tim/lde
074	E	G	032	020	02	LS373	addr2
075	E	E	001	020	02	LS373	addr0
076	E	E	012	016	02	818981	pu_4.7k
077	A	E	044	014	02	LS164	pr_sr
078	F	D	044	014	02	LS260	5nor0
079	E	J	010	064	04	68000	cpu
080	A	E	001	020	02	S244	cbuf
081	A	J	044	014	02	LS74	ff1
082	C	A	001	020	02	LS374	a_st0
083	E	G	012	016	02	LS148	ipl
084	E	E	032	020	02	PAL.L8	dec/lde
085	B	F	032	020	02	LS374	cked
086	E	G	001	020	02	LS373	addr1
087	B	D	032	020	02	S472	h_rom
088	B	B	017	016	02	LS163	h_ctr1
089	D	F	001	020	02	LS2569	x1
090	D	D	012	016	02	LS153	a_mux0
091	C	A	012	016	02	LS163	a_ctr0
092	A	E	035	016	02	LS163	v_ctr0
093	A	G	044	014	02	LS74	ff0
094	F	F	044	014	02	LS260	5nor1
095	C	J	001	020	02	LS2569	y1
096	E	A	012	016	02	LS153	a_mux3
097	D	H	012	016	02	LS153	a_mux2
098	D	F	012	016	02	LS153	a_mux1
099	B	B	035	016	02	82S131	v_rom

100	A	J	017	016	02	LS163	h_ctr0
101	D	H	001	020	02	LS2569	x2
102	A	G	035	016	02	LS163	v_ctr1
103	C	C	012	016	02	LS163	a_ctr1
104	E	J	043	016	02	LS138	io_dec
105	A	E	001	014	02	LS04	inv0
106	A	E	009	014	02	S86	xor0
107	D	B	001	020	02	LS2569	y2
108	F	B	043	016	02	LS139	io_sub
109	C	G	012	016	02	LS163	a_ctr3
110	C	E	012	016	02	LS163	a_ctr2
111	A	J	035	016	02	LS163	v_ctr2
112	B	B	044	014	02	LS74	ff2
113	C	G	001	020	02	PAL.R4	y0
114	D	D	001	020	02	PAL.R4	x0
115	C	E	001	020	02	LS244	mbuf
116	C	C	001	020	02	LS374	a_st1
117	F	B	001	016	02	DIP.SW	dpsw
201	A	E	043	002	02	STRAP	
202	A	G	043	002	02	STRAP	
203	E	F	020	002	02	STRAP	
204	E	H	020	002	02	STRAP	
205	E	F	030	002	02	STRAP	
206	E	H	030	002	02	STRAP	
J03	BJ	01	001	026		RS232	
J05	CJ	01	001	026		KEYBD	
J07	DJ	01	001	026		MICKEY	
J09	EJ	01	001	026		VIDEO	
J11	FJ	01	001	026		DCVOLT	

The following list contains the signals on the circuit board.

SIGNAL	CHIP	CHIP	CHIP	CHIP	CHIP
-12V	63-1	J11-1			
+12V	63-14	J11-14			
$0-0	12-8	201-2			
$0-0A	77-5	201-1			
$0-0B	77-10	202-1			
$0-1	12-10	79-10			
$0-2	62-15	79-17			
$0-3	18-5	62-14	62-13		
$0-4	94-4	18-6			
$0-5	112-9	57-5			
$1-0	57-10	57-9	60-7	59-15	61-10
$1-1	57-8	59-9			
$1-2	60-15	61-7			
$2-0	58-1	78-5			
$2-1	58-2	78-6			
$2-2	58-13	94-5			
$2-3	6-6	62-7	62-8		
$2-4	80-5	24-2	24-1	24-3	24-4
$3-0	106-1	55-12			
$4-0	100-15	88-10			
$4-1	92-15	102-10			
$4-2	91-15	103-10			
$4-3	103-15	110-10			
$4-4	110-15	109-10			
$4-5	102-15	111-10			
$5-0	106-6	106-9			
$5-1	18-8	106-12			
$5-2	18-9	62-12	62-11		
$5-3	62-5	106-8	106-13		
1200	117-5	61-14			
16M	77-8	56-14	57-1		
19200	117-3	60-14			
2M	56-11	58-4	87-1		
300	71-3	71-4	61-12	117-6	
32M	54-8	55-10	56-2		
4M	58-3	56-12	57-13	57-12	
4M-	103-2	109-2	110-2	91-2	92-2
	102-2	111-2	100-2	88-2	85-11
	57-11				
8M	59-2	60-2	41-7	41-6	47-7
	47-6	35-6	35-7	61-2	57-2
	53-6	53-7	56-13	79-15	
9600	117-4	60-13			
AS-	79-6	74-11	75-11	80-2	86-11
CAS-	80-15	29-15			
CAS0-	1-15	7-15	13-15	19-15	30-15
	36-15	42-15	48-15	24-13	
CAS1-	2-15	8-15	14-15	20-15	31-15

	37-15	43-15	49-15	24-14	
CAS2-	3-15	9-15	15-15	21-15	32-15
	38-15	44-15	50-15	24-15	
CAS3-	4-15	10-15	16-15	22-15	33-15
	39-15	45-15	51-15	24-16	
DL-	29-18	73-1	6-1		
DREQ	85-4	87-7			
DSR-	72-23	64-6			
E	80-13	79-20			
HSYN	85-8	87-14			
RAM	84-19	73-7			
RAM/WE-	80-17	29-19			
RAS0-	1-4	7-4	13-4	19-4	30-4
	36-4	42-4	48-4	24-9	
RAS1-	2-4	8-4	14-4	20-4	31-4
	37-4	43-4	49-4	24-10	
RAS2-	3-4	9-4	15-4	21-4	32-4
	38-4	44-4	50-4	24-11	
RAS3-	4-4	10-4	16-4	22-4	33-4
	39-4	45-4	51-4	24-12	
ROM	77-9	84-18			
RXD	64-3	72-2			
SAMPL	85-7	87-13			
TXD	63-4	72-6			
TCAS-	6-3	29-6			
TPCHG	6-14	29-8			
TRAS-	29-5	73-2	6-15		
TREADY-	6-12	29-9			
TROWEN-	6-2	29-7			
VID+	J09-1	80-16			
VMA-	80-8	79-19			
VSYN	99-12	85-3			
WE-	80-11	79-9			
A/EN	103-7	109-7	110-7	91-10	91-7
	87-6				
A/LD-	103-9	109-9	110-9	91-9	87-11
ABUS01	108-14	108-2	68-10	65-10	66-10
	69-10	75-5	70-10	67-10	71-11
	72-11	73-4			
ABUS02	65-9	68-9	69-9	66-9	67-9
	70-9	75-6	90-6	108-3	108-13
ABUS03	65-8	68-8	69-8	66-8	67-8
	70-8	75-9	78-1	90-10	104-1
ABUS04	65-7	68-7	69-7	66-7	67-7
	70-7	75-12	78-2	98-6	104-2
ABUS05	65-6	68-6	69-6	66-6	67-6
	70-6	75-15	78-3	104-3	98-10
ABUS06	65-5	68-5	69-5	66-5	67-5
	70-5	75-16	78-12	97-6	
ABUS07	65-4	68-4	69-4	66-4	67-4
	70-4	75-19	78-13	97-10	

ABUS08	65-3	68-3	69-3	66-3	67-3
	70-3	86-2	78-4	96-6	
ABUS09	65-25	68-25	69-25	66-25	67-25
	70-25	86-5	78-8	96-10	
ABUS10	65-24	68-24	69-24	66-24	67-24
	70-24	90-5	78-9	86-6	
ABUS11	65-21	68-21	69-21	66-21	67-21
	70-21	86-9	78-10	90-11	
ABUS12	65-23	68-23	69-23	66-23	67-23
	70-23	78-11	98-5	86-12	
ABUS13	94-1	98-11	86-15	203-1	65-2
	66-2	67-2	68-2	69-2	70-2
ABUS1314	203-2	84-6			
ABUS14	94-2	97-5	86-16	204-1	205-1
ABUS1415	205-2	84-9			
ABUS15	94-3	97-11	86-19	206-1	
ABUS16	74-2	96-5	94-12		
ABUS17	74-5	84-2	96-11	94-13	
ABUS18	74-6	84-3			
=AS-	80-18	84-7			
AX0	J07-5	114-2			
AX1	J07-3	114-3			
AY0	J07-7	113-2			
AY1	J07-9	113-3			
BERR-	57-6	79-22			
BLANK	93-5	87-19			
BOOT-	58-12	84-1			
BOOTME	94-9	105-10			
BOOTME-	105-11	76-11	117-16		
BUT0	J07-6	105-1			
BUT1	J07-4	105-3			
BUT2	J07-8	105-5			
CLX	89-2	101-2	114-1	114-13	
CLY	95-2	107-2	113-1	113-13	
CPU/A01	75-4	79-29			
CPU/A02	75-7	79-30			
CPU/A03	75-8	79-31			
CPU/A04	75-13	79-32			
CPU/A05	75-14	79-33			
CPU/A06	75-17	79-34			
CPU/A07	75-18	79-35			
CPU/A08	86-3	79-36			
CPU/A09	86-4	79-37			
CPU/A10	86-7	79-38			
CPU/A11	86-8	79-39			
CPU/A12	86-13	79-40			
CPU/A13	86-14	79-41			
CPU/A14	86-17	79-42			
CPU/A15	86-18	79-43			
CPU/A16	74-3	79-44			
CPU/A17	74-4	79-45			

CPU/A18	74-7	79-46			
CPU/A19	74-8	79-47			
CPU/A20	74-13	79-48			
CPU/A21	74-14	79-50			
CPU/A22	74-17	79-51			
CPU/A23	74-18	79-52			
CX1	89-19	101-7			
CY1	95-19	107-7			
D/REQ	93-11	85-5			
DANGER	57-4	84-13			
DBUS00	46-18	34-18	3-2	1-2	65-11
	68-11	70-11	72-22	82-3	79-5
	113-17	114-17	115-18	81-12	
DBUS01	82-4	72-21	70-12	46-16	34-16
	32-2	30-2	65-12	68-12	79-4
	113-16	114-16	112-12	115-16	
DBUS02	65-13	68-13	79-3	113-15	114-15
	115-14	82-7	72-20	70-13	46-14
	34-14	9-2	7-2		
DBUS03	34-12	46-12	38-2	36-2	65-15
	68-15	70-15	72-19	82-8	79-2
	113-14	114-14			
DBUS04	34-9	46-9	72-18	70-16	82-13
	89-16	95-16	79-1	68-16	65-16
	13-2	15-2			
DBUS05	34-7	46-7	44-2	42-2	65-17
	68-17	70-17	72-17	82-14	79-64
	89-15	95-15			
DBUS06	46-5	34-5	21-2	19-2	65-18
	68-18	70-18	72-16	82-17	79-63
	89-14	95-14			
DBUS07	82-18	70-19	72-15	46-3	34-3
	50-2	48-2	65-19	68-19	79-62
	89-13	95-13			
DBUS08	116-3	107-16	101-16	79-61	69-11
	66-11	67-11	71-22	52-18	40-18
	5-18	4-2	2-2		
DBUS09	116-4	107-15	101-15	79-60	69-12
	66-12	67-12	71-21	52-16	40-16
	5-16	33-2	31-2		
DBUS10	116-7	107-14	101-14	79-59	69-13
	66-13	67-13	71-20	52-14	40-14
	5-14	10-2	8-2		
DBUS11	116-8	107-13	101-13	79-58	69-15
	66-15	67-15	71-19	52-12	5-12
	40-12	39-2	37-2		
DBUS12	14-2	16-2	5-9	40-9	52-9
	71-18	67-16	66-16	69-16	79-57
	115-9	116-13			
DBUS13	5-7	40-7	52-7	45-2	43-2
	66-17	67-17	71-17	69-17	79-56

	115-7	116-14			
DBUS14	20-2	22-2	5-5	40-5	52-5
	71-16	67-18	66-18	69-18	79-55
	115-5	116-17			
DBUS15	116-18	115-3	79-54	69-19	66-19
	49-2	51-2	67-19	71-15	52-3
	40-3	5-3			
DCYCLE	90-2	96-2	97-2	98-2	29-16
	73-3				
DISP00	91-14	90-4			
DISP01	91-13	90-12			
DISP02	91-12	98-4			
DISP03	91-11	98-12			
DISP04	103-14	97-4			
DISP05	103-13	97-12			
DISP06	103-12	96-4			
DISP07	103-11	96-12			
DISP08	110-14	90-3			
DISP09	110-13	90-13			
DISP10	110-12	98-3			
DISP11	110-11	98-13			
DISP12	109-14	97-3			
DISP13	109-13	97-13			
DISP14	109-12	96-3			
DISP15	109-11	96-13			
D000	1-14	34-2	11-3		
D001	30-14	34-4	17-3		
D002	7-14	23-3	34-6		
D003	36-14	28-3	34-8		
D004	13-14	34-11	11-4		
D005	42-14	34-13	17-4		
D006	19-14	34-15	23-4		
D007	48-14	28-4	34-17		
D008	2-14	40-2	11-7		
D009	31-14	40-4	17-7		
D010	8-14	40-6	23-7		
D011	37-14	28-7	40-8		
D012	14-14	40-11	11-8		
D013	43-14	40-13	17-8		
D014	20-14	40-15	23-8		
D015	49-14	28-8	40-17		
D016	3-14	46-2	11-13		
D017	32-14	46-4	17-13		
D018	9-14	23-13	46-6		
D019	38-14	28-13	46-8		
D020	15-14	46-11	11-14		
D021	44-14	46-13	17-14		
D022	21-14	23-14	46-15		
D023	50-14	28-14	46-17		
D024	4-14	52-2	11-17		
D025	33-14	52-4	17-17		

D026	10-14	23-17	52-6	
D027	39-14	28-17	52-8	
D028	16-14	11-18	52-11	
D029	45-14	52-13	17-18	
D030	22-14	23-18	52-15	
D031	51-14	52-17	28-18	
DQ	29-4	12-4	12-3	
DREADY-	93-13	29-12		
DREQ-	11-11	17-11	23-11	28-11
	12-5	93-8		
DSR	J03-15	64-4		
DTR	J03-14	63-3		
=E	104-6	80-7	72-14	71-14
EX	89-12	101-12	114-18	
EY	113-18	107-12	95-12	
F/LD-	57-3	55-9		
FC7-	58-8	84-8		
H/CNT1	100-13	87-2		
H/CNT2	100-12	87-3		
H/CNT3	100-11	87-4		
H/CNT4	88-14	87-5		
H/CNT5	88-13	87-16		
H/CNT6	88-12	87-17		
H/CNT7	88-11	87-18		
H/LD-	87-9	88-9	100-9	
H/SYNC	J09-5	85-9		
HANGUP	63-2	105-8		
HANGUP-	105-9	76-12	117-15	
INT1	76-1	83-11	81-6	
INT2	76-2	83-12	71-7	
INT3	83-13	76-3		
INT4	83-1	76-4	112-6	
INT5	76-5	83-2	72-7	
INT6	83-3	76-6		
INT7	83-4	76-7		
IO0-	108-1	115-19	104-15	
IO00-	89-17	101-17	108-4	114-11
IO02-	95-17	107-17	113-11	108-5
IO1-	104-14	72-9		
IO2-	115-1	112-1	104-13	
IO3-	82-11	104-12	116-11	
IO4-	81-11	104-11		
IO5-	104-10	108-15		
IO50-	112-11	108-12		
IO52-	108-11	5-19	5-1	
IO6-	104-9	71-9		
IO7-	81-1	104-7		
LDS-	79-8	84-4	73-5	
M/HIT	112-3	106-11		
MB0	105-2	115-2	106-4	
MB1	105-4	115-4	106-5	

MB2	105-6	115-6	106-10		
MQ	29-3	12-1	12-6		
MREADY	12-9	29-13			
MREQ-	84-17	12-2			
PANEL	J05-8	18-1	18-4		
PANEL-	J05-7	117-2	117-1	18-3	18-2
RAM/A00	19-5	13-5	7-5	1-5	2-5
	8-5	14-5	20-5	26-18	21-5
	15-5	9-5	3-5	4-5	10-5
	16-5	22-5			
RAM/A01	19-7	13-7	7-7	1-7	2-7
	8-7	14-7	20-7	26-16	21-7
	15-7	9-7	3-7	4-7	10-7
	16-7	22-7			
RAM/A02	19-6	13-6	7-6	1-6	2-6
	8-6	14-6	20-6	26-14	21-6
	15-6	9-6	3-6	4-6	10-6
	16-6	22-6			
RAM/A03	19-12	13-12	7-12	1-12	2-12
	8-12	14-12	20-12	26-12	21-12
	15-12	9-12	3-12	4-12	10-12
	16-12	22-12			
RAM/A04	19-11	13-11	7-11	1-11	2-11
	8-11	14-11	20-11	26-9	21-11
	15-11	9-11	3-11	4-11	10-11
	16-11	22-11			
RAM/A05	19-10	13-10	7-10	1-10	2-10
	8-10	14-10	20-10	26-7	21-10
	15-10	9-10	3-10	4-10	10-10
	16-10	22-10			
RAM/A06	19-13	13-13	7-13	1-13	2-13
	8-13	14-13	20-13	26-5	21-13
	15-13	9-13	3-13	4-13	10-13
	16-13	22-13			
RAM/A07	1-9	7-9	13-9	19-9	26-3
	20-9	14-9	8-9	2-9	3-9
	9-9	15-9	21-9	22-9	16-9
	10-9	4-9			
RAM/A08	48-5	42-5	36-5	30-5	31-5
	37-5	43-5	49-5	50-5	44-5
	38-5	32-5	27-18	33-5	39-5
	45-5	51-5			
RAM/A09	48-7	42-7	36-7	30-7	31-7
	37-7	43-7	49-7	50-7	44-7
	38-7	32-7	27-16	33-7	39-7
	45-7	51-7			
RAM/A10	48-6	42-6	36-6	30-6	31-6
	37-6	43-6	49-6	50-6	44-6
	38-6	32-6	27-14	33-6	39-6
	45-6	51-6			
RAM/A11	48-12	42-12	36-12	30-12	31-12

	37-12	43-12	49-12	50-12	44-12
	38-12	32-12	27-12	33-12	39-12
	45-12	51-12			
RAM/A12	48-11	42-11	36-11	30-11	31-11
	37-11	43-11	49-11	50-11	44-11
	38-11	32-11	27-9	33-11	39-11
	45-11	51-11			
RAM/A13	48-10	42-10	36-10	30-10	31-10
	37-10	43-10	49-10	50-10	44-10
	38-10	32-10	27-7	33-10	39-10
	45-10	51-10			
RAM/A14	48-13	42-13	36-13	30-13	31-13
	37-13	43-13	49-13	50-13	44-13
	38-13	32-13	27-5	33-13	39-13
	45-13	51-13			
RAM/A15	48-9	42-9	36-9	30-9	31-9
	37-9	43-9	49-9	27-3	32-9
	38-9	44-9	50-9	51-9	45-9
	39-9	33-9			
RAM/OE0-	34-1	34-19	73-15		
RAM/OE1-	40-1	40-19	73-14		
RAM/OE2-	46-1	46-19	73-13		
RAM/OE3-	52-1	52-19	73-12		
=RAM/WE-	25-1	25-2	25-3	25-4	25-5
	25-6	25-7	25-8	80-3	
=RAS0-	24-8	73-19			
=RAS1-	24-7	73-18			
=RAS2-	24-6	73-17			
=RAS3-	24-5	73-16			
RESET-	62-16	79-18			
ROM/EN0	68-22	69-22	84-16		
ROM/EN1	65-22	66-22	84-15		
ROM/EN2	67-22	70-22	84-14		
ROWEN	90-14	96-14	97-14	98-14	29-14
RSCK	72-3	72-4	117-14	117-13	117-12
	117-11				
RWE0-	25-9	1-3	2-3	3-3	4-3
RWE1-	30-3	25-10	31-3	32-3	33-3
RWE2-	25-11	7-3	8-3	9-3	10-3
RWE3-	25-12	36-3	37-3	38-3	39-3
RWE4-	25-13	13-3	14-3	15-3	16-3
RWE5-	25-14	42-3	43-3	44-3	45-3
RWE6-	25-15	19-3	20-3	21-3	22-3
RWE7-	25-16	48-3	49-3	50-3	51-3
=RXD	J03-5	64-1			
RXDK	J05-2	71-2			
S/LD-	35-15	47-15	41-15	53-15	58-6
SA00	91-3	82-2			
SA01	91-4	82-5			
SA02	91-5	82-6			
SA03	91-6	82-9			

SA04	103-3	82-12			
SA05	103-4	82-15			
SA06	103-5	82-16			
SA07	103-6	82-19			
SA08	110-3	116-2			
SA09	110-4	116-5			
SA10	110-5	116-6			
SA11	110-6	116-9			
SA12	109-3	116-12			
SA13	109-4	116-15			
SA14	109-5	116-16			
SA15	109-6	116-19			
SAMPLE	93-3	85-6			
SRLOAD	58-5	87-12			
=TXD	J03-3	63-6			
TXDK	J05-6	71-6			
UDS-	79-7	84-5	73-6		
UX	89-1	101-1	114-12		
UY	95-1	107-1	113-12		
V/BLANK	93-2	99-11			
V/CNT00	92-14	99-5			
V/CNT01	92-13	99-6			
V/CNT02	92-12	99-7			
V/CNT03	92-11	99-4			
V/CNT04	102-14	99-3			
V/CNT05	102-13	99-2			
V/CNT06	102-12	99-1			
V/CNT10	111-12	99-15			
V/EN	111-7	102-7	92-10	92-7	87-8
V/INV	35-1	41-1	47-1	53-1	81-9
	106-2				
V/LD-	92-9	99-10	102-9	111-9	
V/SYNC	J09-7	81-3	85-2		
VIDEO+	106-3	80-4			
=VMA-	80-12	104-4	104-5		
VPA-	84-12	79-21			
=WE-	71-13	72-13	84-11	80-9	73-8
	29-11				
XA0	83-9	79-25			
XA1	83-7	79-24			
XA2	83-6	79-23			
XB0	58-9	79-28			
XB1	58-10	79-27			
XB2	58-11	79-26			
XC0	26-2	27-2	90-7		
XC1	26-4	27-4	90-9		
XC2	26-6	27-6	98-7		
XC3	26-8	27-8	98-9		
XC4	26-11	27-11	97-7		
XC5	26-13	27-13	97-9		
XC6	96-7	27-15	26-15		

XC7	26-17	27-17	96-9	
XE00	35-2	11-2		
XE01	35-3	11-5		
XE02	35-4	11-6		
XE03	35-5	11-9		
XE04	35-10	11-12		
XE05	35-11	11-15		
XE06	35-12	11-16		
XE07	35-14	11-19		
XE08	41-2	17-2		
XE09	41-3	17-5		
XE10	41-4	17-6		
XE11	41-5	17-9		
XE12	41-10	17-12		
XE13	41-11	17-15		
XE14	41-12	17-16		
XE15	41-14	17-19		
XE16	47-2	23-2		
XE17	47-3	23-5		
XE18	47-4	23-6		
XE19	47-5	23-9		
XE20	47-10	23-12		
XE21	47-11	23-15		
XE22	47-12	23-16		
XE23	47-14	23-19		
XE24	53-2	28-2		
XE25	53-3	28-5		
XE26	53-4	28-6		
XE27	53-5	28-9		
XE28	53-10	28-12		
XE29	53-11	28-15		
XE30	53-12	28-16		
XE31	53-14	28-19		
XF0	55-4	35-13		
XF1	55-5	41-13		
XF2	55-6	47-13		
XF3	55-7	53-13		
ZAP-	112-13	94-6	62-1	62-2

Blit Owner Information

Bart Locanthi

Congratulations! You have chosen the finest in terminal technology. With a little care, your new Blit should provide years of service and enjoyment to you and your family. We want you to be able to make the best of your Blit. However, reading this little blurb can only get you started. **The Blit Programmer's Manual** is must reading for the serious Blit programmer. The various user programs (mpx, jim, jx, etc.) are covered by the copious on-line documentation.

Setting up your Blit

First, you need to find space. The monitor enclosure is large, the keyboard is wide, and the mouse needs a little area for running around. You will need a clear area about 30" square for your jerqstation. This area must be near an electrical outlet and an RS-232 terminal line.

The connectors in the back of the monitor enclosure are labeled and keyed, so there should be no confusion about plugging them in. In order to work, your Blit **must** be connected to power, the keyboard **must** be plugged in, and the RS232 port **must** be connected to a running computer. The only thing that doesn't need to be plugged in to get started is the mouse.

If when you turn on your Blit the screen shows a set of ragged horizontal stripes, it is not receiving the RS232 signal DCD at the connector. This can result from any of the following:

- you have a TDK connection without a magic TDK box
- your host computer is down
- your host computer does not issue DCD

Consult your computing services representative about these problems if they arise. Otherwise your Blit should power up successfully and display a rectangular cursor and possibly a login message.

If the login message is garbled, you will have to set the baud rate. This is done by removing the back of the monitor enclosure (with the power OFF!!) and setting the DIP switch on the logic board. The switch settings are shown below.

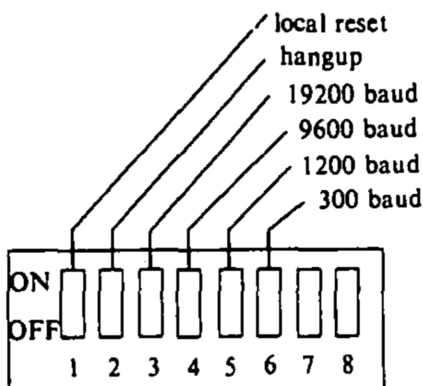


Figure 1. DIP switch settings

There is a small pushbutton behind the left side of the keyboard. When pressed, this "boot" switch will reset your terminal and/or hang up the phone, depending on the settings on the DIP switch. This is necessary when the terminal crashes or when the host-terminal communication gets

confused. Normally the boot switch will be set for **local reset** only. If you are afforded the luxury of a hard-wired line to your host you may wish to set it also for **hangup**. If both settings are selected, you can flick the boot switch to reset just the terminal or you can hold it down to hang up your host connection.

The Mouse

The standard pointing device for Blits is a round palm-sized object called a **mouse**. A mouse is an **incremental** pointing device. That is, it detects changes in position when you slide it across a surface. Because it detects incremental motion, you can pick it up and roll it several times to move a large distance. An area the size of a normal piece of paper is more than enough space for a mouse.

Those programs that make use of the mouse will display a cursor that tracks the mouse position. A text editor may display two cursors, one to indicate typing position in the text, the other tracking the mouse position. The cursor shape may be anything from an arrow to a little square or a meditating Buddha, depending on the needs of the program. Programs using the mouse will also make use of the three mouse pushbuttons to indicate selection or actions to be performed on the selected object. The mouse button functions, like the cursor shape, depend on the program.

What the user should remember about the mouse is that its pushbuttons are not like those of a keyboard. Pressing and releasing a mouse button are two distinct events, and many programs will make use of that distinction. **Menus** are an example of this. Pressing a menu button will cause a menu to appear. The menu selection is made, however, when the button is released. Between these two events the possible menu selection will track the cursor. Each program will have its own way of dealing with the mouse. Read the documentation for a program if you have any questions about how it uses the mouse.

Altered States

Your Blit is different from ordinary terminals in that the program running inside is loadable. This means that

- your Blit is more fun than a barrel full of 2621's.
- your Blit can crash.
- your Blit may not always know how to act like a terminal.
- your Blit may act like a terminal to a program on Unix that doesn't want to talk to a terminal.

The middle two cases are easy to deal with by re-booting your Blit. Dealing with the last case may demand hanging up from Unix. An example of this case is the troff proofing program **jc**. Before sending troff output to the screen, **jc** first loads a program to simulate some of the functions of a typesetter. If for some reason your Blit reverts to being a terminal without having been told to do so by **jc**, you have lost control and must hang up in order to get it back.

Upon being reset (this includes power-on) your Blit starts running a terminal program that makes it look like an ordinary terminal with a large screen. Setting TERM=blit will satisfy most Unix programs that care about what kind of terminal they are talking to. Two programs that are known to have problems with the default Blit terminal program are **vi** and **rogue**. A more compatible terminal program may be loaded by typing "68ld /usr/jerq/lib/viterm" to Unix. Once loaded, **vi** and **rogue** may be used with impunity **until** your Blit is again reset. Since most interesting user programs for the Blit reset the terminal upon exiting, you may wind up reloading viterm several times in the course of the day. Sorry.

The curious may enjoy looking in /usr/jerq/lib/*term and seeing what other, mostly dull, terminal programs there are to play with. Other interesting demonstration programs can be found in /usr/jerq/demo.

A final caveat. Loading a Blit program is currently possible **only** on a VAX, and **only** when your Blit is connected either by a direct hardwired or phone line or TDK (and not, for example, through *cu*).

Blit Hardware Description

Bart Locanthi

ABSTRACT

A walk-through of the low-level details of Blit hardware

Introduction

The Blit is a bitmap display terminal with a Motorola MC68000 microprocessor sharing 256K bytes of storage with a raster scanned display. The display measures 800 dots wide by 1024 lines high and is interlaced at 60 Hz. No hardware support is provided for managing the display aside from the logic that copies 100K bytes from the RAM array to the screen. Included in the terminal are a keyboard, an RS232 serial interface, and a "mouse" pointing device.

This paper is intended as a guide to the schematic diagrams of the Blit hardware. Since a significant portion of the design resides in programmable read-only memory (PROM) and programmed array logic (PAL) chips, an attempt is made to at least describe the function of their programs. Actual signal and chip names are used wherever possible.

Overview

The overall arrangement of a finished terminal is shown in Figure 1. All cables connecting the main enclosure to the outside world are located in the back, along with the main power switch. The mouse is connected via a 9 conductor cable and 9 pin "D" connector, while the keyboard connection is through an 8 conductor telephone cable and modular jacks.

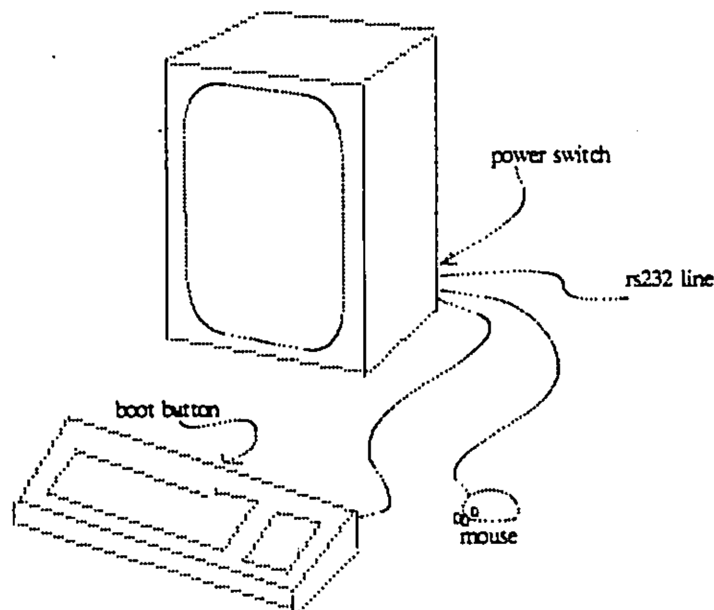
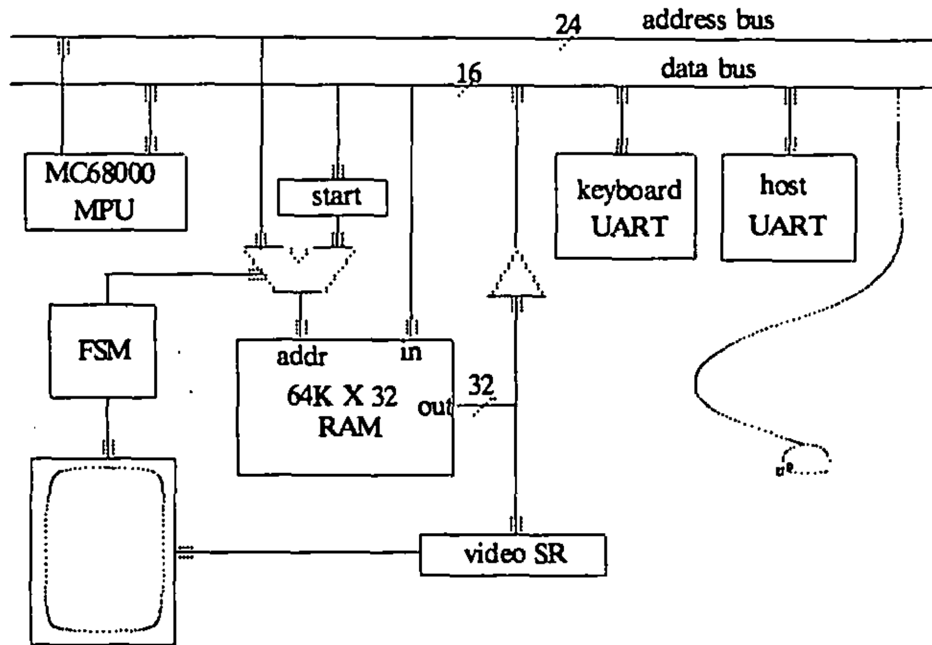


Figure 1. Major Blit components



igure 23. Blit data paths

Figure 2 shows the major elements and data paths of the Blit. The MC68000 microprocessor accesses EPROM storage and peripheral circuits through an unbuffered data bus and an address "bus" for which the MC68000 is always the master. A finite state machine generates sync signals for the display and sequences through a set of contiguous addresses in RAM storage. The RAM array is two-ported between the FSM and the MC68000 and operates asynchronously from either subsystem. The RAM chips are refreshed as a natural consequence of display refresh.

Following are descriptions of the major functional blocks, roughly in order of their dependency on each other.

Video Sync Section

The Ball HD15 monitor is scanned at a line rate of 32.00 KHz. Vertical and horizontal sync are controlled by individual ROM-counter state machines clocked at 4 megahertz. These state machines generate enable, load, and clock signals for counters, registers, and flip-flops. Those signals serving as clock inputs to other chips are first deglitched by clocking them into a state register **cked.** Enable and load signals are fed directly into synchronous chips.

The two counters **h_cnt<0:1>** of the horizontal state machine are always enabled, and reset (by loading zero) when the count reaches 127. The horizontal ROM **h_rom** splits this up into horizontal trace and horizontal flyback, issuing **h_sync** when the count is around 20. This is roughly in the middle of the adjustment range of the horizontal sync delay of the Ball monitor. The monitor should actually start flyback at around count=125.

h_rom also issues an enable signal for the vertical counters **v_cnt<0:2>** twice during horizontal scan. Because the display is interlaced, the vertical state machine resets after counting an odd number of counts. Allowing for the display of 512 lines in each field plus the approximately 500 usec vertical flyback time of the Ball monitor, the period of **v_sync** is 528.5 line times. This is about 17 ms.

h_rom generates most of the interesting signals to do with display timing and display memory access. The most complicated of its tasks is controlling the display address counters **a_cnt<0:3>** with count enable **a_en** and load enable **a_ld-** signals. This is done in concert with display memory access signals in order that the bits on the screen appear to come from a contiguous array of 100K bytes.

The display address is incremented by 1 after each access, except at the end of the horizontal scan. Since the display is interlaced, the address count must be incremented by 25, the number of long words in a scan line. This is done by enabling the address counters during every clock tick of the horizontal flyback.

During vertical flyback he address counters are loaded with the starting address held in the 16-bit register **a_st<0:1>**. This address points to the **longword** displayed first in each display frame time. The starting address is loaded twice during each line time of vertical retrace. On odd display fields this address is last loaded just before horizontal flyback, so it gets incremented by 25 before the first display access of the field. Interlacing is thus achieved at the cost of one flip-flop.

While the dual-ported memory is asynchronous, it is fast enough so that the display logic is guaranteed an access within 4 clock ticks (about 1 usec). The resulting acknowledge **dready-** clears the display request **dreq-** and latches the 32 bits of data from the RAM chips in latches **d_out<0:3>**. During horizontal scan, a display request **dreq-** is generated every 4 clock ticks, followed by the **srload** signal that enables the video shift registers **s_sr<0:3>** to be loaded from the display data latches.

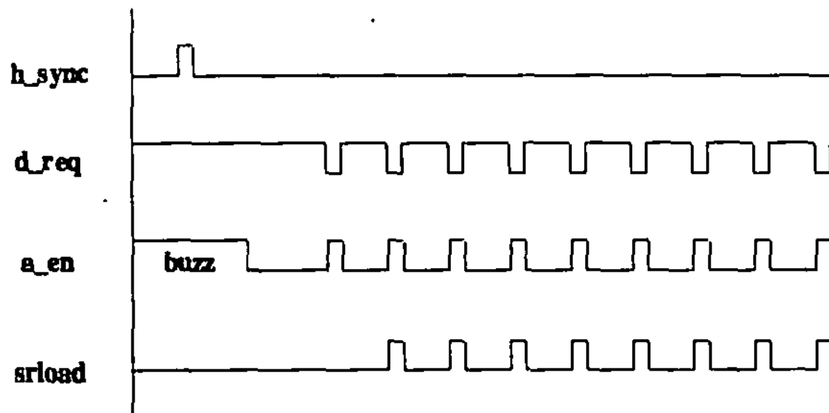


Figure 3. finite state machine timing

Figure 3 shows the relationship between dreq-, h_sync, srload, and a_en. Except for dreq-, these signals depend only on the state machines for proper functioning. dreq- can hang low if the memory controller doesn't work and fails to supply dready- to clear the flip-flop ff0.

Video output uses two stages of shift registers, primarily for reduced power consumption and package count. The 32 bits of RAM data are permuted before they are latched and subsequently loaded into the first shift register stage. This stage consists of 4 74LS166 8-bit shift registers clocked at 8 MHz. The 4 outputs of this stage is parallel- loaded into one 74S195 4-bit shift register clocked at the video bit rate of 32 MHz. This undoes the permutation of data for output to the display, high bit first.

Since display requests are not made during horizontal or vertical flyback, the first stage of shift register will not be loaded during these times. Rather, a constant value v_inv, which is always shifted into the first stage, appears as the output of the first stage. The output of the **last** stage is exclusive ORed with v_inv, taking care of blanking and inverse video at the same time.

Memory Control

The RAM array is shared by the display logic and the MC68000 microprocessor. To the memory control logic these sections appear as independent asynchronous processes. Memory requests are fielded on a first-come first-served basis by an arbiter which then passes the request to the memory timing controller, which generates the RAS/CAS signals for the RAM chips and control signals for the address multiplexors.

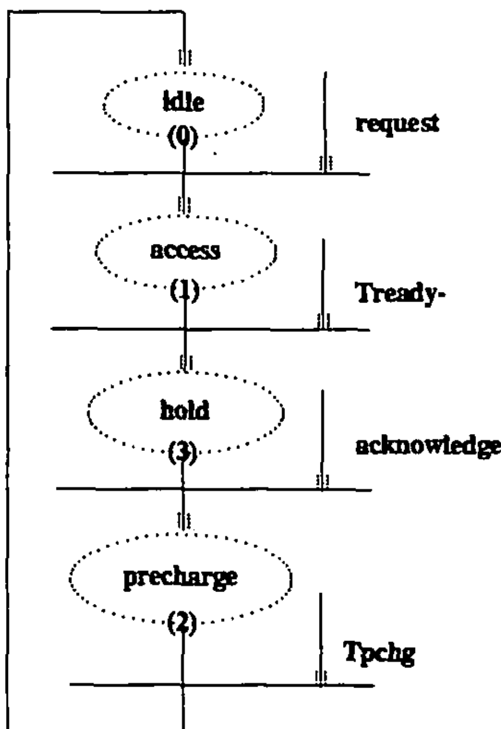


Figure 4. RAM controller state diagram

The arbiter consists of a pair of cross-coupled low-power Schottky NOR gates. The inputs to these gates are negative logic. Hence, the arbiter is really a flip-flop which in steady state is being set from both sides. The arbiter flips or flops in about 10 nsec after one of the requests is issued. If both requests are issued within about 100 picoseconds of each other, the arbiter takes another 15 nsec or so to decide. If the requests arrive **simultaneously** (really - the difference is measured in **hundreths** of an inch of wire) the arbiter goes into a metastable state before deciding. The low-power Schottky realization of this circuit does not oscillate, and in its metastable state the outputs (which are positive logic) are below the ON threshold of TTL logic. (Schottky arbiters, on the other hand, oscillate readily.)

In the current Blit design the MC68000 and display logic are ultimately run from the same clock, so true asynchronous operation is not really needed. However, a design using a faster version of the MC68000 would almost certainly use separate clocks for the microprocessor and the display.

The memory timing controller itself consists of a PAL chip and a 200 nsec delay line with 10 taps. Together they comprise a four state machine with four state transition arcs, each arc corresponding to one signal transition. This fact allows a feedback implementation in PAL logic without race.

The four states of RAM access are **idle**,access, and **precharge.** In the **idle** state there is no activity in the RAM chips and a logic "1" signal has propagated the length of the delay line. When a request comes in, the state changes to **access** and a logic "0" is sent down the delay line. RAS, CAS, and address multiplexing signals are generated as this "0" passes various timing points. When the "0" passes Tready- the state changes to **hold,** an acknowledge is issued, and the RAMs present their data until the corresponding request is removed. When this happens a "1" is sent down the delay line and the state is **precharge** until the "1" passes Tpchg. The state then goes to **idle,** and the controller is ready to service another request from the arbiter. Figure 5 shows a display cycle followed by a MC68000 cycle, with the second request arriving before the display cycle is finished.

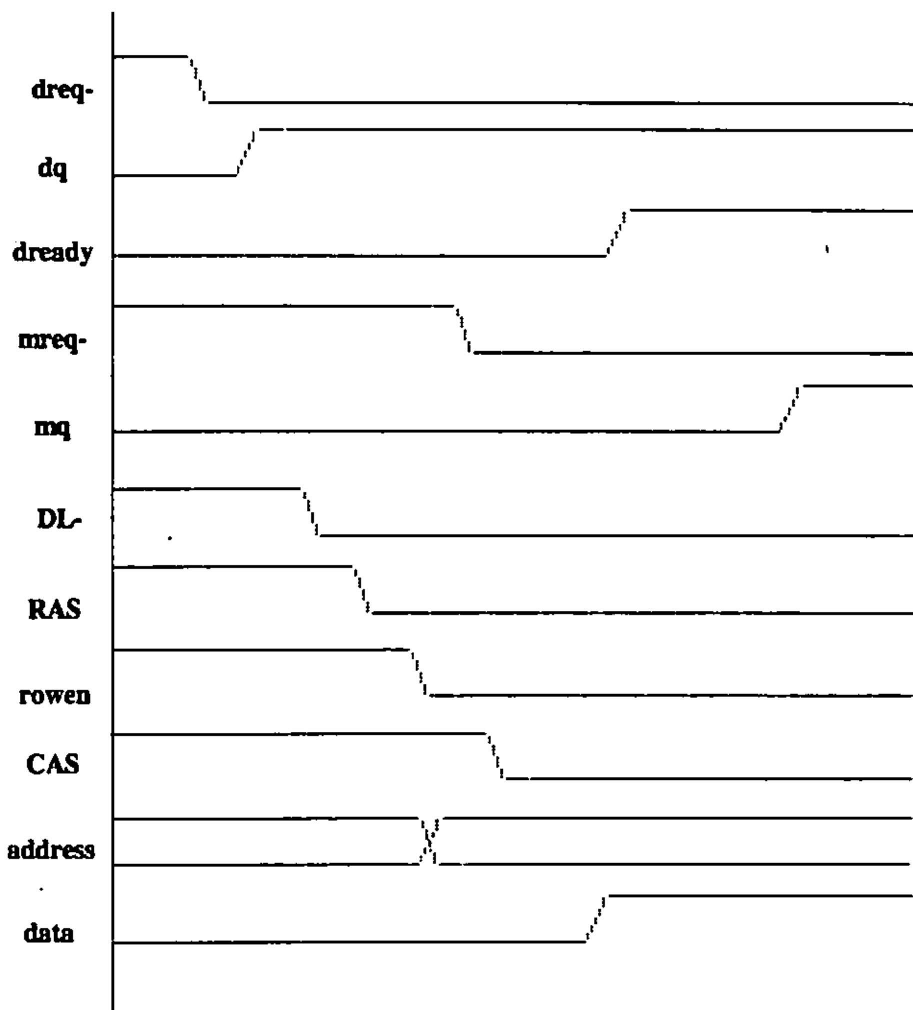


Figure 5. memory cycle timing

RAM Timing

Normally a Blit will be manufactured using 64K RAMs with an access time of 150ns and a cycle time of around 275ns. Because of delays in the memory controller and the fact that taps on a delay line are discrete, the observed access and cycle times are more like 220ns and 350ns, respectively. In the absence of competition for cycles, RAS will be issued approximately 75ns after the corresponding memory request. About 30ns of this is delay in the arbitration circuitry, while the rest is there to satisfy address setup timing in the presence of multiplexing and driving delays in the address paths to the RAM chips.

One glaring asymmetry in the design is that MC68000 cycle timing is different from display cycle timing. For some reason the MC68000 requires an early acknowledge from the memory controller in order to run at full speed. For this reason the **mready** signal is generated at CAS time rather than Tready time. While this allows uncontested memory cycles to complete without wait states in the MC68000, it does mean that one must be careful about adjusting the RAM timing.

The original Blit RAM timing was fairly crude. The following table shows a refinement of that timing that is both faster and more tolerant of slow parts.

signal	old tap	new tap
Trowen-	T3	T1
Tcas-	T5	T3
Tready-	T7	T6
Tpchg	T6	T4

Using a 200ns delay line this change moves **Trowen-**, **Tcas-**, Tpchg up by 40ns, and **Tready-** by 20ns, resulting in an 80ns improvement in cycle time.

68000 Microprocessor

The MC68000 accesses peripherals and operates out of ROM and RAM storage via an unbuffered 16-bit data bus. This is possible through the use of MOS and low-power Schottky peripherals and a short bus length. An address "bus" is generated by a set of 74LS373 tri-state latches which are always enabled.

The RAM array is dual-ported between the MC68000 microprocessor and the display logic. The address lines to the RAM chips are multiplexed between the address bus and the display memory address. The 32 RAM outputs are multiplexed to the 16-bit data bus via tri-state buffers. Both multiplexing functions are controlled by the two-port memory controller. Since the display access to the RAM array is read-only, the data inputs to the RAM chips are connected directly to the data bus.

The MC68000 is reset on power-up and whenever Data Carrier Detect (pin 8) is removed from the RS232 connection. In addition, a boot pushbutton is provided which can lower Data Terminal Ready (DTR) or reset the MC68000, depending on switch settings. Normally the switches should be set to reset the CPU without hanging up the phone.

The control circuitry for the MC68000 is largely out of the Motorola manual. ROM timing is generated by a 74164 shift register clocking in "1" signals at 16 MHz. The shift register is cleared at the beginning of an access to ROM, and a "ready" signal is generated as the "1" passes a timing point on the shift register.

The MC68000 has a compatibility mode that can be invoked when accessing peripheral devices. In this mode it behaves like a Motorola **6800** 8-bit microprocessor. This mode is used for all non-storage accesses. By feeding back all peripheral accesses to the MC68000 through vpa- (valid peripheral address), the MC68000 treats them as **6800** bus operations, for which it generates appropriate signals. In this mode the MC68000 also generates its own interrupt vectors. Using this "simple" peripheral peripheral access mechanism essentially allows one to throw parts at the MC68000 and have it run.

The keyboard and RS232 interfaces both use Motorola 6850 UART chips, with the keyboard running at 300 baud and the RS232 interface running at a maximum of 19200 baud.

All frequencies in the current Blit design are derived from a 32-33 MHz crystal oscillator. Even the UART clocks, which are multiples of 300 baud, are generated by dividing the master clock by 13 first. With a 32.768 MHz master clock, The frequencies so generated are a couple percent high, which is fine for everything except 212 datasets.

The Blit address space is divided rather crudely, with the first 256K bytes of address referencing RAM, followed by 128K bytes allocated to ROM and 128K bytes allocated to I/O. As in the Motorola 68000 design module, access to the low 8 bytes of RAM is diverted to ROM. This allows the MC68000 to fetch a constant program counter and stack pointer upon reset while keeping the interrupt and trap vectors in RAM.

A primitive form of bus error detection is used in the Blit. The address decoding logic detects write access to ROM as well as any access to the first 8 bytes of RAM. If the flip-flop at I/O location 50 is set to 2, such accesses cause the bus error signal **berr-** to be sent to the MC68000, causing a trap. This simple trick catches a fairly large class of runaway pointers, including attempts to follow a NULL pointer, without the complexity (or generality) of real

memory protection.

The following table gives the base addresse, interrupt priority, and vector location of all Blit I/O devices.

location	priority	vector	device
00	-	-	-(mouse Y)
11	int5	116	host ACIA
02	-	-	mouse X
20	int4	112	mouse buttons
30	-	-	display start address
40	-	-	inverse video (1 = BonW)
50	-	8	bus error (2 = set, 0 = clear)
60	int2	104	keyboard ACIA
70	int1	100	vertical sync (0 = clear)

Mouse Interface

The mouse generates a pair of pulse trains for each of two axes of movement. These pulse trains are 90 degrees out of phase, with the **sign** of the phase shift determining the direction of movement along an axis. Each pair of pulse trains is decoded by a PAL chip programmed to generate UP/DOWN and CLOCK signals to a string of counter chips. The PAL chip also provides the low 4 bits of the count.

The mouse also has three pushbuttons. These buttons are XORed together and differentiated to generate an interrupt on any transition of any button. The actual button values can be read to the data bus through tri-state buffers.

Appendix A - Blit circuit board

Figure A shows the circuit board supplied in Blits manufactured by North Atlantic ACS. The board is a Mupac #3244886-01 wire wrapped card with Ansley 10 and 26 pin headers replacing the three-row headers supplied by Mupac.

The 26 pin rs232 connector mates with a 25 conductor flat cable terminating in the 25 pin "D" connector which actually plugs into the host terminal line. **Important:** pin 1 from the rs232 connector goes to pin 26 on the Ansley header.

signal	RS232 pin	
TXD - Transmit Data	2	
RXD - Receive Data	3	
DTR - Data Terminal Ready	20	
DCD - Data Carrier Detect	8	
SGND - Signal Ground	7	
CGND - Chassis Ground	1	

The 10 pin mouse connector ("mickey") mates with a 9 conductor cable terminating in the 9 pin "D" connector that the mouse cable plugs into. In this case pin 1 should match up between the two connectors and the flat cable involved.

Following is the pinout of the mouse connector. This corresponds to the pinout of the Swiss *souris* and not the Hawley *X063X* (read it upside down) mouse.

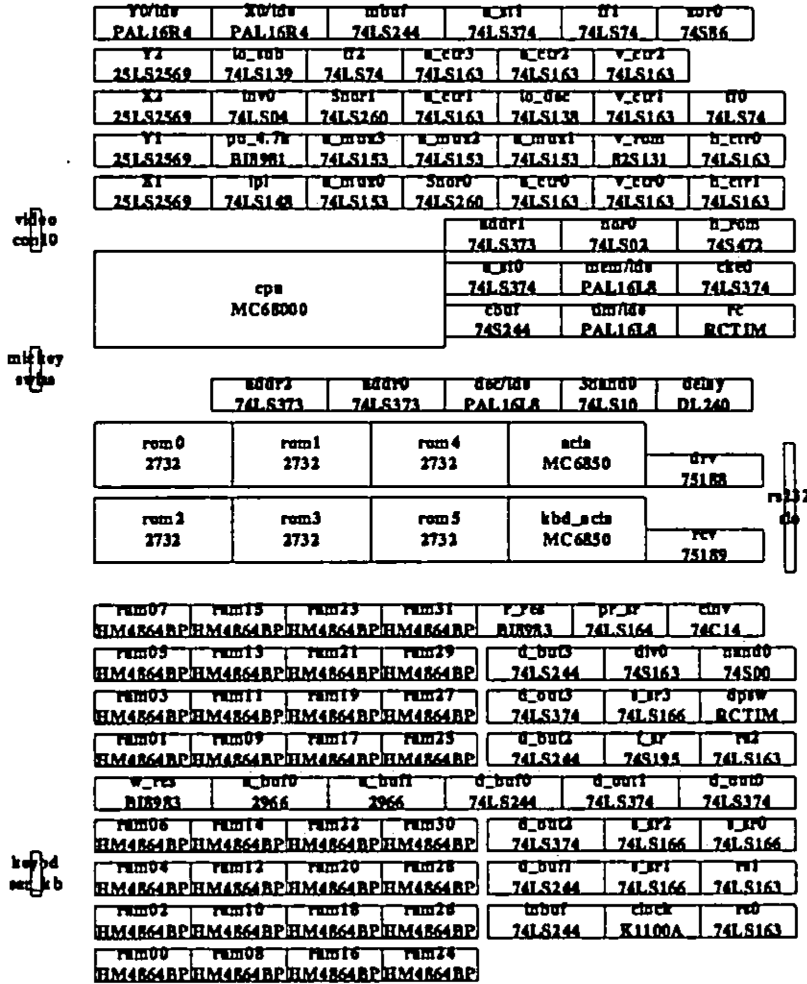


Figure A. Component placement

Pin	Signal
1	VCC
2	YB - trailing edge for increasing Y
3	YA - leading edge for increasing Y
4	XB - trailing edge for increasing X
5	XA - leading edge for increasing X
6	GND
7	middle button
8	right button
9	left button

The conductors of the video flat cable are numbered as follows:

Pin	Signal
1	video
2	GND
5	horizontal sync
6	GND
7	vertical sync
8	GND

These are the conductor assignments for the keyboard flat cable:

Pin	Signal
1	VCC
2	Keyboard output data
3	GND
4	chassis ground
6	keyboard receive data
8	boot pushbutton NC (normally closed)
9	boot C terminal
10	boot NO (normally open)

Parts

Many of the integrated circuits used in the Blit are incompletely or incorrectly specified in the schematic diagrams. The following is a correspondence table for real parts from schematic parts.

MC68000 MC68000L8 8 MHz microprocessor
HM4864BP Fujitsu MB8264-15 150ns 64K RAM plus 0.1uF bypass capacitor.
the bypass capacitor is placed adjacent to pins 8 and 9
of the 16 pin chip, simulating an 18 pin chip.
BI8983 Beckman Instruments 8983 22 Ohm (for r_res) and
47 Ohm (for w_res) series resistor packs
BI8981 Beckman Instruments 8981 4.7k Ohm pull up resistor pack
2966 AMD AM2966 address driver with series resistors
RCTIM 16 pin parts carrier with resistors and capacitors soldered on
2732 Intel D2732 200ns (observed) EPROM - the faster the better.
K1100A Motorola K1100A 32.768 MHz hybrid clock generator
decode
control
timing PAL16L8 chips copied from "D", "C", and "T" PAL chips
mpal PAL16R4 chips copied from "M" PAL chips
82S131 bipolar PROM copied from "V" PROM chip
74S472 bipolar PROM copied from "H" PROM chip

The Blit Programmer's Manual

Rob Pike

ABSTRACT

The Blit is a bitmap terminal with a Motorola 68000 microprocessor, 256K bytes of memory, an 800×1024-bit display and a graphics mouse. It is fully programmable, and because the hardware provides no graphics support other than the dual-ported display memory, graphics operations must be done in software. This document describes the environment available to the Blit programmer, and includes a description of the graphics and process control capabilities of the Blit software.

Overview

The Blit is a bitmap display terminal with an MC68000 microprocessor. The computer has 256K bytes of random access memory, of which 100K bytes map directly to the 800×1024-bit display. The computer also has 24K bytes of read-only memory, currently used to store the bootstrap loader and a simple terminal program. Access to the machine is through a standard RS-232 serial port. Attached to the Blit are a keyboard and a mouse.

There are two compatible programming environments available. One is a stand-alone system, in which the program is the only one running in the terminal. The other is a small time-shared environment, in which several programs, multiplexed in time and screen-space, coexist. A compile-time flag indicates which environment the program will use. This document explains the environments and how to use the Blit.

Compiling

The C compiler (*mcc*), loader (*mld*) and assembler (*mas*) are in the directory /usr/jerq/bin. MC68000 C has 16-bit shorts and ints and 32-bit longs and addresses. The difference in size between an `int` and an address implies that null pointers *must* explicitly be cast to pointers: `foo((char *)0)` passes to `foo` a 32-bit zero but `foo(0)` passes a 16-bit zero. The 68000 is a Big-Endian: the address of a word is the address of its high byte, and the address of a longword is the address of its high word (the opposite of the VAX, for example). This is largely irrelevant except when dealing with the display. To compile a standard C program, say

```
mcc −[jm] source.c
```

One of **−j** or **−m** must be specified, even when compiling object modules for a later load. The flag **−j** invokes the stand-alone world, compiling a program suitable for *jx*(1). The flag **−m** compiles for the multiplexed world of *mpx*(1).

Running a program

The Blit has no resident programs except for a boot loader and a simple terminal program. The terminal program is invoked whenever the Unix machine toggles the DCD lead of the RS-232 connection, or when the Blit is powered on. When DCD drops, the Blit processor stops executing, and when DCD rises again, the terminal program begins execution. (This dependence on DCD is planned to be removed.) When the terminal program receives a control-P character, octal 020, from Unix, it invokes the boot loader. The boot loader clears

the screen, except for a small horizontal glitch in the upper left corner, and waits for the Unix machine to send a packetized representation of the program to run. The program is copied into Blit RAM, starting in low memory. The boot loader sets the display to point to low memory, so the incoming program can be viewed on the display during the boot process. To restart the Blit after some disaster, push the boot button. This will either hang up the connection to Unix, or merely restart the terminal program, depending on the hardware switch settings — see your Blit Owner's Manual for more details.

The Unix program *68ld* talks to the Blit to download a Blit program, such as the terminal program. When the terminal program is running, the Unix program *jx* can be used to down-line load and execute programs compiled with the -j option of *mcc*. The *jx* world includes a Standard I/O interpreter (obtained by #include <blitio.h>). Calling `exit()` will reboot the terminal program and make available the characters sent to standard output and standard error, which have been saved in a file during execution. *jx* calls *68ld* to do the boot loading; *jx* is primarily a control program to interpret Standard I/O calls. *jx* also runs properly under *mpx*: if the source files are all compiled with *mcc −m* the resulting binary may be run in a layer using *jx*. See the *jx* manual page, *jx*(1) for more information.

The *mpx* world is an asynchronous, time-shared "layer" environment in which several programs may be sharing the processor and graphics screen. *68ld* (and hence *jx*) can download layer processes into *mpx*. See *mpx*(1) for information on running *mpx*; the rest of this manual assumes some understanding of its workings. When a layer is created, a terminal process runs in the layer, copying Unix characters to the screen and sending keyboard characters back to the standard input of the Unix process (typically the Shell or a text editor). If a Unix program (usually *68ld*) sends the *ioctl* `JBOOT`, the boot loader is invoked for that layer, and a program compiled with the −m flag of *mcc* may be loaded and run in the layer. If that process calls `exit()`, or if the *ioctl* `JTERM` is sent, the layer will automatically restart the terminal program.

Data Types

The graphics functions use a number of data types, defined in `/usr/include/blit.h`. This section explains their definitions and interpretation.

A `Word` is a short (16-bit) integer: type `int`. It is the quantum of memory used by the graphics software.

A `Point` is two integers, specifying a point on the screen or in a bitmap:

```
typedef struct{
      short x;
      short y;
}Point;
```

The coordinate system is oriented with x positive to the right, ranging from 0 to `XMAX-1` across the screen, and y positive down, ranging from 0 to `YMAX-1` from top to bottom. (The −1's will be explained later).

A `Rectangle` is specified by two Points:

```
typedef struct{
      Point origin;    /* Upper left */
      Point corner;    /* Lower right */
}Rectangle;
```

`origin` is the location of the upper left corner (minimum x and y) of the `Rectangle`, and `corner` is the lower right (maximum x and y). By convention, the right (maximum x) and bottom (maximum y) edges are excluded from the rectangle, so abutting rectangles have no points in common. This is why there are −1's in the ranges of x and y across the screen: the `Rectangle` describing the screen is `{0, 0, XMAX, YMAX}`.

A Bitmap is a storage area for a rectangular image, defined by the enclosing rectangle, and the storage itself:

```
typedef struct{
      Word *base;             /* pointer to start of data */
      unsigned width;         /* width in Words of total data area */
      Rectangle rect;         /* rectangle in data area, screen coords */
}Bitmap;
```

Bitmap storage is arranged just like an array, with x varying fastest. base points to the word of storage containing the upper left corner of the Bitmap. The following words contain the rest of the uppermost scan line of the image. The last word of one scan line is followed immediately in memory by the first word of the next scan line. width is the number of Words of storage consumed by a scan line. rect defines both the image's shape and coordinate system: it is the coordinate system inside the Bitmap, so rect.origin is the coordinates of the upper left point in the bitmap image, and is not necessarily (0, 0). Graphics operations performed on a Bitmap are clipped to rect. rect is often, although not necessarily, also the screen coordinates of the rectangle saved in the Bitmap. The screen itself is described by a global Bitmap called display.

A Texture is a 16×16 rectangle of bits which defines a dot pattern. A Texture is declared something like this:

```
Texture grey={
      0xAAAA, 0x5555, 0xAAAA, 0x5555,
      0xAAAA, 0x5555, 0xAAAA, 0x5555,
      0xAAAA, 0x5555, 0xAAAA, 0x5555,
      0xAAAA, 0x5555, 0xAAAA, 0x5555,
};
```

The Texture looks much like a 16×16 Bitmap: the first Word is the first horizontal scan of the texture, the second is the next, and so on. The routines which use Textures fix the patterns to the absolute screen coordinates, so that, for example, if two overlapping screen rectangles are textured with the same Texture, the dots in each rectangle will mesh properly to form a constant pattern.

Most graphics routines take a Code argument to specify the logical function to use for drawing. The values and meanings of a Code are:

```
F_STORE          target =  source
F_OR             target |= source
F_XOR            target ^= source
F_CLR            target &= ~source
```

In other words, if a Rectangle is copied to another place with Code F_OR, the result will be the bitwise OR of the source Rectangle and the contents of the target area. For line-drawing, points, etc., F_STORE should be avoided; F_OR is the equivalent. F_STORE is only meaningful for copying Rectangles and drawing textures.

Graphics

A Blit program must begin

```
#include <blit.h>
```

to define the data types discussed above, some manifest constants such as XMAX and YMAX, and other useful oddments. Programs to be run by *jx* should also include <blitio.h>, after <blit.h> — see *jx*(1).

Because a program may be running stand-alone, and thereby using the whole screen, or running under *mpx*, confined to a rectangular portion of the screen, there are two coordinate systems: screen and layer coordinates. Screen coordinates refer to the actual pixels of the

screen: `(0, 0)` is the upper left corner, and `(XMAX-1, YMAX-1)` is the lower right corner of the screen. Layer coordinates refer to the rectangular portion of the screen actually used by the program, and are scaled so that `(0, 0)` is the upper left corner, and `(XMAX-1, YMAX-1)` is the lower right corner *only of the screen area available to the program.* (This portion of the screen is called the program's layer.) Layer coordinates are therefore scaled, and adjacent locations in layer coordinates do not necessarily refer to separate screen pixels.

Several global structures describe the screen portion available to the program. `display` is a `Bitmap` that defines the layer. As in all `Bitmaps`, the coordinate system of `display.rect` is screen coordinates. Unfortunately, for technical reasons, the rectangle defined by `display.rect` includes the border surrounding the layer in *mpx*, so the global `Rectangle Drect` is available: it defines the screen area *inside* the border, again in screen coordinates. The `Rectangle Jrect` is always defined to be `{0, 0, XMAX, YMAX}`, even under *mpx*, and therefore describes the screen area in layer coordinates.

Most of the graphics routines take their arguments in screen coordinates, and require a target `Bitmap` to be explicitly passed (by reference, never by value), but those routines whose names begin with a `'j'`[†] take layer coordinates, and operate on the `display Bitmap`. Many programs — "asteroids" is the canonical example — always want to have graphical operations scaled to fit the layer, so that the layer appears to the program just like a full screen. Such programs need therefore only use the `'j'` routines. These routines also have the convenience that they remember a current point — much like dot in *ed* — and so make it simple to draw a curve, for example, by a set of line segments. The appendix describes how these routines interact with the current point. Programs dealing largely with text or off-screen bitmaps must be more careful with how objects appear in the layer, and must work at least sometimes in screen coordinates, avoiding the `'j'` routines. In general, both layer and screen coordinates will be used, which can be confusing, but the careful programmer will be able to write a program that can run unchanged either stand-alone or under *mpx*. Except for the coordinate differences and details of implementation, the stand-alone and *mpx* environments are identical, and all the routines described in the appendix have the same specifications in either environment.

Here begins an introductory walking tour of Blit graphics. Every program must include `<blit.h>` and call `jinit()` before calling any other routines. `jinit` clears the screen or layer to all zeros and sets the video bit to black on white: one bits appear as black on a white background. The routines `BonW()` and `WonB()` control the sense of video, either black ones on white zeros or white ones on black zeros. Unfortunately, `WonB` mode under *mpx* is not the real thing, but an incredible simulation which is not completely accurate, although it may be someday.

Here is a program to draw a grid with spacing defined by `XSPACING` and `YSPACING`:

[†] The origin of this character is lost to history.

```
#include <blit.h>
#define     XSPACING    25
#define     YSPACING    25
main(){
      Point p;
      jinit();
      /* Vertical lines */
      for(p.x=p.y=0; p.x<XMAX; p.x+=XSPACING)
            jsegment(p, Pt(p.x, YMAX), F_OR);
      /* Horizontal lines */
      for(p.x=p.y=0; p.y<YMAX; p.y+=YSPACING)
            jsegment(p, Pt(XMAX, p.y), F_OR);
}
```

Since the program uses `jsegment` to draw the lines, the total number of lines will be the same regardless of the layer's shape. The macro `Pt(x, y)` passes the `Point (x, y)` to `jsegment`, in lieu of structure-valued constants in C. `Pt` and its relatives `Rect` (`Rectangle` from four coordinates) and `Rpt` (`Rectangle` from two `Points`) only work in parameter lists.

The grid program draws the lines in `F_OR` mode, which simply turns on all the bits in the line. The grid may be erased again by executing the same code in `F_CLR` mode, which turns all the bits off. It may also be erased by simply clearing the screen, either

```
jrectf(Jrect, F_CLR);
```

or

```
rectf(&display, Drect, F_CLR);
```

For most graphical operations, `F_OR` and `F_CLR` are inverses, since one is the opposite of the other. It is *not* true, however, that following a `F_OR` operation with a `F_CLR` operation will undo the first, since the `F_CLR` will clear all bits, even those that were set before the `F_OR`. This brings us to the marvelously versatile `F_XOR` mode.

`F_XOR` *inverts* each target bit, turning ones to zeros and zeros to ones (all the modes have slightly different meanings in `bitblt` and `texture`; we shall return to these later). `F_XOR` has several useful properties:

- `F_XOR` is its own true inverse: two adjacent identical `F_XOR` operations cancel exactly, restoring the screen to its previous form.
- `F_XOR` is commutative: `F_XOR` operations may be executed in any order to produce the same final result. Combined with its inverse property, this means that a `F_XOR` operation may be cancelled at any later time by another `F_XOR` operation.
- Because `F_XOR` is its own inverse, the same code can be used to draw or undraw a picture. This is a common action: the mouse cursor, for example, is updated by calling the same routine, using `F_XOR` mode internally, to undraw the old position and draw the new one (the order is irrelevant!).
- Because the mouse cursor is tracked in `F_XOR` mode, any operations done in `F_XOR` mode will never leave "mouse tracks". The same applies to the shading dots in *mpx*.

Of course, these properties break down if `F_XOR` operations are mixed with other modes. However, it is not only possible, but common and even natural, to do *all* graphics in `F_XOR` mode. Again, "asteroids" is the canonical example. `F_XOR` can also be used to simulate the video bit: inverting the screen or some portion of it with a call such as

```
jrectf(Jrect, F_XOR);
```

changes the sense of all subsequent and *previous* `F_XOR` operations.

The `point`, `segment`, and `rectf` routines define a set of points upon which to peform some action defined by the `Code` argument, such as `F_XOR`. There is another, more

fundamental, way to define these routines which both explains the names of the Codes and introduces *the* bitmap operator, `bitblt`. The graphical operators may be defined as constructing an imaginary off-screen bitmap, commensurate with the screen, set to zeros everywhere except the points corresponding to the points on the screen where the operation is to be performed: the points of the approximating line of `segment`, the single point of `point`, or the points in the rectangle of `rectf`. The routines then simply, bit for bit, perform a logical function from the imaginary bitmap into the screen (or whatever target bitmap has been specified):

```
F_OR:       screen location |= imaginary location
F_CLR:      screen location &= ~imaginary location
F_XOR:      screen location ^= imaginary location
```

There is an obvious fourth code to add (and several less obvious ones):

```
F_STORE:    screen location = imaginary location
```

The operator to perform this function is `bitblt`, although (for efficiency rather than practicality) the graphics functions are written without using `bitblt`. `bitblt` is actually more general, doing the bit for bit function from an arbitrary `Rectangle` in an arbitrary `Bitmap` to an arbitrary `Rectangle/Bitmap` destination. It is declared like this:

```
bitblt(sourcemap, sourcerect, destmap, destpt, code)
     Bitmap *sourcemap, *destmap;
     Rectangle sourcerect;
     Point destpt;
     Code code;
```

`sourcerect` and `destpt` are in the coordinate systems of their particular `Bitmaps`. `destpt` specifies the `Point` corresponding to the `origin` of `sourcerect` in the destination `Bitmap`; the `corner` is derived from the destination `Rectangle` being congruent to the source. The source and destination `Bitmaps` may be the same, and the source and destination `Rectangles` may even overlap; `bitblt` always does the assignments in the correct order. Here, for example, is how to scroll the screen up 16 pixels:

```
bitblt(&display, Rect(0, 16, XMAX, YMAX),
     &display, Pt(0, 0), F_STORE);
/* now clear the bottom line */
rectf(&display, Rect(0, YMAX-16, XMAX, YMAX), F_CLR);
```

For simplicity, this code assumes it is running stand-alone, and can scribble freely on the screen.

A common use of `bitblt` is to copy a prepared picture — such as a character — from off-screen onto the display. The following complete program simulates a bouncing ball by building an off-screen bitmap and using `bitblt` to make it bounce:

```
#include <blit.h>
Point spot, v, a;
int Topx, Topy;
int Botx, Boty;
Bitmap *ball;
#define   R    32
main()
{
     jinit();
     Topx=Drect.corner.x-R;
     Topy=Drect.corner.y-R;
     Botx=Drect.origin.x+R;
     Boty=Drect.origin.y+R;
     a.x = 0; a.y = 1;
     spot.x = (Botx+Topx)/2;
     spot.y = Boty;
     /* allocate a bitmap */
     ball=balloc(Rect(0, 0, 2*R+1, 2*R+1));
     rectf(ball, ball->rect, F_CLR);
     /* draw ball as a disc */
     disc(ball, add(ball->rect.origin, Pt(R, R)), R, F_XOR);
     drawball();
     v.x=1; v.y=0;
     for(;;){
          drawball();     /* undraw the old one */
          v = add(v, a);
          spot = add(spot, v);
          if(spot.x >= Topx) {
               spot.x = 2*Topx - spot.x;
               v.x = -v.x;
          } else if(spot.x <= Botx) {
               spot.x = 2*Botx - spot.x;
               v.x = -v.x;
          }
          if(spot.y >= Topy) {
               spot.y = 2*Topy - spot.y;
               v.y = -v.y;
          } else if(spot.y <= Boty) {
               spot.y = 2*Boty - spot.y;
               v.y = -v.y;
          }
          drawball();     /* draw the new one */
          sleep(4);
     }
}
drawball(){
     bitblt(ball, ball->rect, &display, sub(spot, Pt(R, R)), F_XOR);
}
```

`disc`, `balloc`, `sleep` and the arithmetic routines are explained in the appendix. Note also the use of `F_XOR` in `drawball`.

Although a bitmap may only have pixel values zero and one, and is therefore incapable of grey-scale or color drawings (except for green), by turning on pixels in a regular pattern the programmer may draw textures, which may be used for many of the same functions, graphical and mnemonic, as colors on color displays. Textures are specified, as described

above, by a 16×16 array of bits. The simplest way to visualize how textures are implemented is to recall the imaginary off-screen bitmap described earlier, and imagine the array replicated to cover the entire area of the bitmap. Then, rectangles may be textured by copying the corresponding bits, again according to a Code argument, from the imaginary bitmap to the target bitmap or screen. By this definition, images may be textured identically by dividing them up into any set of rectangles: because the texture is bound to the coordinates of the imaginary bitmap, the texture will be replicated smoothly across the image. For example, the display area may be textured by:

```
texture(&display, Drect, grey, F_STORE);
```

but the following, much slower, code will produce the *same* result:

```
register i, j;
for(i=Drect.origin.x; i<Drect.corner.x; i++)
    for(j=Drect.origin.y; j<Drect.corner.y; j++)
        texture(&display, Rect(i, j, i+1, j+1),
            grey, F_STORE);
```

Resources and I/O

I/O from a Blit program is totally different from I/O in a Unix program. The differences are inherent, essential and invaluable, and all stem from the Blit being a *terminal* rather than a CPU running a Unix process.

A program running in a Blit does *not* possess a standard input or standard output, *jx* notwithstanding. Put yourself in a Blit's shoes: connected to a host Unix machine by a single RS-232 connection, your only link to the outside world is a bidirectional, low-speed link. As a terminal, your usual job is twofold: first, send characters typed on the keyboard down the line to the standard input of a Unix process, and second, receive characters sent from the standard output of a Unix process and draw them on the screen. The ideas of standard input and output are meaningless in your isolated world. Moreover, the methods of Unix I/O are of no help. Characters requiring your attention may appear at any time from either the host or the keyboard, so you must be prepared to deal with two I/O devices at any time. A Unix read is useless for this, because there is no way to read from two file descriptors at once, and if you read from, say, the keyboard, you may miss characters sent from the host. Instead of the Unix model, Blit I/O requests never block (wait for a resource to become available). Instead, they return an error status if the request cannot be serviced. A Blit program may, however, explicitly ask to wait for a resource to become available.

A Blit program has access to four I/O resources:

RCV characters received from Unix
SEND characters sent to Unix
KBD characters typed on the keyboard
MOUSE the graphics input device

A program wishing to use one or more of these devices indicates its intention by a `request()` call. Multiple resources may be allocated by OR'ing the resource names together; for example

```
request(KBD|MOUSE);
```

allocates the keyboard and mouse to the program. A `request()` call overrides all previous calls — all desired resources must be `requested` in one call. The semantics of this allocation are described below.

Characters sent to and from Unix are kept in two queues, the RCV queue for characters coming from Unix to the program, and the SEND queue for characters going to Unix. Characters read from the RCV queue were typically written on the standard output or standard

error of the *Unix* process associated with the layer (imagine that the stand-alone world is one full-screen layer). Characters put on the SEND queue are sent to Unix and may be read on the standard input of the associated Unix process. Note that the terms "standard input," "standard output," and "standard error" apply to the *Unix* process, not the process running in the Blit. The RCV and SEND queues in the Blit are similar to standard input and output, but RCV characters come from Unix standard *output*, and SEND characters go to Unix standard `input`. In other words, the Blit and Unix processes have I/O cross-coupled, much like two Unix processes connected by a pair of pipes.

The routine `rcvchar()` reads the next character from the process's RCV queue, or -1 if the queue is empty. `sendchar(c)` similarly puts the character c on the SEND queue, and thereby transmits it to Unix. `sendnchars(n, p)` similarly transmits the n characters pointed to by p. Characters sent by `sendchar()` are processed by the teletype input routine (i.e., erase and kill processing is performed on them), while those sent by `sendnchars()` are sent to the Unix program without further processing.

If the keyboard has been `requested`, `kbdchar()` will return the next character typed to the process on the keyboard, or -1 if no characters are outstanding. If the keyboard has not been `requested`, typed characters are sent automatically, through the teletype input routine, to the standard input of the Unix process. In other words, if the keyboard is not `requested`, it is coupled automatically to the standard input of the Unix process. (In the stand-alone world, the characters are thrown away if the keyboard is not `requested`.) The standard *mpx* terminal program, for example, does not request the keyboard — typed characters are simply sent to the Unix process. But many programs — for example *jim*, the Blit text editor — wish control over the keyboard. As a trivial example, the ball program shown earlier can be made to exit when a character is typed by adding

```
request(KBD);
```

after the call to `jinit()`, and replacing the `for(;;)` by

```
while(kbdchar() == -1)
```

The semantics of mouse allocation are a little more complicated, and relate to the user interface of *mpx*. *mpx* has a current layer — at most one layer which currently owns the keyboard and mouse. Clicking button 1 in a layer, or using the "Current" menu button, hands both the keyboard and mouse to the process in the indicated layer. Pressing a mouse button in the current layer passes the button hit to the layer if it has `requested` the mouse; otherwise, it is interpreted by the system. For example, `button2()` is true only when the process is in the current layer *and* the mouse cursor points to a visible portion of the layer *and* the process has `requested` the mouse *and* button 2 is depressed. The next section explains the interface to the mouse in more detail.

Two primitives, `own()` and `wait()` inform a process about its resources. `own()` returns a bit vector of the allocated resource status:

`own()&RCV`	the RCV queue has a character
`own()&SEND`	always true
`own()&KBD`	the KBD queue has a character
`own()&MOUSE`	`mouse` structure is up to date

Note that ownership of the mouse is independent of the status of the buttons; basically it implies that the system will pass to the current process mouse coordinates and button status as they change. (See the next section for more information about the `mouse` structure.) A process never owns a resource it has not `requested`. (SEND is always implicitly `requested`.) A typical use of `own()` is to see which of several I/O devices need service, like this:

```
main(){
      int got;
      jinit();
      request(KBD|RCV);
      for(;;){
            got=own();
            if(got&KBD)
                  kbdservice();
            if(got&RCV)
                  rcvservice();
            ...
      }
}
```

The Blit operating system does not do interruptive scheduling, so processes must explicitly give up the CPU to enable other processes to run. Therefore, the above program could lock out all other processes running. `wait()` resolves this difficulty by suspending a process until a resource becomes available. `wait()` takes an argument resource bit-vector and waits until at least one of the resources becomes available, and returns which of the requested resources is available. Thus, the above example should probably be written:

```
main(){
      int got;
      jinit();
      request(KBD|RCV);
      for(;;){
            got=wait(KBD|RCV);
            if(got&KBD)
                  kbdservice();
            if(got&RCV)
                  rcvservice();
            ...
      }
}
```

Some programs, such as game programs, wish to run at all times. To enable other processes to run, a fake resource, CPU, which is always implicitly `requested`, may be waited for. `wait(CPU)` enables all other processes which are ready to run, and returns immediately after they have had a chance to run, regardless of the state of other resources. The inner loop of a game program might therefore look like:

```
for(;;){
      wait(CPU);
      updatedisplay();
}
```

The Mouse

The software does automatic cursor tracking and keeps relevant variables about the mouse in a global structure called `mouse`. For example, the following is a complete C program to draw a curve sketched by the mouse whenever button 1 is depressed:

```
#include <blit.h>
main(){
      Point p;
      jinit();
      request(MOUSE);
      for(;;wait(MOUSE)){
            if(button1()){
                  jmoveto(mouse.jxy);
                  for(; button1(); nap(2))
                    jlineto(mouse.jxy, F_OR);
            }else if(button3())
                    exit();
      }
}
```

`mouse.xy` is the mouse position in screen coordinates, `mouse.jxy` is in layer coordinates. The macros `button1()` and its obvious analogues are true when the corresponding buttons are depressed. The macro `button12()` is true when either of button one or two are depressed, and similarly for `button23()` and `button123()`. `nap()` suspends the process for some number (the argument) of 'ticks' of the video refresh. A tick is approximately 1/60 sec. The display is interlaced, so napping for two ticks guarantees that the line is on the screen before proceeding.

It is often necessary to turn off the cursor, for example if only coordinates are wanted, or to guarantee that drawing the picture does not interfere with the cursor. In the program above, the line may be drawn over the cursor, and when the cursor is moved away, "mouse tracks" will be left. The routine `cursinhibit` turns the cursor off until `cursallow` is called. The above program should not call `jlineto` directly, but instead a subroutine which properly brackets the call:

```
lineto(p, f)
      Point p;
{
      cursinhibit();
      jlineto(p, f);
      cursallow();
}
```

Characters and Fonts

The Blit has access to multiple character sets or *fonts*. A font is internally a single `Bitmap`, with some associated information. The `Bitmap` contains the bit pattern for each character, arrayed adjacently into a long horizontal stripe, as on the wall in first grade, like this:

ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz

(It is not necessary that the characters appear in ASCII order.) The characters in the `Bitmap` are all aligned on the same baseline. The `Font` structure is defined in `<font.h>`:

```
typedef struct{
      short n;          /* ascii value of last char in font */
      char height;      /* height of bitmap */
      char ascent;      /* top of bitmap to baseline */
      long unused;      /* for a rainy day */
      Bitmap *bits;     /* where characters are stored */
      Fontchar info[0..n+1];      /* n+2 character descriptors */
}Font;
```

The extra `Fontchar` is present to indicate the right edge of the n'th character. A single

Fontchar structure describes each character:

```
typedef struct{
      short x;            /* left edge of bits in bitmap */
      char top;           /* y of first non-zero scan-line */
      char bottom;        /* y of last non-zero scan-line */
      char left;          /* x offset of baseline (for kerning) */
      char width;         /* width of baseline */
}Fontchar;
```

To draw a character on the screen requires copying the appropriate rectangle from the font bitmap to the screen, at the correct location. For F_OR, F_XOR and F_CLR modes, the minimum enclosing rectangle of the character is all that need be copied. For F_STORE mode, the entire target rectangle, the size of a complete character, must be copied from the Font Bitmap to the destination Bitmap. The routine to draw a character c from Font fp in a Bitmap db at p with code f looks like this:

```
#include <font.h>
drawchar(c, fp, db, p, f)
      char c;
      Font *fp;
      Bitmap *db;
      Point p;
      Code f;
{
      Rectangle r;
      Fontchar *i=fp->info+c;

      if(f == F_STORE){
            r.origin.y = 0;
            r.corner.y = fp->height;
      }else{
            r.origin.y = i->top;
            r.corner.y = i->bottom;
      }
      r.origin.x = i->x;
      r.corner.x = (i+1)->x;
      bitblt(fp->bits, r, db, Pt(p.x+i->left, p.y+r.origin.y), f);
}
```

Note that the coordinate system places the origin of the tallest character at the specified point, not at the baseline. This behavior is consistent with the coordinate system of Rectangles, but requires some programming if characters from several fonts are to be placed on the same baseline. <font.h> must be included in any program that uses the Font or Fontchar structures.

Several service routines provide simple access to fonts stored on the host file system. These routines — infont, outfont, and getfont are described in the appendix.

Caveat Blittor

The Blit is a terminal, not a personal computer. A significant program to use the terminal must have some support from the host Unix machine, most likely to provide access to system resources such as files. In general, a Blit program requires two separate programs to run, and a protocol for their communication. This division has advantages: it allows a clean division of work between two processors, which can lead to a clean, simple design; and it results in a smaller program running in the terminal, which down-loads faster and consumes less precious memory. But the division must be made, and it can be very difficult to choose a

good place to divide. The low bandwidth connection imposes the further constraint that communications across the division should be tightly encoded. The *jx* system provides an easy solution, by *simulating* standard I/O functions in the terminal program. But that is not always a sensible approach. A text editor, for example, is untenable under *jx*, because a context search requires looking at the entire file, which must therefore be read by the program down the low-speed RS-232 line. A Unix program can do file I/O much quicker, so a better approach for an editor is to write a Unix program to do file I/O and associated functions, and let the Blit program manage the display and user interactions.

Don't be lulled by the simplicity of using *jx*. Designing a two-process protocol is a fair fraction of the work required to write a Blit program, but the effort is repaid by a smaller, more efficient system.

Appendix: The Blit Library

In the following summaries, all coordinates are screen or bitmap coordinates (which are scaled the same), unless specified as layer coordinates.

`Word` Word: quantum of display memory

```
typedef short Word;
```

A `Word` is a 16-bit integer, and is the unit of storage used in the graphics software.

`Point` Point: data structure for position on screen

```
typedef struct{
    short x;
    short y;
}Point;
```

A `Point` is a location in a `Bitmap`, such as the display. The coordinate system has x increasing to the right and y increasing down.

`Rectangle` Rectangle: data structure for rectangle on screen

```
typedef struct{
    Point origin;  /* Upper left */
    Point corner;  /* Lower right */
}Rectangle;
```

A `Rectangle` is a rectangular area in a `Bitmap`. By definition, `origin.x<=corner.x` and `origin.y<=corner.y`. By convention, the right (maximum x) and bottom (maximum y) edges are excluded from the represented rectangle, so abutting rectangles have no points in common. Thus, `corner` is the coordinates of the first point beyond the rectangle. The data on the display is contained in the `Rectangle {0, 0, XMAX, YMAX}`, where `XMAX==800` and `YMAX==1024`.

`Bitmap` Bitmap: data structure for bitmap

```
typedef struct{
    Word    *base;          /* pointer to start of data */
    unsigned width;         /* width in Words of total data area */
    Rectangle rect;         /* rectangle in data area, screen coords */
}Bitmap;
```

A `Bitmap` holds a rectangular image, stored in contiguous memory starting at `base`. Each `width` `Words` of memory form a scan-line of the image. `rect` defines the coordinate system inside the `Bitmap`: `rect.origin` is the location in the `Bitmap` of the upper-leftmost point in the image.

`Texture` Texture: data structure for texture

```
typedef Word Texture[16];
```

A `Texture` is a 16×16 dot bit pattern. `Textures` are aligned to absolute display

positions, so adjacent areas colored with the same `Texture` mesh smoothly.

Font Font, Fontchar: data structure for character set

```
typedef struct{
    short n;         /* ascii value of last char in font */
    char height;     /* height of bitmap */
    char ascent;     /* top of bitmap to baseline */
    long unused;     /* for a rainy day */
    Bitmap *bits;    /* where characters are stored */
    Fontchar info[0..n+1];/* n+2 character descriptors */
}Font;
typedef struct{
    short x;         /* left edge of bits in bitmap */
    char top;        /* y of first non-zero scan-line */
    char bottom;     /* y of last non-zero scan-line */
    char left;       /* x offset of baseline (for kerning) */
    char width;      /* width of baseline */
}Fontchar;
```

A `Font` is a character set. The character information is stored in the `Fontchar` structures. The actual images of the characters are stored in the horizontal Bitmap `bits`. A character at point p has its upper-leftmost dot, including empty space above it in the Bitmap , at p. Characters in the Bitmap abut exactly, so the width of a character c is `Font.info[c+1].x-Font.info[c].x`.

add add, sub, mul, div: arithmetic on Points

```
Point add(p, q) Point p, q;
Point sub(p, q) Point p, q;
Point mul(p, a) Point p; int a;
Point div(p, a) Point p; int a;
```

`add` returns the Point sum of its arguments: `(p.x+q.x, p.y+q.y)`. `sub` returns the Point difference of its arguments: `(p.x-q.x, p.y-q.y)`. `mul` returns the Point `(p.x*a, p.y*a)`. `div` returns the Point `(p.x/a, p.y/a)`.

addr addr: Word address of Point in Bitmap

```
Word *addr(b, p) Bitmap *b; Point p;
```

`addr` returns the address of the Word containing the bit corresponding to the Point p in the Bitmap b.

alloc alloc, free: allocate memory

```
char *alloc(nbytes) unsigned nbytes;
void free(s) char *s;
```

`alloc` corresponds to the standard C function `malloc`. It returns a pointer to a block of `nbytes` contiguous bytes of storage, or 0 (`NULL`) if unavailable. The storage is aligned on 4-byte boundaries. Unlike `malloc`, `alloc` clears the storage to zeros.

free frees storage allocated by alloc.

atan2 atan2: inaccurate arc tangent

```
int atan2(x, y) int x, y;
```

atan2 returns the approximate arc-tangent of y/x. The return value is in integral degrees. The approximation is poor; the error may be as large as two degrees.

balloc balloc, bfree: allocate a bitmap

```
Bitmap *balloc(r) Rectangle r;
void bfree(b) Bitmap *b;
```

balloc returns a pointer to a Bitmap large enough to contain the Rectangle r, or 0 (NULL) for failure. The coordinate system inside the Bitmap is set by r: the origin and corner of the Bitmap are those of r, which must itself be in screen coordinates. There is a total of about two thirds of a screenful of off-screen bitmap memory available. bfree frees the storage associated with a Bitmap allocated by balloc.

bitblt bitblt: Rectangle copy

```
void bitblt(sb, r, db, p, f) Bitmap *sb, *db; Rectangle r;
Point p; Code f;
```

bitblt (bit-block transfer) copies the data in Rectangle r in Bitmap sb to the congruent Rectangle with origin p in Bitmap db. The nature of the copy is specified by the function code f.

button1 button[123]: button state

```
int button1(), button2(), button3();
int button12(), button23(), button123();
```

button1 and its counterparts return the state of the associated mouse button: non-zero if the button is depressed, 0 if not. The buttons are numbered left to right. button12 and the other multi-button functions return the *OR* of their states: true if either button 1 or button 2 is depressed (not button 1 *and* button 2).

circle circle, disc, arc: draw a circle

```
void circle(b, p, r, f) Bitmap *b; Point p; int r; Code f;
void disc(b, p, r, f) Bitmap *b; Point p; int r; Code f;
void arc(b, p0, p1, p2, f) Bitmap *b; Point p0, p1, p2;
Code f;
```

circle draws the best approximate circle of radius r at point p in the Bitmap b with code f. The circle is guaranteed to be symmetrical about the horizontal, vertical and diagonal axes. disc draws the corresponding disc. arc draws a circular arc centered

on p0, travelling counter-clockwise from p1 to the point on the circle closest to p2.

cos cos, sin: cosine and sine

```
int cos(d) int d;
int sin(d) int d;
```

cos and sin return scaled integer approximations to the trigonometric functions. The argument values are in degrees. The return values are scaled so that cos(0)==1024. Therefore, to calculate, for example, the mathematical expression x=x0*cos(d) to calculate a projection, the multiplication must be scaled: x=muldiv(x0, cos(d), 1024)

cursinhibit cursinhibit, cursallow: control cursor tracking

```
void cursallow(), cursinhibit();
```

cursinhibit turns off interrupt-time cursor tracking (the drawing of the cursor on the screen), although the mouse coordinates are still kept current and available in the global structure mouse. cursallow enables interrupt-time cursor tracking. cursallow and cursinhibit stack: to enable cursor tracking after two calls to cursinhibit, two calls to cursallow are required.

cursswitch cursswitch: switch cursor

```
void cursswitch(t) Texture t;
```

cursswitch changes the mouse cursor (a 16×16 pixel image) to that specified by the Texture t. If the argument is (Texture *)0, the cursor is restored to the default arrow. The cursor is global to all the processes under *mpx*, so cursswitch is best used just to indicate temporary changes of state of a running program.

display display, Drect, Jrect: globals describing display

```
Bitmap display;
Rectangle Drect, Jrect;
```

display is a global Bitmap describing the display area, in screen coordinates. Drect is a Rectangle defining, in screen coordinates, the display area available to the program. It is not necessarily display.rect because of the Art Deco border around each layer in *mpx*. Jrect is the Rectangle {0, 0, XMAX, YMAX}.

ellipse ellipse, eldisc, elarc: draw an ellipse

```
void ellipse(bp, p, a, b, f) Bitmap *bp; Point p; int a, b;
Code f;
void eldisc(bp, p, a, b, f) Bitmap *bp; Point p; int a, b;
Code f;
void elarc(bp, p0, a, b, p1, p2, f) Bitmap *bp; Point p0,
p1, p2; int a, b; Code f;
```

ellipse draws an ellipse centered at p with horizontal semi-axis a and vertical semi-axis b in Bitmap bp with code f. eldisc draws the corresponding elliptical disc. elarc draws the correspoding elliptical arc, travelling counter-clockwise from the ellipse point closest to p1 to the point closest to p2. (Beware the regrettable

difference between the calling conventions for `arc` and `elarc`.)

eqpt eqpt: compare two points for equality

```
int eqpt(p, q) Point p, q;
```

`eqpt` returns the equality of its arguments: 0 if unequal, 1 is equal. Two `Points` are equal if the corresponding coordinates are equal.

eqrect eqrect: compare two rectangles for equality

```
int eqrect(r, s) Rectangle r, s;
```

`eqrect` returns the equality of its arguments: 0 if unequal, 1 if equal. Two `Rectangles` are equal if all four corresponding coordinates are equal.

exit exit: cease execution

```
void exit();
```

`exit` terminates the process. Unlike on Unix, `exit` does not return an exit status to a parent. In either the stand-alone or *mpx* world, calling `exit` replaces the running process by the appropriate terminal program. Any associated Unix process must arrange for its own demise — `exit` is a purely local function.

infont infont, getfont, outfont, ffree: read a font from Unix

```
#include <font.h>
Font *infont(inch) int (*inch)();
Font *getfont(file) char *file;
int outfont(f, ouch) Font *f; int (*ouch)();
void ffree(f) Font *f;
```

`infont` creates a font by reading the byte-wise binary representation returned by successive calls to `inch`. It returns `(Font *)0` on error. `inch` must return successive bytes of the Unix file representation of the font, and `((int)-1)` at end-of-file. `outfont` calls the routine `ouch` to write successive bytes of the binary representation of font `f`. It returns `-1` on error, as must `ouch`. For programs running (only) under *jx*, `getfont` returns a pointer to a font read from the named `file`, essentially by calling `infont` with argument routine `getc`. It too returns `((Font *)0)` on error. `ffree` frees a font allocated by `infont` or `getfont`.

inset inset: inset a Rectangle for a border

```
Rectangle inset(r, n) Rectangle r; int n;
```

`inset` returns the Rectangle (`r.origin.x`+n, `r.origin.y`+n, `r.corner.x`-n, `r.corner.y`-n). The following code creates a clear rectangle `r` with a 2-dot wide border *inside* `r`:

```
rectf(&display, r, F_STORE);
rectf(&display, inset(r, 2), F_CLR);
```

`jcircle` jcircle, jdisc, jarc: draw scaled circle on display

```
void jcircle(p, r, f) Point p; int r; Code f;
void jdisc(p, r, f) Point p; int r; Code f;
void jarc(p0, p1, p2, f) Point p0, p1, p2; Code f;
```

`jcircle` draws in the display Bitmap the approximate circle of radius `r` centered at `p` with code `f`. `jdisc` draws the corresponding disc. `jarc` draws the circular arc centered at `p0` counterclockwise from `p1` to the point on the circle closest to `p2`. All coordinates and radii are in layer coordinates, so under *mpx*, because the layer is scaled, these routines are actually implemented by calls to the ellipse routines.

`jellipse` jellipse, jeldisc, jelarc: draw ellipse on display

```
void jellipse(p, a, b, f) Point p; int a, b; Code f;
void jeldisc(p, a, b, f) Point p; int a, b; Code f;
void jelarc(p0, a, b, p1, p2, f) Point p0, p1, p2; int a,
b; Code f;
```

`jellipse` draws in the display Bitmap an approximate ellipse centered at `p`, with horizontal semi-axis `a` and vertical semi-axis `b`. `jeldisc` draws the corresponding elliptical disc. `jelarc` draws the corresponding elliptical arc, counterclockwise from the ellipse point closest to `p1` to the ellipse point closest to `p2`. All coordinates and semi-axes are in layer coordinates.

`jinit` jinit: initialize world

```
void jinit();
```

`jinit` initializes the display and environment. It must be the first library function called in any program, even in programs that do no graphics.

`jline` jline, jlineto, jsegment: draw line on display

```
void jline(p, f) Point p; Code f;
void jlineto(p, f) Point p; Code f;
void jsegment(p, q, f) Point p, q; Code f;
```

`jline` draws a line, with function code `f`, in the display Bitmap, from the current point (initially `(0, 0)`), along the relative vector, in layer coordinates, `p`. `jlineto` draws a line from the current point to the absolute layer coordinate `p`. `jsegment` draws a line from the layer coordinate `p` to the layer coordinate `q`. `p` is in layer coordinates. All three routines leave the current point at the end of the line.

`jmove` jmove, jmoveto: move relative to current point on display

```
void jmove(p) Point p;
void jmoveto(p) Point p;
```

`jmove` moves the current point by the relative vector `p`, in layer coordinates. `jmo-`

veto sets the current point to the absolute location p, which is in layer coordinates.

jpoint jpoint: draw single pixel on display

```
void jpoint(p, f) Point p; Code f;
```

jpoint sets the pixel at location p, in layer coordinates, in the display Bitmap according to the function code f.

jrectf jrectf: rectangle function on display

```
void jrectf(r, f) Rectangle r; Code f;
```

jrectf performs the action specified by the function code f on the Rectangle r in the display Bitmap. r is in layer coordinates.

jstring jstring: draw string on display

```
Point jstring(s) char *s;
```

jstring draws, in F_XOR mode, the null-terminated string s in the display Bitmap, so that the origin of the rectangle enclosing the first character is at the current point.

jtexture jtexture: draw texture in rectangle on display

```
void jtexture(r, t, f) Rectangle r; Texture t; Code f;
```

jtexture draws with function specified by f in the rectangle r, in the display Bitmap, the texture specified by t. The texture is a 16×16 pattern of dots which is replicated to cover r. Although r is in layer coordinates, in which a unit of distance may not represent a screen dot, the texture is 16×16 screen dots.

kbdchar kbdchar: read character from keyboard

```
int kbdchar();
```

kbdchar returns the next keyboard character typed to the process. If no characters have been typed, kbdchar returns -1. If KBD has not been request'ed, kbdchar will always return -1.

menuhit menuhit: present user with menu and get selection

```
int menuhit(m, n) Menu *m; int n;
```

menuhit presents the user with a menu specified by the Menu pointer m and returns an integer indicating the selection made, or -1 for no selection. n specifies which button to use for the interaction: 1, 2 or 3. The user makes a selection by lifting the button when the cursor points at the desired selection; lifting the button outside the menu indicates no selection. The menu consists of an array of strings, terminated with a NULL pointer, set up as follows:

```
char *menutext[]={"Item 0", "Item 1", "Item 2", NULL};
Menu menu={ menutext };
```

and used as follows:

```
switch(menuhit(&menu, 1)){ /* Use left button */
    case 0:
        item_0();
        break;
    case 1:
        item_1();
        break;
    case 2:
        item_2();
        break;
    case -1:
        noselection();
        break;
}
```

mouse mouse, buttons: state of mouse

```
struct {Point xy; Point jxy; int buttons;} mouse;
```

mouse is a global location containing the current mouse coordinates and button states. xy is in screen coordinates; jxy is in layer coordinates. The buttons are encoded with the 04 bit giving the state of button 1 (non-zero implies depressed), 02 button 2 and 03 button 3. The buttons are most easily interpreted using the macros button1(), button2(), etc.

muldiv muldiv: calculate (a*b)/c accurately

```
int muldiv(a, b, c) int a, b, c;
```

muldiv is a macro that returns the 16-bit result (a*b)/c. (a*b) is calculated to 32 bits, so no precision is lost. muldiv is convenient for calculating transformations.

nap nap, sleep: relax for a while

```
void nap(nticks) int nticks;
void sleep(nticks) int nticks;
```

nap does nothing for nticks ticks of the 60 Hz internal clock. To avoid beating with the display, programs drawing rapidly changing scenes should call nap(2) between updates, to synchronize the display and memory. nap busy loops until the time is up; sleep is identical except that it gives up the processor for the interval. Unless using the mouse, a program should call sleep in preference to nap.

norm norm: return norm of three-dimensional vector

```
int norm(x, y, z) int x, y, z;
```

norm returns the norm of the vector (x, y, z).

own own: which resources have data

```
int own();
```

own returns a bit vector (see the description of request) of which I/O resources

have data available. For example, `own()&KBD` can be used to indicate if a character is available to be read by `kbdchar()`.

point point: draw a single pixel in a bitmap

```
void point(b, p, f) Bitmap *b; Point p; Code f;
```

`point` draws the pixel at location `p` in the Bitmap `b` according to function code `f`.

Pt Pt: create a point from two coordinates

```
Point Pt(x, y) int x, y;
```

`Pt` is a macro to pass a coordinate pair as a Point to a function. It only works in parameter lists.

ptinrect ptinrect: is point within a rectangle?

```
int ptinrect(p, r) Point p; Rectangle r;
```

`ptinrect` returns 1 if `p` is a point within `r`, and 0 otherwise.

raddp raddp, rsubp: arithmetic on Rectangles

```
Rectangle raddp(r, p) Rectangle r; Point p;
Rectangle rsubp(r, p) Rectangle r; Point p;
```

`raddp` returns the Rectangle `(add(r.origin, p), add(r.corner, p))`.
`rsubp` returns the Rectangle `(sub(r.origin, p), sub(r.corner, p))`.

rcvchar rcvchar: receive character from host

```
int rcvchar();
```

`rcvchar` returns the next character received from the host, typically written on the standard output of a Unix process. If there are no characters available, or RCV was not requested, `rcvchar` returns -1.

Rect Rect: create a rectangle from four coordinates

```
Rectangle Rect(a, b, c, d)    int a, b, c, d;
```

`Rect` is a macro to pass four coordinates (two coordinate pairs) as a Rectangle, to a function. It only works in parameter lists.

rectXrect rectXrect: do rectangles overlap?

```
int rectXrect(r, s) Rectangle r, s;
```

`rectXrect` returns 1 if `r` and `s` share any point; 0 otherwise.

`rectclip` rectclip: clip rectangle to another rectangle

```
int rectclip(rp, s) Rectangle *rp, s;
```

`rectclip` clips in place the Rectangle pointed to by `rp` so that it is completely contained within `s`. The return value is 1 if any part of `*rp` is within `s`. Otherwise, the return value is 0 and `*rp` is unchanged.

`rectf` rectf: perform function on rectangle in bitmap

```
void rectf(b, r, f) Bitmap *b; Rectangle r; Code f;
```

`rectf` performs the action specified by the function code `f` on the Rectangle `r` within the Bitmap `b`.

`request` request: request I/O resources

```
void request(r) int r;
```

`request` announces a program's intent to use I/O devices and resources, and is usually called once early in the program. `r` is a bit vector indicating which resources are to be used, composed by OR'ing together one or more of the elements `KBD` (keyboard), `MOUSE`, `RCV` (characters received by Blit from Unix) and `SEND` (characters sent from Blit to Unix). For example, `request(MOUSE|KBD)` indicates that the process wants to use the mouse and keyboard. If the keyboard is not requested, characters typed will be sent to the standard input of the Unix process. If the mouse is not requested, mouse events in the process's layer will be interpreted by the system rather than passed to the process. `SEND` and `CPU` (see `wait`) are always implicitly `requested`.

`Rpt` Rpt: create a rectangle from two points

```
Rectangle Rpt(p, q) Point p, q;
```

`Rpt` is a macro to pass two points as a Rectangle to a function. It only works in parameter lists.

`rol` rol, ror: rotate bits

```
int rol(x, n) int x, n;
int ror(x, n) int x, n;
```

`rol` returns x bit-rotated left by n. `ror` returns x bit-rotated right by n.

`screenswap` screenswap: swap screen rectangle and bitmap

```
void screenswap(b, r, s) Bitmap *b; Rectangle r, s;
```

`screenswap` does an in-place exchange of the screen Rectangle `s` and the Rectangle `r` within the Bitmap `b`. Its action is undefined if `r` and `s` are not congruent. `s` is *not*

clipped to the display area, only to the screen.

segment segment: draw a line segment in a bitmap

```
void segment(b, p, q, f) Bitmap *b; Point p, q; Code f;
```

segment draws a line segment in Bitmap b from Point p to q, with function code f. Like all the other graphics operations, segment clips the line so that only the portion of the line intersecting the bitmap is displayed.

sendchar sendchar, sendnchars: send a character to host

```
void sendchar(x) int x;
void sendnchars(n p) int n; char *p;
```

sendchar sends a single byte to the host, which will normally be read on the standard input of the Unix process. The characters are passed to the Unix teletype input routine, and will be processed as though they were typed on the keyboard by a user. sendnchars sends to the host n characters pointed to by p. The characters are taken literally, and are not processed by the Unix teletype input routine. A process may always send characters: these routines never block.

string string, defont: draw string in bitmap

```
#include <font.h>
Point string(ft, s, b, p, f) Font *ft; char *s; Bitmap *b;
Point p; Code f;
Font defont;
```

string draws the null-terminated string s using characters from font ft in Bitmap b at Point p, with function code f. The return value is the location of the first character *after* s; passed to another call to string, the two strings will be catenated. The characters are drawn such that the origin point of the bounding rectangle of a maximum height character lies at p. Therefore, a character drawn on the screen at (0, 0) will occupy the upper-leftmost character position on the screen. string draws characters as they are in the font. No special action is taken for control characters such as tabs or newlines. The global defont is predefined, and is the name of the standard font (not a pointer to it).

strwidth strwidth: width of character string

```
int strwidth(s) char *s;
```

strwidth returns the width, in full screen coordinates, of the null-terminated string s, interpreted in the Font defont. The height of a character string is simply defont.height.

texture texture: draw texture in rectangle in bitmap

```
void texture(b, r, t, f) Bitmap *b; Rectangle r; Texture t;
Code f;
```

texture draws with function specified by f in the rectangle r, in the Bitmap b, the texture specified by t. The array is taken to be a 16×16 pattern of dots which is repli-

cated to cover r.

wait wait: wait for resources

```
int wait(r) int r;
```

r is a bit vector composed according to the description of request above. wait suspends the process, enabling others, until at least one of the requested resources is available. The return value is a bit vector indicating which of the requested resources are available — the same as own()&r. For example, if a process wants to read from the keyboard or the host, the following fragment illustrates how to use request and wait:

```
request(KBD|RCV);
for(;;){
      r=wait(KBD|RCV);
      if(r&KBD)
            keyboard(kbdchar());
      if(r&RCV)
            receive(rcvchar());
}
```

Processes wishing to give up the processor to enable other processes to run may call wait(CPU). It will return as soon as all other active processes have had a chance to run. CPU is a fake resource which is always requested. wait(SEND) is a no-op, as is request(SEND). These are not guaranteed to remain no-ops.

muldiv: calculate	(a*b)/c accurately	muldiv
muldiv: calculate (a*b)/c	accurately	muldiv
	add, sub, mul, div: arithmetic on Points	add
	addr: Word address of Point in Bitmap	addr
addr: Word	address of Point in Bitmap	addr
	alloc, free: allocate memory	alloc
balloc, bfree:	allocate a bitmap	balloc
alloc, free:	allocate memory	alloc
circle, disc,	arc: draw a circle	circle
atan2: inaccurate	arc tangent	atan2
add, sub, mul, div:	arithmetic on Points	add
raddp, rsubp:	arithmetic on Rectangles	raddp
	atan2: inaccurate arc tangent	atan2
	balloc, bfree: allocate a bitmap	balloc
balloc,	bfree: allocate a bitmap	balloc
	bitblt: Rectangle copy	bitblt
addr: Word address of Point in	Bitmap	addr
balloc, bfree: allocate a	bitmap	balloc
Bitmap: data structure for	bitmap	Bitmap
point: draw a single pixel in a	bitmap	point
rectf: perform function on rectangle in	bitmap	rectf
screenswap: swap screen rectangle and	bitmap	screenswap
segment: draw a line segment in a	bitmap	segment
string, defont: draw string in	bitmap	string
texture: draw texture in rectangle in	bitmap	texture
	Bitmap: data structure for bitmap	Bitmap
rol, ror: rotate	bits	rol
inset: inset a Rectangle for a	border	inset
button[123]:	button state	button1
	button[123]: button state	button1
mouse,	buttons: state of mouse	mouse
muldiv:	calculate (a*b)/c accurately	muldiv
exit:	cease execution	exit
rcvchar: receive	character from host	rcvchar
kbdchar: read	character from keyboard	kbdchar
Font, Fontchar: data structure for	character set	Font
strwidth: width of	character string	strwidth
sendchar, sendnchars: send a	character to host	sendchar
circle, disc, arc: draw a	circle	circle
	circle, disc, arc: draw a circle	circle
jcircle, jdisc, jarc: draw scaled	circle on display	jcircle
rectclip:	clip rectangle to another rectangle	rectclip
eqpt:	compare two points for equality	eqpt
eqrect:	compare two rectangles for equality	eqrect
cursinhibit, cursallow:	control cursor tracking	cursinhibit
Pt: create a point from two	coordinates	Pt
Rect: create a rectangle from four	coordinates	Rect
bitblt: Rectangle	copy	bitblt
	cos, sin: cosine and sine	cos
cos, sin:	cosine and sine	cos
Pt:	create a point from two coordinates	Pt
Rect:	create a rectangle from four coordinates	Rect
Rpt:	create a rectangle from two points	Rpt
jmove, jmoveto: move relative to	current point on display	jmove
cursinhibit,	cursallow: control cursor tracking	cursinhibit
	cursinhibit, cursallow: control cursor tracking	cursinhibit
cursswitch: switch	cursor	cursswitch
cursinhibit, cursallow: control	cursor tracking	cursinhibit
	cursswitch: switch cursor	cursswitch
own: which resources have	data	own

Bitmap:	data structure for bitmap	Bitmap
Font, Fontchar:	data structure for character set	Font
Point:	data structure for position on screen	Point
Rectangle:	data structure for rectangle on screen	Rectangle
Texture:	data structure for texture	Texture
string,	defont: draw string in bitmap	string
display, Drect, Jrect: globals	describing display	display
circle,	disc, arc: draw a circle	circle
display, Drect, Jrect: globals describing	display	display
jcircle, jdisc, jarc: draw scaled circle on	display	jcircle
jellipse, jeldisc, jelarc: draw ellipse on	display	jellipse
jline, jlineto, jsegment: draw line on	display	jline
jmoveto: move relative to current point on	display jmove,	jmove
jpoint: draw single pixel on	display	jpoint
jrectf: rectangle function on	display	jrectf
jstring: draw string on	display	jstring
jtexture: draw texture in rectangle on	display	jtexture
display	display, Drect, Jrect: globals describing	display
Word: quantum of	display memory	Word
add, sub, mul,	div: arithmetic on Points	add
circle, disc, arc:	draw a circle	circle
segment:	draw a line segment in a bitmap	segment
point:	draw a single pixel in a bitmap	point
ellipse, eldisc, elarc:	draw an ellipse	ellipse
jellipse, jeldisc, jelarc:	draw ellipse on display	jellipse
jline, jlineto, jsegment:	draw line on display	jline
jcircle, jdisc, jarc:	draw scaled circle on display	jcircle
jpoint:	draw single pixel on display	jpoint
string, defont:	draw string in bitmap	string
jstring:	draw string on display	jstring
texture:	draw texture in rectangle in bitmap	texture
jtexture:	draw texture in rectangle on display	jtexture
display,	Drect, Jrect: globals describing display	display
ellipse, eldisc,	elarc: draw an ellipse	ellipse
ellipse,	eldisc, elarc: draw an ellipse	ellipse
ellipse, eldisc, elarc: draw an	ellipse	ellipse
	ellipse, eldisc, elarc: draw an ellipse	ellipse
jellipse, jeldisc, jelarc: draw	ellipse on display	jellipse
	eqpt: compare two points for equality	eqpt
	eqrect: compare two rectangles for equality	eqrect
eqpt: compare two points for	equality	eqpt
eqrect: compare two rectangles for	equality	eqrect
exit: cease	execution	exit
	exit: cease execution	exit
infont, getfont, outfont,	ffree: read a font from Unix	infont
set	Font, Fontchar: data structure for character	Font
infont, getfont, outfont, ffree: read a	font from Unix	infont
Font,	Fontchar: data structure for character set	Font
alloc,	free: allocate memory	alloc
jrectf: rectangle	function on display	jrectf
rectf: perform	function on rectangle in bitmap	rectf
infont,	getfont, outfont, ffree: read a font from Unix	infont
display, Drect, Jrect:	globals describing display	display
rcvchar: receive character from	host	rcvchar
sendchar, sendnchars: send a character to	host	sendchar
atan2:	inaccurate arc tangent	atan2
from Unix	infont, getfont, outfont, ffree: read a font	infont
jinit:	initialize world	jinit
inset:	inset a Rectangle for a border	inset

	inset: inset a Rectangle for a border	inset
request: request	I/O resources	request
jcircle, jdisc,	jarc: draw scaled circle on display	jcircle
display	jcircle, jdisc, jarc: draw scaled circle on	jcircle
jcircle,	jdisc, jarc: draw scaled circle on display	jcircle
jellipse, jeldisc,	jelarc: draw ellipse on display	jellipse
jellipse,	jeldisc, jelarc: draw ellipse on display	jellipse
display	jellipse, jeldisc, jelarc: draw ellipse on	jellipse
	jinit: initialize world	jinit
	jline, jlineto, jsegment: draw line on display	jline
jline,	jlineto, jsegment: draw line on display	jline
on display	jmove, jmoveto: move relative to current point	jmove
display jmove,	jmoveto: move relative to current point on ..	jmove
	jpoint: draw single pixel on display	jpoint
display, Drect,	Jrect: globals describing display	display
	jrectf: rectangle function on display	jrectf
jline, jlineto,	jsegment: draw line on display	jline
	jstring: draw string on display	jstring
	jtexture: draw texture in rectangle on display	jtexture
	kbdchar: read character from keyboard	kbdchar
kbdchar: read character from	keyboard	kbdchar
jline, jlineto, jsegment: draw	line on display	jline
segment: draw a	line segment in a bitmap	segment
alloc, free: allocate	memory	alloc
Word: quantum of display	memory	Word
menuhit: present user with	menu and get selection	menuhit
selection	menuhit: present user with menu and get	menuhit
mouse, buttons: state of	mouse	mouse
	mouse, buttons: state of mouse	mouse
jmove, jmoveto:	move relative to current point on display	jmove
add, sub,	mul, div: arithmetic on Points	add
	muldiv: calculate (a*b)/c accurately	muldiv
	nap, sleep: relax for a while	nap
norm: return	norm of three-dimensional vector	norm
	norm: return norm of three-dimensional vector	norm
infont, getfont,	outfont, ffree: read a font from Unix	infont
rectXrect: do rectangles	overlap?	rectXrect
	own: which resources have data	own
rectf:	perform function on rectangle in bitmap	rectf
point: draw a single	pixel in a bitmap	point
jpoint: draw single	pixel on display	jpoint
	Point: data structure for position on screen ..	Point
	point: draw a single pixel in a bitmap	point
Pt: create a	point from two coordinates	Pt
addr: Word address of	Point in Bitmap	addr
jmove, jmoveto: move relative to current	point on display	jmove
ptinrect: is	point within a rectangle?	ptinrect
add, sub, mul, div: arithmetic on	Points	add
Rpt: create a rectangle from two	points	Rpt
eqpt: compare two	points for equality	eqpt
	Pt: create a point from two coordinates	Pt
	ptinrect: is point within a rectangle?	ptinrect
Word:	quantum of display memory	Word
	raddp, rsubp: arithmetic on Rectangles	raddp
	rcvchar: receive character from host	rcvchar
infont, getfont, outfont, ffree:	read a font from Unix	infont
kbdchar:	read character from keyboard	kbdchar
rcvchar:	receive character from host	rcvchar
	Rect: create a rectangle from four coordinates	Rect

ptinrect: is point within a	rectangle?	ptinrect
rectclip: clip rectangle to another	rectangle	rectclip
screenswap: swap screen	rectangle and bitmap	screenswap
bitblt:	Rectangle copy	bitblt
screen	Rectangle: data structure for rectangle on	Rectangle
inset: inset a	Rectangle for a border	inset
Rect: create a	rectangle from four coordinates	Rect
Rpt: create a	rectangle from two points	Rpt
jrectf:	rectangle function on display	jrectf
rectf: perform function on	rectangle in bitmap	rectf
texture: draw texture in	rectangle in bitmap	texture
jtexture: draw texture in	rectangle on display	jtexture
Rectangle: data structure for	rectangle on screen	Rectangle
rectclip: clip	rectangle to another rectangle	rectclip
raddp, rsubp: arithmetic on	Rectangles	raddp
eqrect: compare two	rectangles for equality	eqrect
rectXrect: do	rectangles overlap?	rectXrect
	rectclip: clip rectangle to another rectangle	rectclip
	rectf: perform function on rectangle in bitmap	rectf
	rectXrect: do rectangles overlap?	rectXrect
jmove, jmoveto: move	relative to current point on display	jmove
nap, sleep:	relax for a while	nap
request:	request I/O resources	request
	request: request I/O resources	request
request: request I/O	resources	request
wait: wait for	resources	wait
own: which	resources have data	own
norm:	return norm of three-dimensional vector	norm
	rol, ror: rotate bits	rol
rol,	ror: rotate bits	rol
rol, ror:	rotate bits	rol
	Rpt: create a rectangle from two points	Rpt
raddp,	rsubp: arithmetic on Rectangles	raddp
jcircle, jdisc, jarc: draw	scaled circle on display	jcircle
Point: data structure for position on	screen	Point
Rectangle: data structure for rectangle on	screen	Rectangle
screenswap: swap	screen rectangle and bitmap	screenswap
	screenswap: swap screen rectangle and bitmap	screenswap
	segment: draw a line segment in a bitmap	segment
segment: draw a line	segment in a bitmap	segment
menuhit: present user with menu and get	selection	menuhit
sendchar, sendnchars:	send a character to host	sendchar
	sendchar, sendnchars: send a character to host	sendchar
sendchar,	sendnchars: send a character to host	sendchar
cos,	sin: cosine and sine	cos
cos, sin: cosine and	sine	cos
point: draw a	single pixel in a bitmap	point
jpoint: draw	single pixel on display	jpoint
nap,	sleep: relax for a while	nap
strwidth: width of character	string	strwidth
	string, defont: draw string in bitmap	string
string, defont: draw	string in bitmap	string
jstring: draw	string on display	jstring
Bitmap: data	structure for bitmap	Bitmap
Font, Fontchar: data	structure for character set	Font
Point: data	structure for position on screen	Point
Rectangle: data	structure for rectangle on screen	Rectangle
Texture: data	structure for texture	Texture
	strwidth: width of character string	strwidth

add,	sub, mul, div: arithmetic on Points	add
screenswap:	swap screen rectangle and bitmap	screenswap
cursswitch:	switch cursor	cursswitch
atan2: inaccurate arc	tangent	atan2
Texture: data structure for	texture	Texture
	Texture: data structure for texture	Texture
	texture: draw texture in rectangle in bitmap	texture
texture: draw	texture in rectangle in bitmap	texture
jtexture: draw	texture in rectangle on display	jtexture
norm: return norm of	three-dimensional vector	norm
cursinhibit, cursallow: control cursor	tracking	cursinhibit
getfont, outfont, ffree: read a font from	Unix infont, ...	infont
menuhit: present	user with menu and get selection	menuhit
norm: return norm of three-dimensional	vector	norm
wait:	wait for resources	wait
	wait: wait for resources	wait
strwidth:	width of character string	strwidth
	Word: quantum of display memory	Word

NAME

68ld − bootstrap loader for Blit

SYNOPSIS

68ld [−d] [−r] [−s] file

DESCRIPTION

68ld loads the named *file* for execution in the Blit on connected to its standard output. *file* must be a 68000 object file. *68ld* does all necessary bootstrap and protocol procedures. There are several options. **−d** causes a printout of the sizes of the text, data and bss portions of *file*. **−s** causes *68ld* to sleep indefinitely after loading, and is used primarily for loading stand-alone programs which have no standard input. This option is now obsolete. **−r** causes *file* to be relocated to an address established by a separate protocol during bootstrap. It is invoked automatically when loading a process into *mpx*(1), and usually need not be set explicitly.

The environment variable **JPATH** is the analog of the shell's **PATH** variable to define a set of directories in which to search for *file*.

EXAMPLES

Loading asteroids into *mpx :*

```
68ld asteroids
```

Reloading the terminal program on /dev/tty7:

```
68ld /usr/jerq/lib/term > /dev/tty7
```

SEE ALSO

jx(1), mpx(1)

NAME

cip – picture drawing program for Blit terminals

SYNOPSIS

cip

DESCRIPTION

Cip is an interactive drawing system for the Blit bitmap display terminal. It provides a pallette of *pic(1)* shapes (box, circle, ellipse, line, arc, spline, and text) for drawing and editing pictures on a screen. There is a macro facility for treating a collection of drawn shapes as a single entity. The principal input device is a *mouse* that provides positional information. It has three buttons that invoke drawing, editing, and command functions. The keyboard is used only for typing in text strings and filenames. The pictures are saved as *pic* descriptions, and can be included with text, tables, and equations as input to *troff*. Existing *pic* descriptions of pictures can be displayed and edited as well. *cip*.

SEE ALSO

pic(1)
TM 82-11276-1 Cip User's Manual: One Picture is Worth a Thousand Words

BEWARE

Cip relies on the host machine only for file transfers. If the host crashes while you are creating or modifying a picture, there may be no way to save the picture. The **Blit** keyboard is used only for entering filenames and text strings. Every character you type will be read by *cip*, though not necessarily when typed. Be careful about venting your frustrations by pounding on the keyboard: you may regret it later.

NAME

jc − photypesetter simulator

SYNOPSIS

jc [**−a** l w] [file]

DESCRIPTION

Jc interprets its input (standard input default) as troff output for a typesetter. Typical usage:

troff − t file | jc

At the end of each page *jc* waits for a newline (empty line) from the keyboard before continuing on to the next page. In this wait state, the command e will suppress the screen erase before the next page and sN will cause the next N pages to be skipped.

The **−a** option multiplies the default aspect ratio, 1.0, of a displayed page by *l/w*.

SEE ALSO

troff(1), plot(1)

BUGS

It is tuned for the Linotron 202; other typesetters may not be as well simulated.
The fonts are ugly.

NAME

jim − Blit text editor

SYNOPSIS

jim [files]

DESCRIPTION

Jim is the text editor for the Blit terminal. It relies heavily on the mouse for selecting text and commands. It only runs under *mpx*(1). *Jim*'s screen consists of two *textframes*, a one-line command and diagnostic textframe at the bottom and a larger multi-line file textframe above it. Except where indicated, these frames behave identically. One of the frames is always the current frame, the one to which commands and typing refer.

A textframe has at any time a selected region of text, indicated by reverse video highlighting. The selected region may be a null string between two characters, indicated by a narrow vertical bar between the characters. The editor has a single *save buffer* containing an arbitrary string. The editing commands simply invoke transformations between the selected region and the save buffer.

The mouse buttons are used for the most common operations. Button 1 (left) is used for selection. Clicking button 1 in a textframe which is not the current frame makes the indicated frame the current frame. Clicking button 1 in the current frame selects the null string closest to the mouse cursor. By pushing and holding button 1, an arbitrary contiguous visible string may be selected. Button 2 provides a small menu of text manipulation functions, described below. Button 3 holds promise for the future.

The button 2 menu entries are:

cut Copy the selected text to the save buffer and delete it from the textframe. If the selected text is null, the save buffer is unaffected.

paste Replace the selected text by the contents of the save buffer.

snarf Copy the selected text to the save buffer. If the selected text is null, the save buffer is unaffected.

Ordinary typing *of any character, including control characters,* simply replaces the selected text with the typed text. If the selected text is not null, the first character typed forces an implicit cut. The only characters which are not inserted are BS and ESC. BS is the usual backspace character, which erases the character before the selected text (which is a null string when it takes effect). ESC selects the text typed since the last button hit or ESC. If an ESC is typed immediately after a button hit or ESC, it is identical to a **cut**. ESC and **paste** provide the functionality for a simple undo feature.

The bottom line textframe is used for a few typed commands, modeled on *ed*(1). When a carriage return is typed in the bottom line, the line is interpreted as a command. The bottom line (in some sense) scrolls, but when the first character *after* a newline is to be displayed there, not when the newline appears. Thus, typically, after some message appears in the bottom line, a command need only be typed; the contents of the line will be automatically cleared when the first character of the command is typed. The commands available are:

e Edit the named file, or use the current file name if none specified.

w Write the named file, or use the current file name if none specified.

q Quit the editor.

/ Search forward for the literal string after the slash. If found, the matching text is selected.

? Search backwards for the literal string after the query. If found, the matching text is selected.

94 Select the text of line 94, as in *ed*.

The upper frame has a *scroll bar* — a black vertical bar down the left edge. A small tick in the bar indicates the relative position of the frame within the file. Pointing to the scroll bar and clicking a button controls scrolling operations in the file:

button 1
: Move the line at the top of the screen to the y position of the mouse.

button 2
: Move to the absolute position in the file indicated by the y position of the mouse.

button 3
: Move the line at the y position of the mouse to the top of the screen.

FILES

/usr/jerq/lib/jim.j terminal support program

BUGS

The program is infantile. It is still under heavy development. In particular, it needs support for regular expressions and multiple files.
Buttons 2 and 3 must be depressed with the mouse inside the current layer to be effective.

NAME

jx − Blit execution and stdio interpreter

SYNOPSIS

jx file
... | jx file

DESCRIPTION

Jx downloads the program in *file* to the Blit on /dev/tty and runs it there, simulating most of the standard I/O library functions. *stdout* and *stderr* are properly redirected, while *stdin* is only redirected properly if its input is not from the keyboard. Programs wishing to read from the keyboard should use kbdchar().

stdout and *stderr*, if directed to a controlling Blit, will be squirrelled away during execution to be cat'ed after the terminal program has been rebooted.

Programs intended for use by *jx* should include <blitio.h> and call exit() upon termination. exit() returns control to the shell and causes a reboot of the ROM terminal program.

FILES

/usr/jerq/include/jerqio.h
/usr/jerq/lib/sysint   standard I/O interpreter

BUGS

keyboard standard input doesn't work

NAME

mcc − MC68000 C compiler

SYNOPSIS

mcc [option] ... file ...

DESCRIPTION

Mcc is the C compiler for the Motorola 68000.

Mcc accepts several types of arguments:

Arguments whose names end with '.c' are taken to be C source programs; they are compiled, and each object program is left on the file whose name is that of the source with '.o' substituted for '.c'. The '.o' file is normally deleted, however, if a single C program is compiled and loaded all at one go.

In the same way, arguments whose names end with '.s' are taken to be assembly source programs and are assembled, producing a '.o' file.

The following options are interpreted by *mcc*. See *mld*(1) for load-time options.

−c Suppress the loading phase of the compilation, and force an object file to be produced even if only one program is compiled.

−j Compile the named programs, and load and link them for running stand-alone on a Blit terminal.

−m Compile the named programs, and load and link them for running under the *mpx*(1) environment on a Blit terminal.

−w Suppress warning diagnostics.

−O Invoke an (unimplemented) object-code improver.

−S Compile the named C programs, and leave the assembler-language output on corresponding files suffixed '.s'.

−E Run only the macro preprocessor on the named C programs, and send the result to the standard output.

−C prevent the macro preprocessor from eliding comments.

−o *output*
: Name the final output file *output*. If this option is used the file 'a.out' will be left undisturbed.

−D*name*=*def*
−D*name*
: Define the *name* to the preprocessor, as if by '#define'. If no definition is given, the name is defined as "1".

−U*name*
: Remove any initial definition of *name*.

−I*dir* '#include' files whose names do not begin with '/' are always sought first in the directory of the *file* argument, then in directories named in **−I** options, then in directories on a standard list.

−B*string*
: Find substitute compiler passes in the files named *string* with the suffixes cpp, ccom and c2. If *string* is empty, use a standard backup version.

−t[p012]
: Find only the designated compiler passes in the files whose names are constructed by a **−B** option. In the absence of a **−B** option, the *string* is taken to be '/usr/c/'.

Other arguments are taken to be either loader option arguments, or C-compatible object programs, typically produced by an earlier *mcc* run, or perhaps libraries of C-compatible routines. These programs, together with the results of any compilations specified, are loaded (in the order given) to produce an executable program with name **a.out**.

FILES

file.c	input file
file.o	object file
a.out	loaded output
/tmp/ctm?	temporary
/lib/cpp	preprocessor
/usr/jerq/lib/ccom	compiler
/usr/m/lib/ccom	compiler (sometimes)
/usr/jerq/lib/occom	backup compiler
/usr/jerq/lib/l.o	runtime startoff for **−j**
/usr/jerq/lib/notsolow.o	runtime startoff for **−m**
/usr/jerq/lib/libc.a	standard library
/usr/jerq/lib/libj.a	graphics library (used in **−lj**).
/usr/jerq/lib/libsys.a	system and I/O library (used in **−lj**).
/usr/jerq/include	standard directory for '#include' files

OTHER PROGRAMS

The usual array of associated object-code manipulating programs exists, with specifications identical to the usual Unix programs, and with names prefixed with an 'm.' These programs include:

mas	assembler
mlorder	order library (there is no mranlib)
mnm	name list
msize	object code size
mstrip	strip symbol table

SEE ALSO

B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978
B. W. Kernighan, *Programming in C—a tutorial*
D. M. Ritchie, *C Reference Manual*
mld(1)

DIAGNOSTICS

The diagnostics produced by C itself are intended to be self-explanatory. Occasional messages may be produced by the assembler or loader.

BUGS

The compiler currently ignores advice to put **char** and **unsigned char** variables in registers.

NAME

mld − MC68000 link editor

SYNOPSIS

mld [option] ... file ...

DESCRIPTION

Mld combines several Motorola 68000 object programs into one, resolves external references, and searches libraries. In the simplest case several object *files* are given, and *mld* combines them, producing an object module which can be either executed or become the input for a further *mld* run. (In the latter case, the **−r** option must be given to preserve the relocation bits.) The output of *mld* is left on **a.out**. This file is made executable only if no errors occurred during the load.

The argument routines are concatenated in the order specified. The entry point of the output is the beginning of the first routine.

If any argument is a library, it is searched exactly once at the point it is encountered in the argument list. Only those routines defining an unresolved external reference are loaded. If a routine from a library references another routine in the library, the referenced routine must appear after the referencing routine in the library. Thus the order of programs within libraries may be important.

The symbols 'etext', 'edata' and 'end' are reserved, and if referred to, are set to the first location above the program, the first location above initialized data, and the first location above all data respectively. It is erroneous to define these symbols.

Mld understands several options. Except for **−l**, they should appear before the file names.

−b relocate the program so its first instruction is at the absolute position indicated by the decimal *address* after the **−b** option.

−B Similar to **−b**, but only set the base address for the BSS segment. This option is usually used in conjunction with **−b** when loading programs to run from ROM.

−d Force definition of common storage even if the **−r** flag is present.

−l*x* This option is an abbreviation for the library name '/usr/lib/lib*x*.a', where *x* is a string. If that does not exist, *mld* tries '/usr/jerq/lib/lib*x*.a' A library is searched when its name is encountered, so the placement of a **−l** is significant.

−o The *name* argument after **−o** is used as the name of the *mld* output file, instead of **a.out**.

−r Generate relocation bits in the output file so that it can be the subject of another *mld* run. This flag also prevents final definitions from being given to common symbols, and suppresses the 'undefined symbol' diagnostics.

−R Similar to **−r**, but flag an error if there are undefined symbols.

−M Set the resulting a.out's magic number to 0406, to signify a binary runnable under *mpx*(1).

−v Generate a copious amount of debugging information on standard output.

FILES

/usr/jerq/lib/lib*.a	libraries
/usr/lib/lib*.a	more libraries
a.out	output file

SEE ALSO

mcc(1), ar(1)

NAME

mpx − layer multiplexor for Blit

SYNOPSIS

mpx [terminal program]

DESCRIPTION

Mpx manages asynchronous windows, or layers, on the Blit terminal. Upon invocation, it loads the Blit with a terminal program (default /usr/jerq/lib/mpxterm) that is the primary user interface.

Each layer is in most ways functionally identical to a separate terminal. Characters typed on the keyboard are sent to the standard input of the Unix process running in the layer, and characters written on the standard output appear in the layer. When a layer is created, a separate shell (sh only) is established, and bound to the layer.

Layers are created, deleted, and otherwise dealt with using the mouse on the Blit. Depressing button 3 activates a menu of layer operations. Lifting button 3 then selects an operation. At this point, the square grey cursor indicates that an operation is pending. Hitting button 3 again activates the operation on the layer pointed to by the cursor. The New operation, to create a layer, requires a rectangle to be swept out, across any diagonal, while button 3 is depressed. The box outline cursor indicates that a rectangle is to be created. The Reshape operation, to change the size and location of a layer on the screen, requires first that a layer be indicated (grey cursor) and a new rectangle be swept out (box cursor). The other operations are self-explanatory.

Button 1 is a shorthand for Current and Top, which pulls a layer to the front of the screen and makes it the current layer for keyboard input. Layers which are not current are indicated by being partially obscured with a stipple pattern.

FILES

/usr/jerq/lib/mpxterm: terminal program

SEE ALSO

68ld(1), jx(1)

DIAGNOSTICS

Error messages from *mpx* are written directly to the layer which caused them. They are usually self-explanatory only to system administrators, and indicate system difficulties.

ARCANA

A couple of keystroke sequences permit control of, or recovery from, difficult user-level processes under *mpx*. Control-Shift-BREAK resets the mouse cursor to its default state. SET-UP freezes *mpx* and complements the video of the layer of the current user-level terminal process. Hitting mouse button 2 in this state will attempt to kill the process; 1 or 3 will leave it running.

BUGS

Under 4.1bsd, not all ioctl's are supported. In particular, the C shell will not run under *mpx*.
Reshape only works properly for processes that arrange to see if they have been reshaped. Currently, only the default terminal program and *jcif* make this arrangement.

Bell Laboratories

Subject: **Overlapping Bitmap Graphics**
Case- 11173 -- File- 39199-11

date: **September 30, 1981**

from: **Rob Pike**

TM: **81-11271-10**

MEMORANDUM FOR FILE

1. Introduction

Bitmap displays are in vogue. Despite their many drawbacks, they are cheap enough to become personal displays instead of computer center resources. But before such displays become widespread, some difficult problems with drawing on them must be solved.

The peculiar properties of bitmap displays make them difficult to program for general graphics such as line drawing, and demand significant hardware or software support for even the simplest operations. They are fundamentally rectilinear devices. Diagonal lines and circles are relatively difficult to draw in a bitmap, but horizontal and vertical lines and rectangles are easy. It is therefore fortuitous that we are fond of right angles, because bitmaps are effective for a particular model of interactive graphics: possibly overlapping rectangles each containing a working environment, much like sheets of paper on a desk. The bitmap bandwagon charges off from here, attempting to convert the office of the future into a set of desks of the future, each electronically created on a display. But I am a programmer. My desk is a mess, and I rarely use paper. The most common rectangle in my world is a computer terminal's screen, and therein lies my attraction to bitmap displays. Suitably programmed, a bitmap display can become several displays at once — different areas on the screen can be emulating teletypes, graphics terminals, drafting tables or even arcade games.

The screen areas need not be disjoint — the display areas of two programs may overlap, with one area fully visible and the other partially or wholly obscured. Software to manipulate these overlapping areas, or *layers*, has been implemented on the Blit[†] display terminal. On the Blit display, a program (running locally in the terminal's microprocessor) drawing in a layer is fully isolated from other programs drawing on the same screen, even if one layer overlaps, and thereby obscures part of, the other. In particular, each program manipulates its layer just as it would manipulate the full screen if it were the only program running in the terminal. Each program in the Blit's multiprocessing graphics environment is removed from considerations regarding, for example, which layers obscure which, and two programs may (almost) simultaneously draw in their respective layers, even though some of the drawn objects are currently obscured from view.

This paper describes the primitives to support asynchronous graphics in multiple, overlapping bitmap layers, as implemented on the Blit.

[†] an MC68000-based bitmap terminal, developed by Bart Locanthi, with dual-ported display memory, but no other hardware graphics support.

Definitions

The programs described herein, written in a pseudo-C dialect and based on the Blit software, use several simple defined types and primitive bitmap operations. This section describes the basic concepts for bitmap operations, but does not attempt to be complete. The next section defines bitmaps precisely.

A Point is an ordered pair

```
typedef struct{
      int x, y;
}Point;
```

that defines a location in a bitmap such as the screen. The coordinate axes are oriented with x positive to the right and y positive *down*, with (0,0) in the upper left corner of the screen. A Rectangle is defined by a pair of Points at the upper left and lower right

```
typedef struct{
      Point origin;      /* upper left */
      Point corner;      /* lower right */
}Rectangle;
```

By definition, `corner.x >= origin.x` and `corner.y >= origin.y`. Rectangles are half-open: a Rectangle contains the horizontal and vertical lines through `origin`, and abuts, but does not contain, the lines through `corner`. Two abutting rectangles r_0 and r_1, with r_1`.origin = (`r_0`.corner.x, `r_0`.origin.y)`, therefore have no point in common. The same applies to lines in any direction: a line segment drawn from (x_0,y_0) to (x_1,y_1) does not contain (x_1,y_1). These curious definitions simplify drawing objects in pieces, which is essential to the Blit software.

The subroutine `rectf(b, r, f)` performs a function specified by an integer code `f`, in a rectangle `r`, in a bitmap `b`. The function code is one of:

F_CLR:	clear rectangle to zeros
F_OR:	set rectangle to ones
F_XOR:	invert bits in rectangle

The routine `bitblt(sb, r, db, p, f)` (bit-block transfer) copies a source Rectangle `r` in a bitmap `sb` to a corresponding Rectangle with `origin p` in a destination bitmap `db`. `bitblt()` is therefore a form of Rectangle assignment operator, and the function code `f` specifies the nature of the assignment:

F_STORE:	`source = dest`
F_OR:	`source |= dest`
F_CLR:	`source &= ~dest`
F_XOR:	`source ^= dest`

For example, `F_OR` specifies that the destination Rectangle is formed from the bit-wise OR of the source and destination Rectangles before the `bitblt()`.

`bitblt()` is a fundamental bitmap operation. It is used to draw characters, save screen rectangles and present menus. Defined more generally, it includes `rectf()`. For a thorough description, see [ing81]. `bitblt()` is also a very expensive operation, however. In the general case, the data from the source Rectangle must be shifted or rotated and masked before being written to the destination Rectangle. A Rectangle may consist of several tens of kilobytes of memory, so it is possible that a single `bitblt()` may consume a substantial fraction of a second of processor time. A display designer may therefore be tempted to add hardware assist for `bitblt()`, but as we shall see, software techniques can make the *important* cases of `bitblt()` fast enough for software alone to compete favorably with hardware-supported `bitblt()`.

Bitmaps

A bitmap is a dot-matrix representation of a rectangular image. The details of the representation depend on the display hardware, or, more specifically, on the arrangement of memory in the display. For the idea of a bitmap to mesh well with the software in the display, the screen must appear to the program as a bitmap with no special properties other than its visibility. Because images (bitmaps) are stored off-screen, off-screen memory should have the same format as the screen itself, so that copying images to and from the screen is not a special case in the software. The simplest way to achieve this generality is to make the screen a contiguous array of memory, with the last word in a scan line followed immediately by the first word of the next scan line[†]. Under this scheme, bitmaps become something familiar to the programmer: a two-dimensional array.

Few displays represent bitmaps as ordinary arrays. More commonly, memory is sliced into fixed-length bit strings (typically 1024 bits long), while only a portion (typically the low order 800 bits) of each slice is displayed. This organization wastes the non-displayed portion of the slice, for although bitmaps can be stored in the off-screen memory, the arrangement *forces off-screen bitmaps into a specific format:* they must be less than about 200 bits wide. Moreover, the off-screen memory is not contiguous, so the usual methods of storage allocation cannot be used to manage it. The problem of storage allocation becomes one of two-dimensional bin-packing!

In summary, the organization of the display hardware can have a all-important effect on the software. There are two ways to organize memory: force a display format on all memory, or force memory format (i.e. a stream of words) on the display. The former severely limits the programmer. The latter leads to simpler software, and is surprisingly easy to arrange in hardware. The rest of this discussion assumes the latter arrangement, as on the Blit.

Given a two-dimensional array in which to store the actual image, some auxiliary information is required for its interpretation. Figure 1 illustrates how a bitmap is interpreted. The hatched region is the location of the image. When a bitmap is allocated, the allocation routine, `balloc()`, assumes its data will correspond to a screen rectangle, for example, a part of one layer obscured by another. `balloc()` creates the left and right margins of the bitmap to word-align the bitmap with the screen, so word boundaries in the bitmap are at the same relative positions as in the screen. In Figure 1, the unused margin to the left of the image area in the bitmap is storage wasted to force the word-alignment. If the first bit of the image were always stored at the high bit of first word, there would only be wasted storage at the right edge of the bitmap, but copying the bitmap to the screen would require each full word in the bitmap to be rotated or shifted and masked. By avoiding the shifts, `bitblt()` (on the Blit) is roughly four times faster than if it were not word-aligned. The small price in memory is well worth the increase in speed for the common operation of saving and restoring screen rectangles. Some bitmaps, such as icons, may be copied to an arbitrary screen location, so the word-alignment does not assist them. Other than the extra space, however, no penalty is paid for the bitmap structure's generality, because such images must usually be shifted when copied to the screen, and the choice of origin bit position is, on the average, irrelevant.

`balloc()` takes one argument, the on-screen rectangle which corresponds to the bitmap image, and returns a pointer to a data structure of type Bitmap. Bitmap is defined thus:

[†]Ideally, memory could be bit addressable, but current computers are inept at bit-level manipulations. Hence for efficiency bitmaps are aligned on word boundaries in the Blit software.

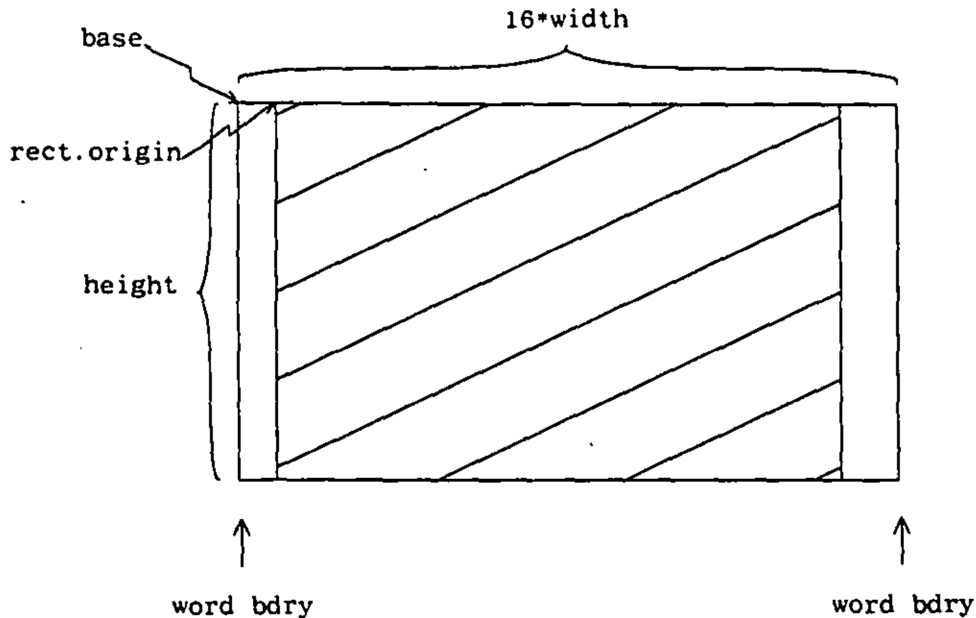


Figure 1. Layout of a bitmap. The storage begins and ends on word boundaries, with the last word of one scan line followed immediately by the first word of the next scan line. The hatched area, Bitmap.rect, stores the image.

```
typedef struct{
    Word*base;        /* start of data */
    unsigned width;   /* width in words */
    Rectangle rect;   /* image rectangle  */
}Bitmap;
```

The elements of the structure are illustrated in Figure 1. width is in Words, which are 16 bits long on the Blit. rect is the argument to balloc(), and defines the coordinate system inside the Bitmap. The storage in the Bitmap outside rect — the unhatched portion in Figure 1 — is unused, as described above.

Typically, width is the number of Words across the Bitmap, between the arrows in Figure 1. A Bitmap may be contained in another Bitmap, however, if width is the width of the outer Bitmap, and base points as usual to the first Word in the Bitmap. Although such Bitmaps are not created by balloc(), they have utility — such as representing the portion of the screen occupied by a layer.

balloc() and its obvious counterpart bfree() hide all issues of storage management for bitmaps. Initially, balloc() called the standard C allocator to acquire storage for the images. Because the sizes of bitmaps are widely variable, and because off-screen storage is only about one screenful on the Blit, balloc() was rewritten to do garbage compaction, moving all allocated bitmaps to one end of the arena when space runs low, thereby ensuring that memory does not fragment.

The Bitmap structure is used throughout the Blit software. Graphics primitives operate on points, lines and rectangles within Bitmaps, not necessarily on the screen. The screen itself is simply a globally accessible Bitmap structure, called screenmap, and is unknown within the graphics primitives.

Layers

A layer is a rectangular portion of the screen and its associated image. It may be thought of as a virtual display screen. Layers may overlap (although they need not), but the image in the obscured portion of a layer is always kept current. Typically, an asynchronous process, such as a terminal program or circuit design system, draws pictures and text in a layer, just as it might draw on a full screen if it were the only process on the display. Because processes are asynchronous, drawing actions can take place at any time in an obscured layer, and a graphical object such as a line may be partially visible on the screen and partially in the obscured portion of the layer. The layer software isolates a program drawing in an isolated region on the screen from other such programs in other regions, and guarantees that the image on- and off-screen is always correct, regardless of the configuration of the layers on the screen.

Layers are different from the common notion of windows†. Windows are used to *save* a programming or working environment, such as a text editing session, to process "interrupts" such as looking at a file or sending mail, or to keep several *static* contexts, such as file contents, on the screen. Layers are intended to *maintain* an environment, even though it may change because the associated programs are still running. For example, a programmer could be editing the program source in one layer while watching for diagnostic messages from a long compilation taking place in another layer, and waiting for intruders into his corner of the labyrinth in a real-time game in a third layer. There is, of course, nothing deep in the distinction between layers and windows; the term "layer" was coined to avoid the more cumbersome phrase "asynchronous windows." Nonetheless, the difference between layers and windows is significant. The concept of multiple active contexts is natural to use and powerful to exploit.

Truly asynchronous graphics operations are difficult to support, because the state of a layer may change while a graphics operation is underway. The obvious simple solution is to perform graphical operations atomically. This partially asynchronous strategy is used throughout the Blit software. Processes explicitly call the scheduler when they are at a suitable stopping point — there is no interruptive scheduling. Although this technique forces an extra discipline on the Blit programmer (as distinct from a user), it adds little in complexity to Blit programs and significantly simplifies the Blit run-time environment. It also avoids many potential race conditions, protocol problems, and difficulties with non-reentrant compiled code for structure-valued functions in C. For the purely single-user environment of a display terminal, such a scheme offers most of the benefits of preemptive scheduling, but with smaller, simpler, software.

The data structures for layers are illustrated in Figures 2 and 3. A partially obscured layer has an *obscured list*: a list of rectangles *in that layer* obscured by another layer or layers. In Figure 2, layer A obscures layer B. Layer B's obscured list has a single entry, which is marked "obscured by A." If more than one layer obscures a rectangle, the rectangle is marked as obscured by the *frontmost* (unobscured) layer intersecting the rectangle. This is illustrated by rectangle 4 in Figure 3. Rectangle 4 is an obscured part of both layers B and C, so these layers store their obscured pieces off-screen, and mark them blocked by layer A.

Rectangles 3 and 4 in layer B (Figure 3) may be stored as a single rectangle, as they were in Figure 2. They are stored as two because if layer C is later moved to the front of the screen (i.e. the top of the pile of layers), it will obscure portions of both layers A and B. Rectangle 4 in layer B would be obscured by C, but rectangle 3 would still be obscured by A. To simplify the algorithms for rearranging layers, the layer creation routine does all necessary subdivision when the layer is first made, so when layer C is created‡, the obscured rectangle in B is split in two along the edge of the new layer.

---

† The term "window" in common usage is a misnomer (see e.g. [new80]) The meaning is actually closer to that of the older term "viewport."

‡ Despite Figure 3, layer C is actually created at the *front*, but the argument is nonetheless true.

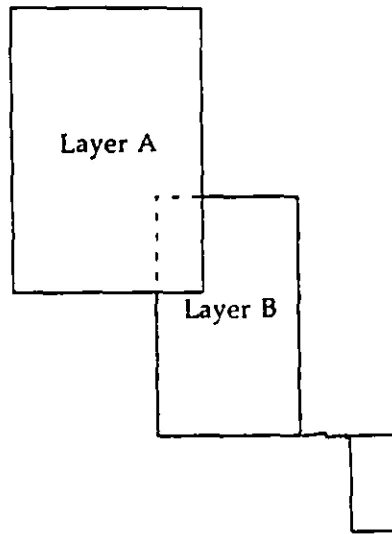


Figure 2. Obscured portions of a layer are stored off-screen and linked to the obscured layer.

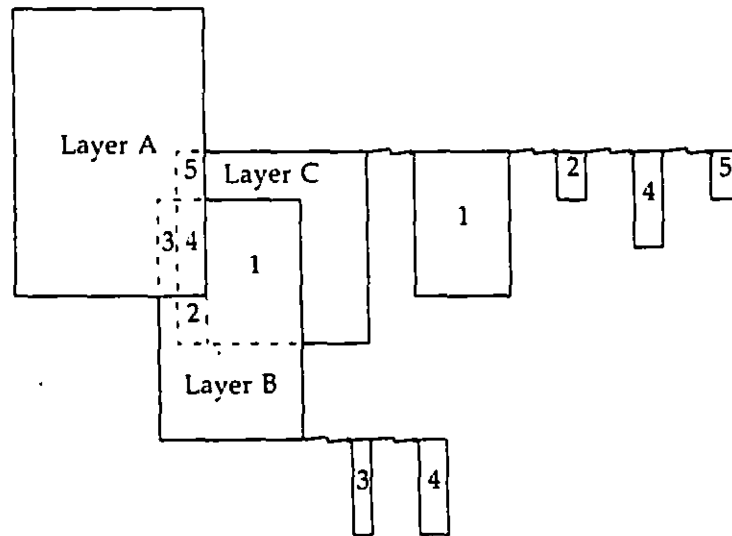


Figure 3. Rectangle 4 is obscured in layers B and C, and is maintained off-screen by both. The addition of layer C divides the obscured part of layer B into two rectangles, 3 and 4.

The layer structure is:

```
typedef struct{
    Rectangle rect;   /* bounding box of layer */
    Obscured *obs;    /* linked list of obscured rectangles */
    Layer *front;     /* adjacent layer in front */
    Layer *back;      /* adjacent layer behind */
} Layer;
typedef struct{
    Layer *lobs;      /* frontmost obscuring Layer */
    Bitmap *bmap;     /* where the obscured data resides */
    Obscured *next;   /* chaining */
    Obscured *prev;
} Obscured;
```

The individual layers are chained together as a doubly-linked list, in order from "front" to "back" on the screen (when they do not overlap, the order is irrelevant). Besides the link pointers, a layer structure contains a pointer to the list of obscured rectangles and the bounding rectangle on the screen. The obscured lists are also doubly linked, but in no particular order. Each element in the obscured list contains a Bitmap for storing the off-screen image, and a pointer to the frontmost Layer which obscures it. As we shall see later, an Obscured element need only record which (unobscured) Layer is on the screen "in front" of it, not any other obscured Layers which also share that portion of the screen. `Obscured.bmap->rect` is the screen coordinates of the obscured Rectangle. All coordinates in the layer manipulations are screen coordinates.

Layerop

`layerop()` is the main interface between layers and the graphics primitives. Given a Layer, a Rectangle within the Layer, and a bitmap operator, it recursively subdivides the Rectangle into Rectangles contained in single Bitmaps, and invokes the operator on the Rectangle/Bitmap pairs. To simplify the operators, `layerop()` also passed along, unaltered, a pointer to a set of parameters to the bitmap operator. For example, to clear a rectangle in a layer, `layerop()` is called with the target Layer, the rectangle within the layer in screen coordinates, and a procedure (the bitmap operator) to invoke `rectf()`. `layerop()` divides the rectangle into its components in obscured and visible portions of the layer, and calls the procedure to clear the component rectangles. A complete example is worked through at the end of this section. `layerop()` itself does no graphical operation, it merely *controls* graphical operations done by the bitmap operator handed to it. It turns a bitmap operator into a layer operator.

`layerop()` first clips the target Rectangle to the Layer, then calls the recursive routine `Rlayerop()` to do the subdivision:

```
/*
 * Clip to outer rectangle of layer, then call Rlayerop()
 */
layerop(lp, fn, r, otherargs)
    Layer *lp;
    void (*fn)();     /* Pointer to bitmap operator */
    Rectangle r;
    misc otherargs;  /* Other arguments used by (*fn)() */
{
    r=intersection of r and lp->rect;
    if(r not null)
        Rlayerop(lp, fn, r, otherargs, lp->obs);
}
```

`Rlayerop()` recursively chains along the obscured list of the Layer, performing the

operation on the intersection of the argument Rectangle and the obscured Bitmap, and passing non-intersecting portions on to be intersected with other Bitmaps on the obscured list. When the obscured list is empty, the rectangle must be drawn on the screen.

The code to test if two rectangles overlap is simple, but not well known:

```
rectXrect(r, s)  /* Do r and s intersect? */
    Rectangle r, s;
{
#define c corner
#define o origin
    return(r.o.x<s.c.x && s.o.x<r.c.x && r.o.y<s.c.y && s.o.y<r.c.y);
}
```

Here is Rlayerop:

```
/*
 * Rlayerop -- recursively subdivide and intersect
 *      rectangles with obscured bitmaps in layer
 */
Rlayerop(lp, fn, r, otherargs, op)
    Layer *lp;
    void (*fn)();
    Rectangle r;
    misc otherargs;
    Obscured *op;     /* Element of obscured list with which
                         to intersect r */
{
    if(op==NULL)      /* This rectangle not obscured */
        (*fn)(lp, r, &screenmap, otherargs, op);  /* Draw on screen */
    else if(rectXrect(r, op->bmap->rect)==FALSE)  /* They miss */
        Rlayerop(lp, fn, r, otherargs, op->next); /* Chain */
    else{   /* They must intersect */
        if(r.origin.x < op->bmap->rect.origin.x){
            Rectangle temp=piece of r left of op->bmap->rect;
            Rlayerop(lp, fn, temp, otherargs, op->next);
            r->origin.x=op->bmap->rect.origin.x;
        }
        /* etc. for other three sides of rectangle */
        /* What's left goes in this obscured bitmap */
        (*fn)(lp, r, op->bmap, otherargs, op);
    }
}
```

The Layer pointer and Obscured pointer are passed to the bitmap operator (`(*fn)()`) because, although they are clearly not needed for graphical operations, `layerop()`'s subdivision is useful enough to be exploited by some of the software to maintain the layers themselves; we will see an example of such usage later on.

So far, `otherargs` has been referred to in a deliberately vague manner. On the Blit, `layerop()` works something like `printf()`: after the arguments required by `layerop()` (the Layer, bitmap operator and Rectangle), the calling function passes the further arguments needed by the Bitmap operator. `layerop()` passes the address of the first of these arguments through to the operator, which therefore sees a pointer to a structure containing the necessary arguments[†].

[†] This is *not* portable, but works on the 68000 and is efficient in both code and programmer effort.

The following example illustrates the action of layerop(). Although we will see several examples later, this is the simplest and most common use. lblt() uses layerop() and bitblt() to copy an offscreen Bitmap to a Rectangle within a layer. The Bitmap may contain, for example, a character.

```
Lblt(l, r, db, fp, o)
    Layer *l;
    Rectangle r;
    Bitmap *db;         /* Destination Bitmap */
    struct{
        Bitmap *sb;     /* Source Bitmap */
        int f;          /* Function code */
    } *fp;
    Obscured *o;
{
    bitblt(fp->sb, r, db, r.origin, fp->f);
}

lblt(l, sb, r, f)
    Layer *l;
    Bitmap *sb;
    Rectangle r;
{
    layerop(l, Lblt, r, sb, f);
}
```

Notice that the bulk of the code is declarations. This code assumes the source Bitmap is in screen coordinates; if it were not, extra arguments to lblt() would be passed to the operation routine Lblt() to describe the necessary coordinate transformation.

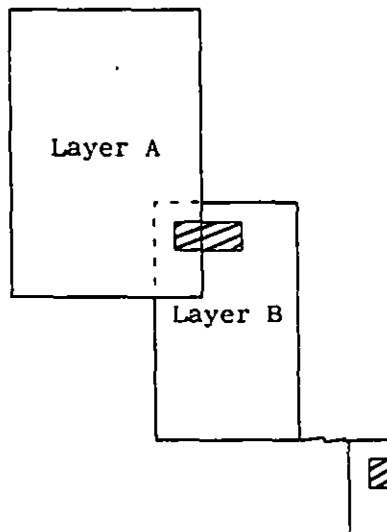


Figure 4. Drawing a rectangle (for example a character) in a layer. Some of the rectangle is drawn on the screen, some in the obscured parts of the layer. The portion of the rectangle overlapping layer A is drawn in the obscured bitmap, not on the screen.

As an example, consider drawing a character (the hashed rectangle) in Layer B in

Figure 4. The first call to Rlayerop( ) has op set to the obscured, off-screen, element of layer B. op is not NULL, nor is the overlap of r and rp->rect, so we fall into the last block of the large if-else: part of the rectangle must be drawn in this obscured bitmap. The test

```
if(r.origin.x < op->bmap->rect.origin.x)
```

succeeds, so the right portion of the rectangle (to the right of Layer A) is cut off and passed on to a second invocation of Rlayerop( ). Here, the first if succeeds, the visible portion of the rectangle is drawn on the screen, and Rlayerop returns to its first invocation. No further clipping is required, so the left portion of the rectangle is drawn in the obscured bitmap and the job is complete.

Although it is an expensive operation, layerop( ) is cheap compared to the expensive graphics operations it invokes. To be honest, however, for the commonest operation — drawing characters in the topmost, and therefore unobscured, layer — the Blit terminal program recognizes that it need not call layerop( ) and does the bitblt( ) directly to the screen, resulting in a measurable speed-up. Still, the argument holds: when layerop( ) is called, its overhead is usually unimportant. This is largely because, from the beginning, the data structures to manipulate layers were explicitly designed to make *drawing* as rapid as possible, because that is the most common operation. The less common inter-layer manipulations such as creating and deleting layers are slower, but their price is still reasonable, considering how infrequently they are invoked.

Upfront

There are three basic transformations that can in principle be applied to layers: changing the front-to-back positions of overlapping layers (stacking); changing the dimensions of a layer (scaling); and changing the position of a layer on the screen (translation).

Any stacking transformation can be defined as a sequential set of one-layer rearrangement operations, moving a single layer to another position, such as the front or back of the stack of layers. For example, the stack can be inverted by an action similar to counting through a deck of cards. upfront( ) is an operator that moves a layer to the front of the stack, making it completely visible. It is the only stacking operator in the layer software, because in the couple of places where a different operation is required, the desired effect can be achieved, with acceptable efficiency, by calls to upfront( ). The action of pulling a layer to the front was chosen because it is the most natural. When something interesting happens in a partially obscured layer, the instinctive reaction is to pull the layer to the front where it can be studied. upfront( ) also turns out to be a useful operation during the creation and deletion of layers.

Scaling and translation operators have not been written, primarily because the subdivided nature of the layer data structures makes them difficult to implement. Since their effect can be achieved with creation and deletion operators, anyway, this limitation is not considered a serious drawback. This is not to say that scaling and translation are not important transformations. Even as I was typing the first draft of this section into the computer, I discovered I had made too small a layer, and had to create a new layer, start the editor again, and delete the original. Scaling and translation services must be provided in the higher level software (they are not, at the time of writing), if not in the low-level routines.

upfront( ) has a simple structure. Most of the code is concerned with maintaining the linked lists. The basic algorithm is to exchange the obscured rectangles in the layer with those of the layer obscuring them, swapping the contents of the obscured bitmap with the screen (Figure 5). Since the obscured rectangle has the same dimensions before and after the swap, the exchange can be done in place, and it is not necessary to allocate a new bitmap, we need merely link it into the new obscured layer. Obscured rectangles are marked with the *frontmost* obscuring layer for upfront( )'s benefit: the frontmost layer is the layer that occupies the portion of the screen the rectangle would occupy were it at the front.

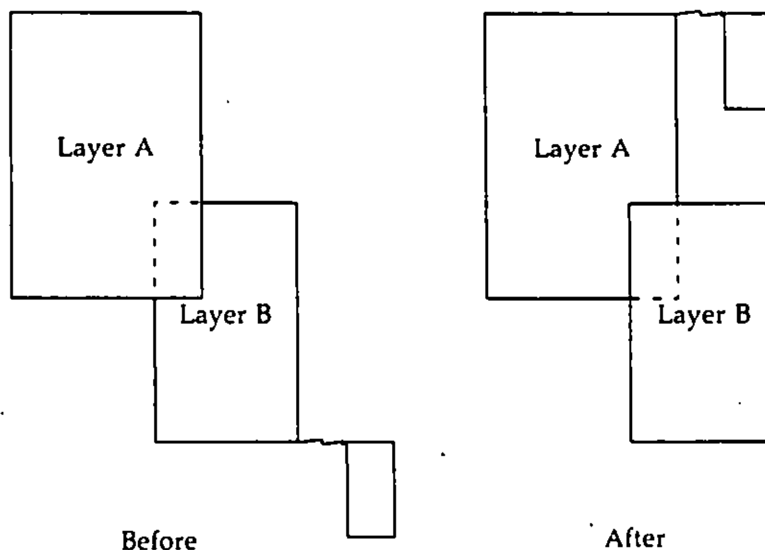


Figure 5. Moving a layer to the front requires interchanging its off-screen bitmaps with screen rectangles.

```
/*
 * upfront -- pull layer to the front of the screen
 */
upfront(lp)
    Layer *lp;
{
    Layer *fr;   /* a layer in front of lp */
    Layer *beh;  /* a layer behind lp */
    Obscured *op;

    for(fr=each layer in front of lp){
        for(op=each obscured portion of lp){
            if(op->lobs==fr){    /* fr obscures op */
                screenswap(op->bitmap, op->rect);
                unlink op from lp;
                link op into fr;
            }
        }
    }
    move lp to front of layer list;
    for(beh=all other Layers from back to front)
        for(op=each obscured portion of beh)
            if(lp->rect overlaps op->bmap->rect)
                op->lobs=lp; /* mark op obscured by lp */
}
```

screenswap() interchanges the data in the bitmap with the contents of the rectangle on the screen, in place. It is easily implemented, without auxiliary storage, using three calls to bitblt() with function code F_XOR. Note that because of the fragmentation of the obscured portions done when a new Layer is created, if lp->rect and op->bmap->rect intersect, the Layer must *completely* obscure it. Note also that it is upfront() which enforces the rule that the *frontmost* Layer obscuring a portion of a second Layer is the Layer marked as obscuring it. Only if these two Layers are interchanged is the screen updated.

The last loop is required; it is not sufficient to mark obscured rectangles only in the main loop, because rectangles initially behind lp may also need to be marked.

Dellayer

It is simpler to delete a Layer than to create one, so we will discuss deletion first. The algorithm is easy:

1) Pull the layer to the front. It now has no obscured pieces, and is a contiguous rectangle on the screen.
2) Color the screen rectangle the background color.
3) Push the layer to the back. All storage needed for the obscured portions of the layer is now bound to the layer, since it obscures no other layer.
4) Free all storage associated with the layer.
5) Unlink the layer from the layer list.

A special routine, the opposite of upfront(), could be written to push the layer to the back, but upfront() can be used for the task:

```
/*
 * dellayer -- delete a layer
 */
dellayer(lp)
     Layer *lp;
{
     Obscured *op;

     upfront(lp);
     background(lp->rect);
     /* Push to back using upfront */
     while(lp!=rearmost layer)
          upfront(rearmost layer);
     /* Free the storage */
     for(op=each obscured part of lp){
          bfree(op->bmap);
          free(op);
     }
     unlink lp from Layer list;
}
```

Using successive calls to upfront() to push a layer to the *back* gives the screen a curious appearance during the deletion. dellayer() is so simple, though, that it seems unnecessary to give it extra support by providing a more efficient "downback()" routine. upfront() does *not* join disconnected obscured bitmaps which could be joined because of the deletion, although it could. The difficulty of recognizing which bitmaps can be joined is too high to justify the small storage compaction and execution efficiency that would be gained thereby.

Newlayer

Making a new layer is the most difficult layer operation, for it may require modifying obscured lists of other layers. If the new layer creates any new overlaps, the obscured list of the overlapped layer must be restructured so that upfront() need not subdivide any rectangles to pull the obscured layer to the front. The creation routine, newlayer(), pays the price for the simplicity of upfront() and layerop().

The basic structure of newlayer() is to build the layer at the back, constructing the obscured list by intersecting the layer's rectangle with the obscured rectangles and visible

portions of the current layers. After allocating storage for the obscured bitmaps, the layer is pulled to the front, making it contiguous on the screen and forcing the rectangles obscured by the new layer to contain the new storage required by the addition of the new layer. Finally, the screen rectangle occupied by the new layer is cleared to complete the operation.

Several ancillary routines are used by `newlayer()`. `addrect()` adds rectangles to the obscured list, obs, of the new layer. Since the new layer is built at the "back" of the screen, any obscured rectangle of the new layer will be obscured by a layer already on the screen. `addrect()` builds the list of unique obscured rectangles, marked by which layer is currently occupying the screen in each rectangle. To be sure that a rectangle is unique, it is sufficient to check just the origin point of the rectangle. The rectangles passed to `addrect()` are ordered so that the first layer associated with a particular rectangle occupies the screen in that rectangle.

```
/*
 * addrect -- add (unique) rectangle to
 *      obscured list of new layer
 */
Obscured *obs;    /* Pointer to obscured list for new layer */
addrect(r, lp)
    Rectangle r;
    Layer *lp;    /* Layer currently occupying r on screen */
{
    Obscured *op, *newop;

    for(op=each element of obs)
        if(op->rect.origin == r.origin)
            return; /* Not unique */
    newop=new Obscured;
    newop->rect=r;
    newop->lobs=lp;
    link newop into obs list;
}
```

Because it is called once for each rectangle, `addrect()` takes time proportional to the square of the number of obscured rectangles in the new layer. Since `newlayer()` will ultimately perform a series of expensive `bitblt()`'s, however, the time taken by the list operations is insignificant.

`addobs()` does recursive subdivision of the obscured rectangles that intersect the new layer, calling `addrect()` when an overlap is established. It is similar to `layerop()` except that it does not chain along the obscured list, and no special action (i.e. storage allocation) is required if the rectangles match exactly. As subdivided pieces are added to the obscured list of a current layer, the original rectangle must remain in the list until all the subdivided pieces are also in the list, whereupon it is deleted. New pieces must therefore be added *after* the original piece. When the topmost call to `addobs()` returns, the subdivision (if any) is complete, and the return value is whether the argument rectangle was subdivided. `newlayer()` then removes the original rectangle from the list if `addobs()` returns TRUE.

```
/*
 * addobs -- add obscured rectangle to list, subdividing obscured
 *      portions of layers as necessary
 */
int
addobs(op, argr, newr, lp)
    Obscured *op;
    Rectangle argr;  /* Obscured rectangle */
    Rectangle newr;  /* Complete rectangle of new layer */
    Layer *lp;   /* Layer op belongs to */
{
    Obscured *newop;
    Rectangle r;
    Bitmap *bp;

    r=argr;  /* argr will be unchanged through addobs() */
    if(rectXrect(r, newr)){
        /* This is much like layerop() */
        if(r.origin.x < newr.origin.x){
            Rectangle temp=piece of r left of newr;
            addobs(op, temp, newr, lp);
            r.origin.x=newr.origin.x;
        }
        /* etc. for other three sides */
        /* r is now contained in rectangle of new layer */
        if(r == argr){   /* no clip, just bookkeeping */
            addrect(r, lp);
            return FALSE;    /* No subdivision */
        }
        addrect(r, lp);
    }
    bp=balloc(r);
    newop=new Obscured;
    /* Copy the subdivided portion of the image */
    bitblt(op->bmap, r, bp, bp->rect.origin, F_STORE);
    newop->bmap=bp;
    newop->rect=r;
    newop->lobs=lp;  /* Layer lp obscures this part of
                the new layer */
    link op into lp->obs;
    return TRUE; /* Subdivision */
}
```

newlayer( ) is long but straightforward.

```
Obscured obs;     /* obscured list of new layer when at back */
/*
 * newlayer -- make a new layer in rectangle r
 */
Layer *
newlayer(r)
        Rectangle r;
{
        Layer *lp, *newlp;
        Obscured *op;
        Bitmap *bp;

        /*
         * First build, in obs, a list of all obscured rectangles which
         * will be obscured by the new layer, doing subdivision with
         * addobs()
         */
        obs=NULL;
        for(lp=each layer from front to back){
                for(op=each obscured portion of lp){
                        if(rectXrect(r, op->rect) &&
                           addobs(op, op->rect, r, lp)){
                                unlink op from lp->obs;
                                bfree(op->bmap);
                                free(op);
                        }
                }
        }
        /*
         * Now add the rectangles not currently obscured, but that will
         * be obscured by new layer, by building layer & calling layerop()
         */
        newlp=new Layer;
        newlp->rect=r;
        newlp->obs=obs;  /* currently obscured ... */
        for(lp=each layer from front to back)
                layerop(lp, addpiece, lp->rect);
        newlp->obs=obs;  /* ... and soon to be */
        for(op=each element of obs)
                op->bmap=balloc(op->rect);
        link newlp into back of layer list;
        upfront(newlp);
        rectf(newlp->rect, F_CLR);    /* Clear the screen rectangle */
        return newlp;
}
```

addpiece() is a trivial routine to add to the obscured list the rectangles that are currently unobscured (i.e. have only one layer) but that will be obscured by the new layer.

```
addpiece(lp, r, bp, otherargs, op)
    Layer *lp;
    Rectangle r;
    Bitmap *bp;
    char *otherargs; /* Unused */
    Obscured *op;
{
    if(op==NULL) /* This piece occupied by one layer only */
        addrect(r, lp);
    /* Otherwise it's already in obs list */
}
```

Scrolling

So far, we have only discussed *support* for graphics, not graphical actions themselves. This and the next section, on scrolling and line drawing, illustrate how difficult graphics can be on a bitmap display, and how layerop() helps.

Scrolling a layer — moving a layer full of text up one line — is a special case of a bitblt operation. The general case of a bitblt in a layer is messy to implement, although not difficult — it is more a matter of bookkeeping than programming. The difficulties arise because, like a string copy routine, bitblt() must in general copy the data in a certain direction to avoid over-writing the source. With bitblt(), however, the situation is two-dimensional. Scrolling encounters the same difficulties, with the same solutions, but without the tedious detail.

Scrolling a screen rectangle r is a single bitblt:

```
#define NLSZ 16 /* height of a text line */
bitblt(&screenmap, Rect(Pt(r.origin.x, r.origin.y+NLSZ), r.corner),
    &screenmap, r.origin, F_STORE);
```

that copies all but the first line of text up one line. Rect() and Pt() create Rectangles and Points from their arguments. Scrolling a layer requires a series of bitblts, which must also be done in a certain order.

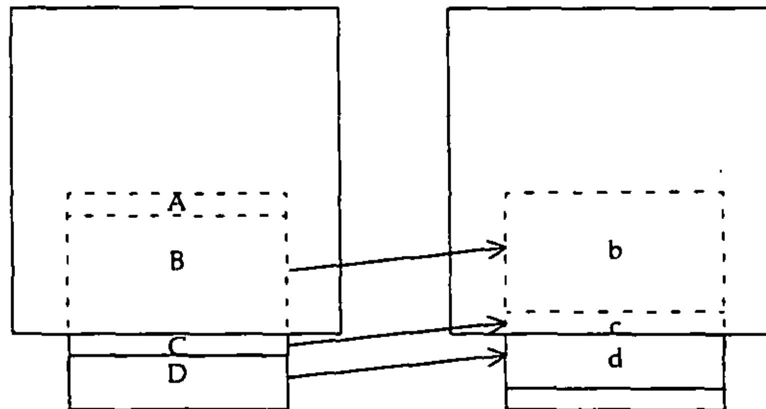


Figure 6. To scroll the rear layer, the source rectangles B through D are copied to the destination rectangles b through d.

Figure 6 illustrates a simple example of scrolling a partially obscured layer. The scroll is

broken into three bitblts, so that the source and destination rectangles each lie within a single bitmap.

Two invocations of layerop( ) do all required subdivision. The first generates a list of source rectangles by dividing the original rectangle ((B+C+D) in Figure 6, with (A+B) offscreen and (C+D) visible) into a set of rectangles contained within single bitmaps. A second pass further divides each rectangle so the destination rectangles are also contained in single bitmaps (separating C and D in Figure 6 because c is in a different bitmap from d).

lscroll( ) scrolls a layer by this method. The defined type Packet

```
typedef struct{
    Rectangle r; /* source rectangle, screen coords */
    Bitmap *sb;  /* source bitmap */
    Bitmap *db;  /* destination bitmap */
} Packet;
```

couples the source and destination bitmaps to the rectangles.

```
Rectangle source(), dest();
lscroll(lp)
    Layer *lp;
{
    Packet *p;

    layerop(lp, Pass1, source(l->rect));
    for(p=each pass 1 packet)
        layerop(lp, Pass2, dest(p->r), p->sb);
    for(p=each pass 2 packet, in increasing y order)
        bitblt(p->sb, source(p->r), p->db, p->r.origin, F_STORE);
}
```

source( ) and dest( ) convert their argument rectangles to the corresponding (one text line smaller) rectangles for bitblt( ). Pass1( ) and Pass2( ) are trivial routines to create the packet lists:

```
Pass1(lp, r, sb, otherargs, op)
    /* the usual declarations */
{
    add packet {r, sb, (Bitmap *)NULL} to pass 1 list;
}
Pass2(lp, r, db, otherargs, op)
    /* the usual declarations */
    struct{
        Bitmap *sb;
    }*otherargs;
{
    add packet {r, otherargs->sb, db} to pass 2 list;
}
```

The order of bitblts is correct because the packets are copied in increasing y order (recall that y increases *down*, and look at Figure 6). The order can either be built into the list by Pass2, or created by the for loop in lscroll( ). Again, because lscroll( ) controls expensive bitblt operations, the choice of sort technique is unimportant to efficiency.

Changing lscroll( ) to implement a general bitblt requires changing the arithmetic (the functions source( ) and dest( ) are naive) and extending the sorting method to work for any direction of copy. If the source and destination bitmaps are distinct in the uppermost call, such as in drawing a character in a layer, no sort is required. The details of

implementation are largely uninteresting.

Lines

Drawing lines on a dot-matrix display is an old problem (see e.g. [new79][†]). Drawing lines in layers is more difficult, because the sequence of dots approximating a line must be independent of the structure of the layer. To draw a line efficiently in a partially obscured layer, the line-drawing algorithm must draw the line as a set of line segments in bitmaps, but the endpoints of the segments (which must be in integer coordinates) might not be points on the actual line being approximated. A line from (0,0) to (300,100) is not approximated by the same points as two lines, one from (0,0) to (100,33) and one from (100,33) to (300,100). Of course, if efficiency is not an issue, lines may be drawn by generating the set of dots for the approximation and, for each dot, deciding in which bitmap it will be drawn. Here we will assume that a line segment may be drawn efficiently in a bitmap (which is true on any reasonable display) and that the problem is one of dividing up the original line.

Most line-drawing algorithms simulate a digital differential analyzer, or DDA, to reduce the strength of the rational arithmetic in

$$y = \frac{\Delta y}{\Delta x} x + y_0$$

to increments and decrements. Different DDA's approximate a given line by different sets of points. Bresenham's algorithm [bre65], the one used on the Blit, has no multiplications or divisions, and can be written using only integer arithmetic.

```
int e, Δx, Δy;
e=2Δy-Δx;
for(i=1; i<Δx; i++){
    drawpoint(x, y);
    if(e>0){
        y++;
        e+=2Δy-2Δx;
    }else
        e+=2Δy;
    x++;
}
```

(As mentioned earlier, this algorithm can be implemented efficiently; `drawpoint()` is not a subroutine call). Here, e is an (offset and scaled) error term that represents the vertical distance between the actual line and the approximating points. Each time the algorithm draws a point, e is adjusted to track the local deviation from the actual line. When e is positive, the actual line is too far above the dots, and the dots are therefore moved up one unit in y. We are assuming $0 < \Delta y/\Delta x \leq 1$, a shallow line of positive slope. If we further assume the real line passes through (0,0) we can analytically specify points generated by Bresenham's algorithm. The equation of the actual line is

$$y\Delta x - x\Delta y = 0,$$

so at a generated point

$$y\Delta x - x\Delta y = r$$

where r is the residual — the deviation from the actual line, measured parallel to the y axis. For computational efficiency, e is offset and scaled from r:

[†] On page 22 appears the phrase, "ideally, this computation should be done by special purpose hardware." In this section we will show that to draw a line in a layer requires an algorithm that no display has in hardware.

$$e = -2r + 2\Delta y - \Delta x \tag{1}$$

To minimize r, we round to calculate y:

$$y = \left\lfloor \frac{x 2\Delta y + \Delta x}{2\Delta x} \right\rfloor \tag{2}$$

implying, as above,

$$r = y\Delta x - x\Delta y.$$

Substituting,

$$e = (2x+2)\Delta y - (2y+1)\Delta x \tag{3}$$

Equation 3 can also be derived directly from the algorithm (and vice versa). Given equations 2 and 3, we can break a line into a set of segments in bitmaps independently of the layer's configuration by resetting the DDA for each segment. This method generates a unique approximation because the state of the DDA is a strict function of x.

One problem remains. The clipping routine (see [new79]) that divides the line into segments individually within bitmaps may clip to a horizontal or a vertical edge of a bitmap. If the edge is horizontal, several possible x values may be chosen (see Figure 7). Because lines are half-open, the desired value is the least x such that (x,y) is a point on the approximated line, as defined by (1). It can be shown that the desired value is

$$x = \left\lceil \frac{2y\Delta x - \Delta x}{2\Delta y} \right\rceil. \tag{4}$$

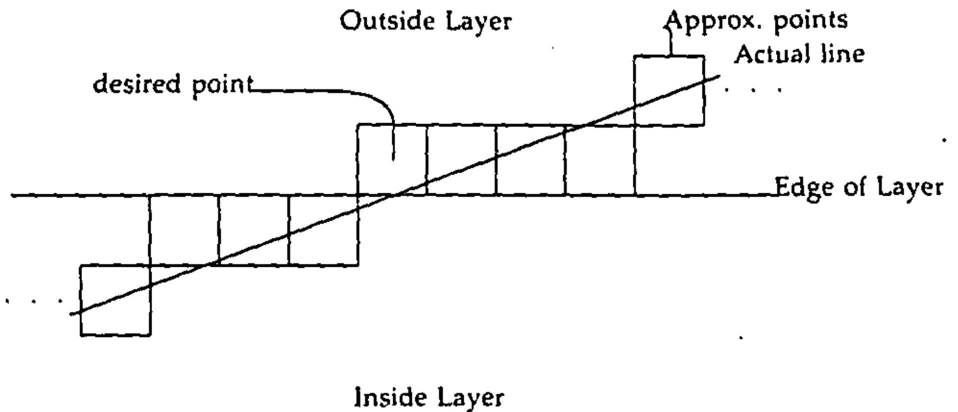


Figure 7. Lines must be clipped to the first point outside a layer or bitmap.

Turning these equations into working code involves a few obvious transformations to cover all possible cases. The transformations must be done carefully, to preserve the half-open property of lines. Given a routine `clipline()` which clips a line segment to a bitmap and draws it, using a line-drawer which loads the DDA based on the initial x value, `layerop()` can control drawing a line in a layer:

```
Lline(lp, r, db, fp, op)
    Layer *lp;
    Rectangle r;
    Bitmap *db;
    struct{
        Point p0, p1;    /* Endpoints of complete line */
        int mode;    /* Function code */
    }*fp;
    Obscured *op;
{
    clipline(db, r, fp->p0, fp->p1, fp->mode);
}
lline(lp, p, q, f)
    Layer *lp;
    Point p, q;
    int f;
{
    setline(p, q);   /* Initialize global parameters for line() */
    layerop(lp, Lline, lp->rect, p, q, f);
}
```

`setline()` initializes the global variables for the line, in particular Δx and Δy, and canonicalizes the line by coordinate transformation so that $0 < \Delta y/\Delta x \leq 1$. This line algorithm has advantages over an unadorned DDA, even for non-overlapping bitmaps: the sequence of dots generated is independent of the direction in which the line is drawn, and portions of a line may be "undrawn". Both these advantages stem from the DDA being a strict function of x, instead of the distance from the starting point of the line.

Acknowledgements

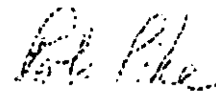
Many people helped develop the algorithms discussed herein. The data structures grew out of discussions with Ken Thompson; Doug McIlroy guided me through the mazes of integer geometry; Peter Weinberger got my line-drawing theorems straight; Andrew Koenig simplified the Bitmap data structure; and Brian Kernighan, Dave MacQueen and Chris van Wyk made helpful constructive criticism on an early manuscript, which led me to rethink and rewrite large portions of the software as well as the paper. I thank them all. I am most deeply indebted, however, to Bart Locanthi, who wrote a remarkable implementation of `bitblt()`, on which it all depends, greatly improved the interface to `layerop()`, and, most important, designed and built the Blit, thereby challenging me to tame the bitmap world.

References

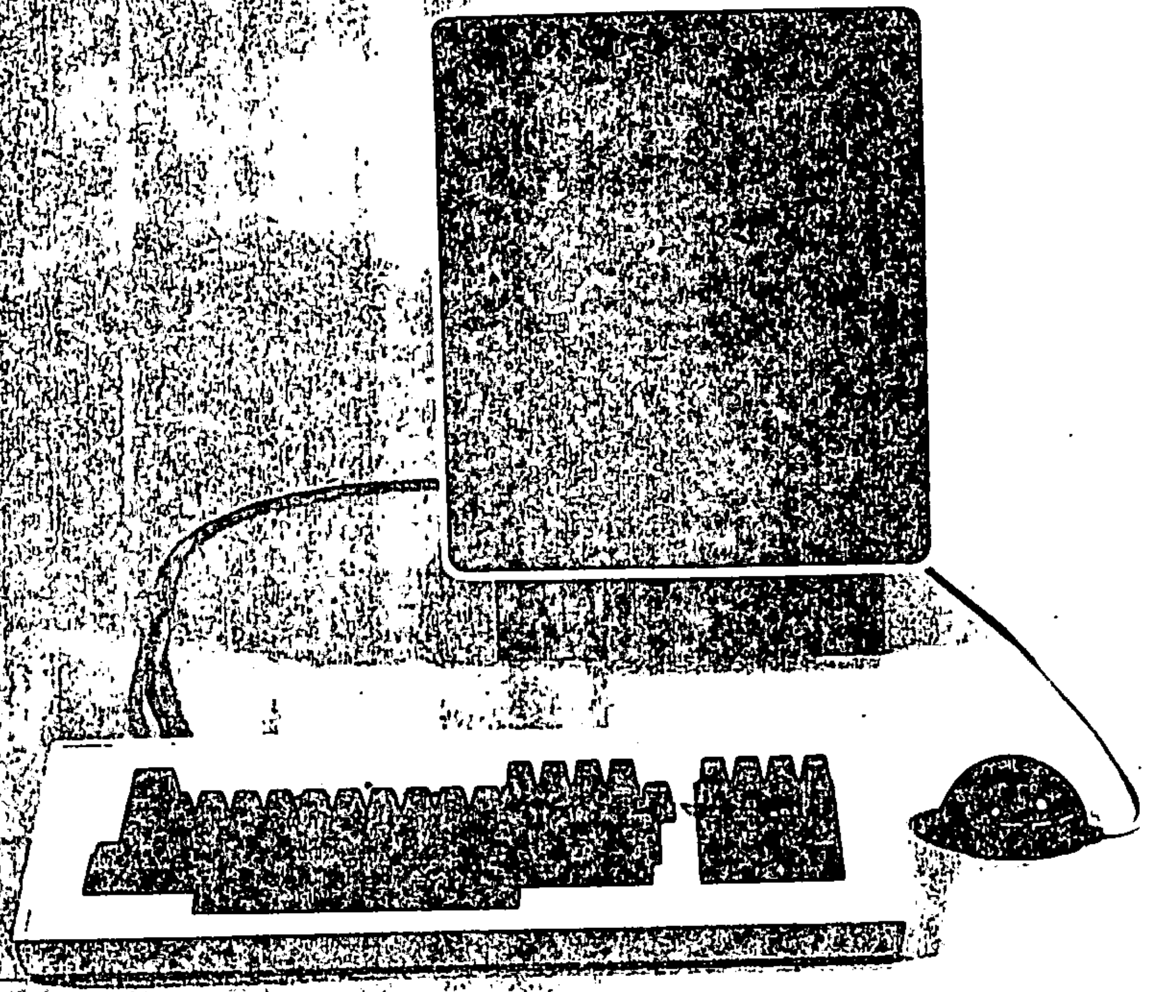
[bre65] Bresenham, J. E., "Algorithm for computer control of a digital plotter," *IBM Syst. J.*, 4 (1):25-30, 1965

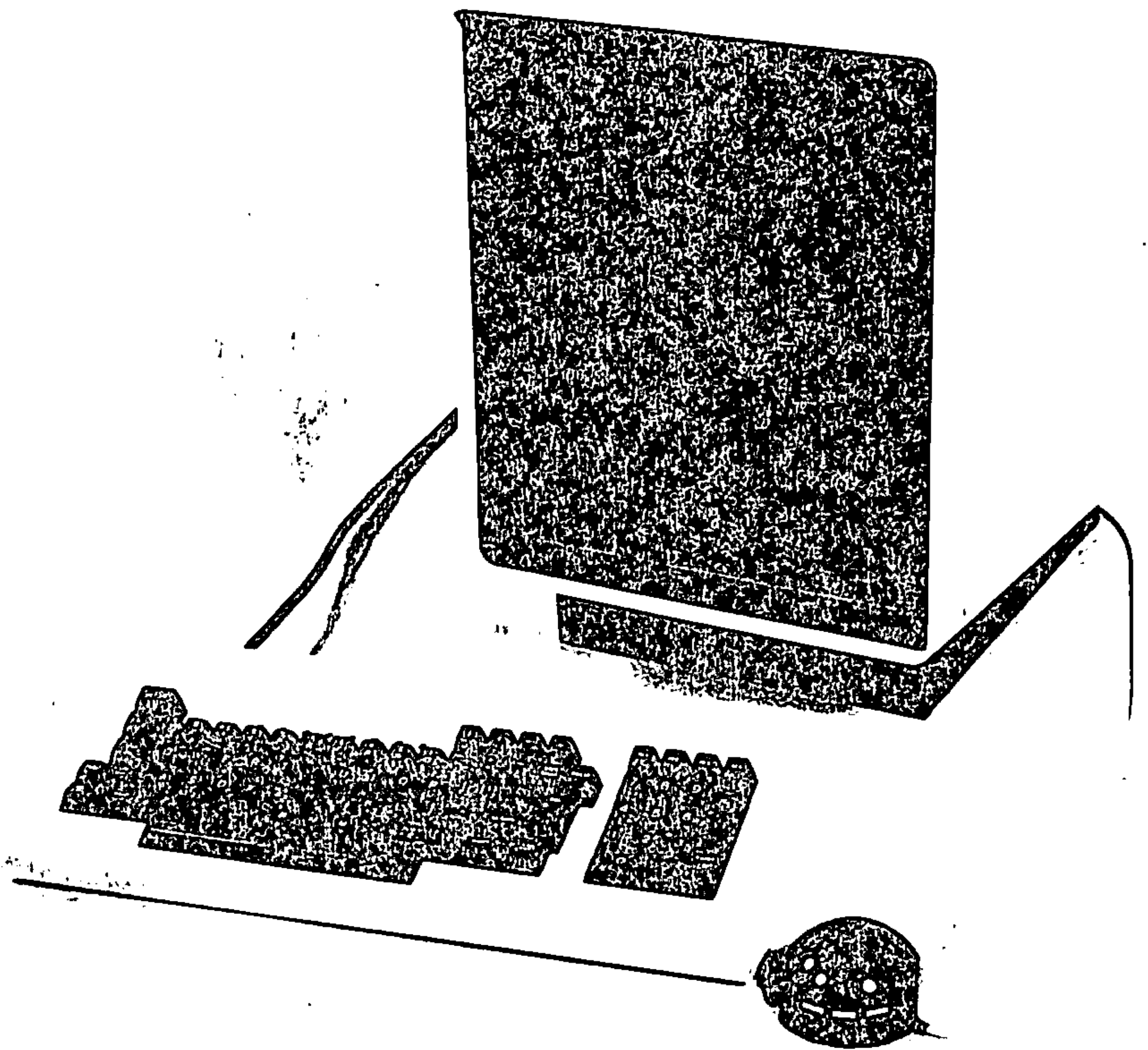
[new79] Newman, W. M., and R. F. Sproull, *Principles of Interactive Computer Graphics, 2nd Ed.*, McGraw Hill, New York, 1979

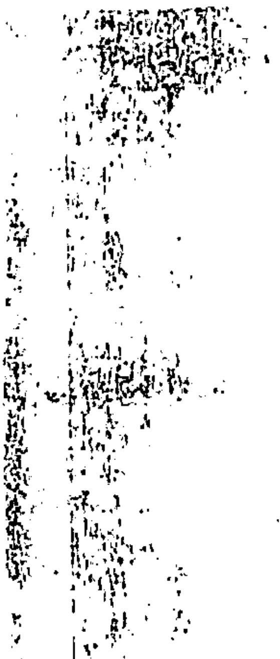
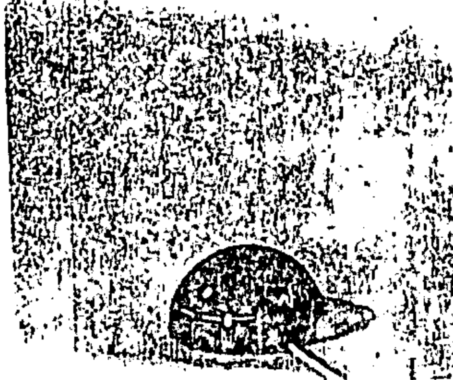
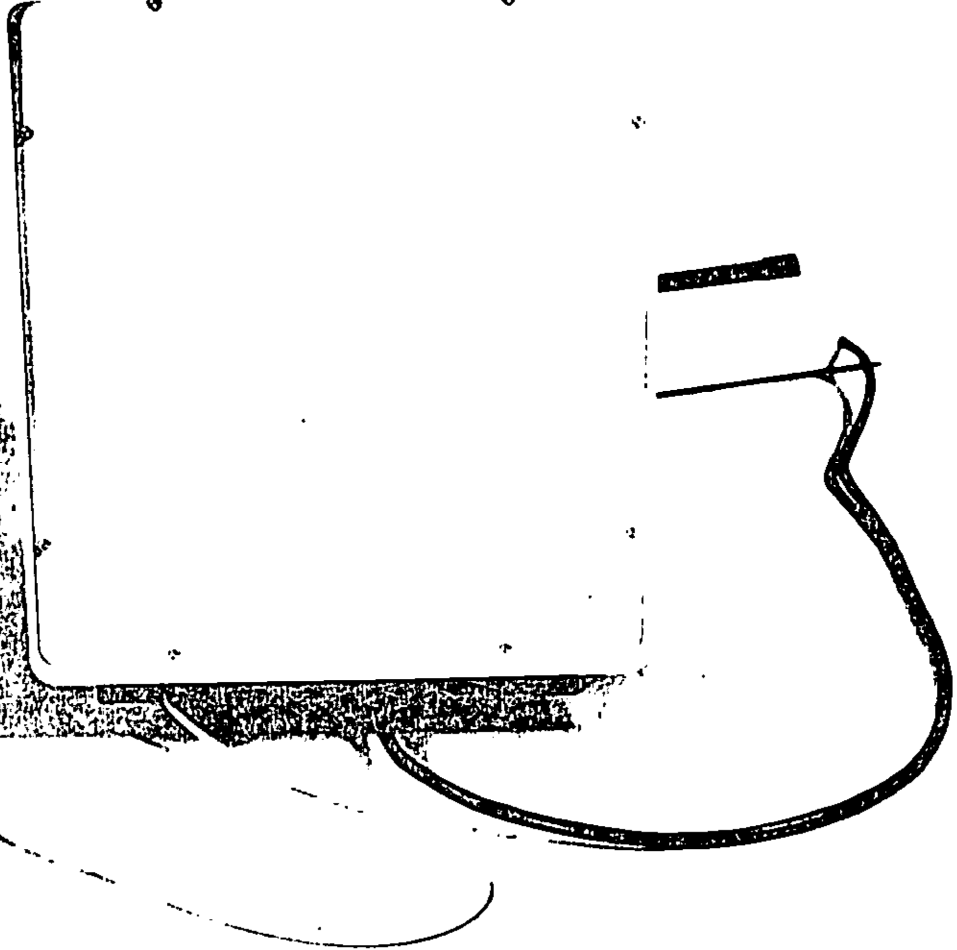
[ing81] Ingalls, D. H. H., "The Smalltalk Graphics Kernel," *Byte*, 6 (8): 168, August 1981.



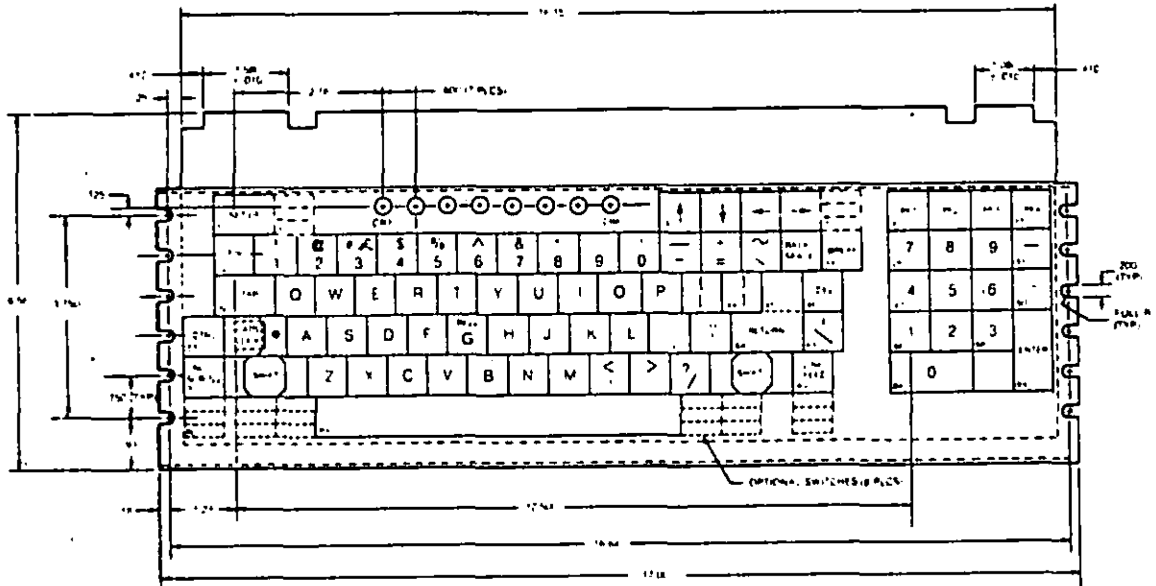
Rob Pike





KEYBOARD



ASCII CODE TABLE

Key	Unshift	Shift	Control	Control & Shift
1*	FE	FE	FE	FE
2	F0	F0	F0	F0
3	F1	F1	F1	F1
4	F2	F2	F2	F2
5	F3	F3	F3	F3
6	F4	F4	F4	F4
7	F5	F5	F5	F5
8	F6	F6	F6	F6
9	F7	F7	F7	F7
10	F8	F8	F8	F8
11	F9	F9	F9	F9
12*	ESC	ESC	ESC	ESC
13	1	!	1	!
14	2	@	NUL	NUL
15	3	#	ES	ES
16	4	$	4	$
17	5	%	5	%
18	6	^	RS	RS
19	7	&	7	&
20	8	*	8	*
21	9	(	9	(
22	0	)	0	)
23	—	—	US	US
24	=	+	=	+
25	`	~	`	~
26	BS	BS	BS	BS
27*	E0	EA	EB	EC
28	E1	E1	E1	E1
29	E2	E2	E2	E2
30	E3	E3	E3	E3
31	E4	E4	E4	E4
32*	HT	HT	HT	HT
33	q	Q	DC1	DC1
34	w	W	ETB	ETB
35	e	E	ENQ	ENQ
36	r	R	DC2	DC2
37	t	T	DC4	DC4
38	y	Y	EM	EM
39	u	U	NAK	NAK
40	i	I	HT	HT
41	o	O	SI	SI
42	p	P	DLE	DLE
43	[	{	ESC	ESC
44	]	}	GS	GS
45	D4	D6	D4	D6
46	DEL	DEL	DEL	DEL

Key	Unshift	Shift	Control	Control & Shift
47	D0	D0	D0	D0
48	D1	D1	D1	D1
49	D2	D2	D2	D2
50	D3	D3	D3	D3
51	CONTROL			
52	CAPS LOCK			
53	a	A	SOH	SOH
54	s	S	DC3	DC3
55	d	D	EOT	EOT
56	f	F	ACK	ACK
57	g	G	BEL	BEL
58	h	H	BS	BS
59	j	J	LF	LF
60	k	K	VT	VT
61	l	L	FF	FF
62	;	:	;	:
63	'	"	'	"
64*	CR	CR	CR	CR
65	\	¦	FS	FS
66	C0	C0	C0	C0
67	C1	C1	C1	C1
68	C2	C2	C2	C2
69	C3	C3	C3	C3
70*	B0	B0	B0	B0
71	SHIFT			
72	z	Z	SUB	SUB
73	x	X	CAN	CAN
74	c	C	ETX	ETX
75	v	V	SYN	SYN
76	b	B	STX	STX
77	n	N	SO	SO
78	m	M	CR	CR
79	,	<	,	<
80	.	>	.	>
81	/	?	/	?
82	SHIFT			
83	0A	0A	0A	0A
84	B1	B1	B1	B1
85	B2	B2	B2	B2
86	A0	A0	A0	A0
87	A1	A1	A1	A1
88	A2	A2	A2	A2
89	SP	SP	SP	SP
90	A3	A3	A3	A3
91	A4	A4	A4	A4
92	A5	A5	A5	A5

*These Keys do not auto-repeat
Also, note that there is no auto-repeat in the control or shift/control modes

ELECTRICAL DATA:

Input Power . . . . + 5 VDC @ 500 ma. typ.
Data Output . . . . 8 Bit ASCII (+ Logic, − Logic Optional)
Serial Output . . . . 1 Start Bit, 8 Data Bits, 1 Stop Bit (+ Logic, 300 Baud)
Parallel Data Strobe . . . . Pulsed (+ Logic)

MECHANICAL DATA:

Key Total Travel	0.171 in. (4.34 mm)
Key Actuating Force	2.0 oz (57 gr.)
Keytop Color	Ink Brown (Cycolac 82741)
Switch Reliability	100 Million MCBF

CONNECTOR DETAIL:

Serial Port:

PIN	FUNCTION
1	+ 5 VDC
2	Output
3	Ground
4	Chassis Ground
6	Input

SERIAL INPUT DATA:

Data Bit	Audible Data (B0 = 0)	LED Data (B1 = 1)
B 1	Short Beep (1)	LED 1 & 2 (complements)
B 2	Long Beep (1)	LED 3
B 3	Clicker Disable (1) or Clicker enable (0)	LED 4
B 4	*	LED 5
B 5	N/A	LED 6
B 6	N/A	LED 7
B 7	N/A	LED 8

* A "1" causes the keyboard to output data code AA (can be used for power up verification)