SCREENSPLITTER USER'S MANUAL

7900 Westpark Dr., Suite 308
McLean, Virginia 22101
(703) 827-0888

TABLE OF CONTENTS

SCREENSPLITTER PACKING LIST

1. User's Manual

2. SCREENSPLITTER PC board

3. 8' coaxial cable

4. Tube(s) 1: eight 2114's, 1 5320 sync generator (CAUTION: sensitive to static!)

5. Tube(s) 2: character generator EPROM (S, G or A) and Window Package EPROM (WP-n) (CAUTION: sensitive to static!)

6. Packet 1: Integrated Circuits:

1-74LS00	3-74LS02	2-74LS04	1-74LS10
1-74LS20	1-74LS27	1-74LS42	5-74LS74
1-74LS86	1-74LS93	2-74LS125	3-74LS157
5-74LS161	1-74166	2-74LS174	2-74LS175
4-74367	1-NE555		

TOTAL= 37

7. Packet 2: Discrete Components:

4	1.0 microfarad tantalum capacitor
5	6.8 microfarad tantalum capacitor
1	2.2 microfarad tantalum capacitor
1	470 picofarad disc capacitor
1	0.1 microfarad disc capacitor
33	0.01 microfarad disc capacitor
1	220 ohm 1/2 watt resistor
1	220 ohm 1/4 watt resistor
2	330 ohm 1/4 watt resistor
1	100K ohm 1/4 watt resistor
1	68K ohm 1/4 watt resistor
1	75 ohm 1/4 watt resistor
1	110 ohm 1/4 watt resistor
1	560 ohm 1/4 watt resistor
1	1N4742 12 volt 1 watt zener diode
2	2N2222A transistor
2	LM340T-5 voltage regulator
1	LM320T-5 voltage regulator
1	LM340T-12 voltage regulator
1	10 mhz crystal

TOTAL= 62 pieces

8. Packet 3: Hardware:

1	coax connector (three piece assembly)
1	nylon clamp
2	solder post
24	jumper posts
2	heat sink
5	6-32 x 3/8" machine screw
5	6-32 lock washer
5	6-32 hex nut
1	10' bundle solder

TOTAL= 46 pieces

9. Packet 4: 1 8-pin DIP socket, 2 24-pin DIP socket

10. Packet 5: 18 14-pin DIP socket

11. Packet 6: 20 16-pin DIP socket

12. Packet 7: 8 18-pin DIP socket

ASSEMBLY GUIDE

SCREENSPLITTER ASSEMBLY GUIDE

Introduction

First, verify that you have received a complete kit (see "Packing List"). The kit consists of:

1. The SCREENSPLITTER Owner's Manual, of which this document is a part
2. the SCREENSPLITTER printed circuit board
3. 8 feet of coaxial cable
4. 7 packets of components: (a) IC's, (b) 14-pin sockets, (c) 16-pin sockets, (d) 18-pin sockets, (e) 8-pin and 24-pin sockets, (f) discrete components, (g) hardware
5. eight 2114 RAM's, one 5320 sync generator
6. two (programmed) 2708 EPROM's marked CG (the character generator), and WP-n (the Window Package, assembled for the nth 8K block of a 65K address space), respectively

Assembly Tools

You will need the following tools to assemble SCREENSPLITTER:

1. A fine tipped soldering iron. SCREENSPLITTER is a relatively dense board, with many close traces. The best style soldering iron tip is one that is about the size of a dull writing pencil. We recommend a 27 watt UNGAR soldering iron with a #PL-340 Iron Clad tip (available from Heathkit stores, among other places, for about $7). You should not attempt to assemble the board with a soldering iron or tip that is not close to the recommended values.

2. a pair of diagonal cutters for clipping component leads after soldering

3. a pair of fine needle-nosed pliers

4. a pair of wire-strippers, or a utility knife for stripping the coaxial cable

5. a relatively small-tipped screwdriver

Soldering Technique

Solder connections should result in a small shiny mound of solder on the PC board pad. The tip of the iron should be positioned so that it contacts and heats both of the objects to be connected. It is generally best to touch the solder, which we provide in adequate quantity for each kit, only to the components being heated, and not directly to the soldering iron tip itself. Generally, you will find that connections can be made in just a couple of seconds. For the most part, you will not be soldering anything that is extremely sensitive to heat, so that several seconds of heating are acceptable. However, don't go above several seconds: it is possible to damage the discrete components (transistors, diode, capacitors), and it is possible to lift pads from the PC board with excessive heat.

Except as noted, solder connections are made only on the solder side of the PC board. Make a point of visually inspecting each joint as you advance to the next, paying special attention to the quality of the joint (its shininess and completeness of flow), and to possible solder bridges (gobs of solder that inadvertently drop or flow to other pads or traces). After each several dozen joints, rest a few moments, and reinspect all the joints you have just completed.

If you create a really bad solder bridge, obtain some "solder wick" (from, e.g., Radio Shack). This is a fine braided strand of copper that will sop up undesired solder from holes, pads and traces.

Preliminaries

Before getting started, it is recommended that you:

1. Verify that the contents of the individual component bags match the parts list at the end of this section.

2. Visually inspect the PC board, both sides, for possible anomalies in the etching. On a board of this density, quality control of the etching and drilling is essential. Although your board has already been visually inspected, you should take 5 or 10 minutes to scrutinize both sides for things such as (a) adjacent traces that may not have been completely separated by the etching process, (b) nicks on the board that may have severed a trace, (c) badly aligned holes that do not make contact with the appropriate pad. The chances are small that there are any problems, but it is wise to try to catch them at this point if any exist.

Assembly Nomenclature and Procedures

For reference, the side of the board with the large metal area and silk-screened printing indicating component numbers, etc. is called the "component side", while the reverse side the "solder side". The gold-plated fingers are the "buss fingers". Except as noted, all positions and orientations refer to the PC board positioned with its component side up, buss fingers nearest you. The "top" of the board is the edge opposite the side with the buss fingers.

All components and sockets are pushed into the board from the component side, with their leads projecting out the solder side. After soldering, all long component leads should be trimmed as close as possible.

During the assembly, consult the Assembly Guide, which shows the component layout, and the Parts List, which associates the Assembly Guide symbols with actual components and component values.

Assembly Steps

1. Mount IC Sockets

Mount the 49 IC sockets, following the table below. Although socket orientation is not important, you may want to mount all sockets so that the "clipped" interior corner of each socket is at the top right. Note that all IC's are positioned with pin 1 at the top (the square pin pad), so that by positioning all sockets with the clipped corner in this position, you will be reminded of the proper IC orientation when inserting the IC's.

Before inserting each socket, ensure that all its pins are straight. Insertion should require little force, and if you have to push hard, something is wrong. Before soldering, verify that all pins project through to the solder side, since it is virtually impossible to unsolder incorrectly mounted sockets!

We recommend mounting a column of sockets at a time. After inserting a column's worth of sockets, stretch a rubber band around the board over them to hold them in place. Flip the board over and solder two diagonally opposite pins of each socket. Remove the rubber band, then flip the board back to the component side and inspect the sockets to ensure they are all flat against the board. By reheating solder joints, adjust any that are not flat, then solder all remaining pins of each socket. Repeat until all columns of sockets have been mounted.

U1	14 pin	U17	14 pin	U33	16 pin
U2	8 pin	U18	16 pin	U34	24 pin
U3	16 pin	U19	16 pin	U35	16 pin
U4	14 pin	U20	14 pin	U36	16 pin
U5	14 pin	U21	14 pin	U37	16 pin
U6	14 pin	U22	14 pin	U38	16 pin
U7	14 pin	U23	16 pin	U39	16 pin
U8	14 pin	U24	16 pin	U40	16 pin
U9	14 pin	U25	16 pin	U41	18 pin
U10	14 pin	U26	16 pin	U42	18 pin
U11	14 pin	U27	24 pin	U43	18 pin
U12	14 pin	U28	16 pin	U44	18 pin
U13	14 pin	U29	16 pin	U45	18 pin
U14	14 pin	U30	16 pin	U46	18 pin
U15	14 pin	U31	16 pin	U47	18 pin
U16	14 pin	U32	16 pin	U48	18 pin
				J8	16 pin

2. Mount Voltage Regulators and Heat Sinks

Mount voltage regulators VR1-VR4, as shown in the Assembly Guide. The two +5 volt regulators VR1, VR2 are mounted on the large metal pad of the PC board, and are accompanied by the two heat sinks. VR3 and VR4 are mounted without heat sinks.

VR1,VR2	+5 volt	LM340T-5 or 7805
VR3	-5 volt	LM320T-5 or 7905
VR4	+12 volt	LM340T-12 or 7812

For each regulator, first position it so that its hole coincides with the PC board hole, and gauge where to bend the three leads for insertion into the three PC board holes. Bend the leads appropriately, then fasten the regulator to the board: push a bolt through, then, on the solder side, slip a lock washer over the bolt and screw down the assembly with a nut. In the cases of VR1 and VR2, the heat sink should be mounted directly under the regulator, between it and the large metal area on the component side of the PC board. Note that the heat sinks have a slight notch on the side from which the regulator projects.

After bolting down the 4 regulators, solder their leads to the pads on both the component and solder sides, then clip excess leads.

3. Mount Discrete Components

a. Mount R1 and Z1. R1 is the larger of the two 220 ohm (red-red-~~red~~ BRN) resistors; its orientation is not important. Z1 is the zener diode, about the size of a resistor, but with a black band at one end rather than a color code. Z1's orientation is important; mount it with the black band to the left (on the "bar" side of the "bar-arrow" symbol on the silk screen).

b. Mount the 100 microfarad electrolytic filter capacitor, C1; its orientation is important. Mount it so that its positive lead coincides with the + symbol on the Assembly Guide and silk screen.

c. Mount tantalum filter capacitors C2-C11, as listed below. These are the green, or blue-green-gray tear-drop shaped capacitors, marked either with numeric values or color codes. (The blue-green-gray ones

are 6.8 microfarads.) Orientation on all these is important: the positive lead is either marked as such, or is on the side with the gap in the gray band of the 6.8 microfarad capacitors. Insert all of these with the positive lead oriented as shown on the Assembly Guide and silk screen.

C2 6.8 microfarad tantalum (as marked, or blue-gray-green)
C3 6.8 microfarad tantalum
C4 1.0 microfarad tantalum
C5 6.8 microfarad tantalum
C6 1.0 microfarad tantalum
C7 6.8 microfarad tantalum
C8 1.0 microfarad tantalum
C9 6.8 microfarad tantalum
C10 1.0 microfarad tantalum
C11 2.2 microfarad tantalum

d. Mount disc ceramic capacitors C12 and C13. These are the disc-shaped capacitors with values marked numerically; their orientation is unimportant.

C12 470 picofarad disc
C13 0.1 microfarad disc

e. Mount the 33 0.01 microfarad disc ceramic capacitors C14-C46 at the various locations around the board shown on the silk screen and Assembly Guide (orientation is unimportant). You may wish to omit C45 to accommodate one of the coaxial cable termination posts as described in step 4.

f. Mount the 10 mhz crystal XTAL1 flat against the board, taking care to keep its leads clear of the underlying traces on the component side.

g. Mount resistors R2-R9 (orientation is unimportant):

R2 330 ohm (orange-orange-brown)
R3 330 ohm (orange-orange-brown)
R4 100K ohm (brown-black-yellow)
R5 68K ohm (blue-gray-orange)
R6 75 ohm (purple-green-black)
R7 220 ohm (red-red-brown)
R8 560 ohm (green-blue-brown)
R9 110 ohm (brown-brown-brown)

h. Mount the two 2N2222A transistors Q1, Q2. Insert them with their flat faces toward the buss fingers, bending the center lead slightly to

match the triangularly spaced holes on the PC board. Take care not to apply more than several seconds of heat to their leads when soldering.

4. Attach Coaxial Cable

(Note: The PC board has been designed for mounting the coaxial cable at the top left corner, leaving the top edge of the board. If physical constraints in your application preclude this mounting style, take them into account when following this step!)

The coaxial cable attaches to two pads at the top left corner of the board. The square pad with the large hole accepts the twisted outer braiding of the coax, while the round pad with the smaller hole accepts the inner solid wire. Although you may solder directly to the pads, it is recommended that you use the two posts provided, so that if you need to remove the cable for some reason in the future, you will not have to unsolder directly from the pads (running the risk of damaging them). If you choose to use the posts, insert one in the round pad for the inner coax wire, and use the top hole of C45 (ground) for the connection to the braided outer shield wire of the coax. In this configuration, you will not use the square ground pad with the large hole. If you use the posts, push them into the holes firmly, solder them on both sides of the board for strength, then clip off the excess on the solder side.

Strip off about an inch of the coaxial cable's outer jacket with a utility knife, taking care not to cut through the braided shield wire directly underneath. We recommend first making a 1" longitudinal slit, then using diagonal cutters around the perimeter to remove the jacket. After removing it, unbraid the outer shield and twist it into a single strand.

The cable will be held down by the nylon clamp, mounted with a bolt through the hole at the top left of the board. To gauge where the inner conductor's insulation should be stripped, slip the nylon clip over the cable and position it over the hole. Strip off the necessary amount of insulation from the inner conductor of the cable, then fasten the cable and clamp into their final positions, using a bolt and a lock washer-nut combination on the back of the PC board. (The nylon clamp fits the cable precisely. If the cable

is at all loose in the clamp, wrap one or two turns of black plastic electrical tape around it to provide a snug fit.) Solder the twisted (formerly braided) lead to either the square ground pad or the post in the upper hole of C45, and the inner solid lead to the circular pad, or the post in it. Clip any excess leads. Finally, inspect the outer braiding of the coax to ensure that it is not touching the PC board traces directly underneath.

After attaching the cable to the PC board, attach the connector to the other end of the cable. Make sure that the type provided fits your TV monitor, and substitute if necessary. If you use the standard connector provided, follow the following mounting instructions.

There are three pieces to the connector provided: the inner adaptor sleeve, the male plug, and the outer sleeve. Unscrew and remove both the outer sleeve and the inner adaptor sleeve from the male plug. Slide the inner adaptor sleeve onto the cable, large end first; slide the outer sleeve onto the cable, small end first. Strip off about an inch of the cable's outer jacket, using the technique described above. Unbraid and twist the outer shielding into a single strand. Strip off all but about 3/8 inch of the inner cable conductor's insulation, then slip the male plug onto the end of the cable. The inner conductor should protrude out the top of the inner hollow post; the twisted braiding should protrude through one of the holes midway up the body of the male plug. Ensure that the inner insulation presses firmly up to the base of the hollow inner post inside the unit.

Holding the cable in firmly, screw in the inner adaptor sleeve tightly. It should clamp down on the twisted braiding inside the unit, providing the ground connection. Clip off any excess twisted braiding protruding out of the hole. Finally, solder the inner conductor to the tip of the hollow post (using a minimum of solder), clip the conductor flush with the end of the post, and screw the outer sleeve back on the completed assembly. You can check the soundness of the connections by testing for continuity with an ohmmeter. (Resistance across the center post of the coax plug and its outer grounded jacket will be about 75 ohms.)

5. Install Jumpers

Install the eight jumpers, locations noted on the Assembly Guide. Each

jumper consists of three holes. Insert and solder one jumper post into each of the 24 holes. Use some of the clipped off component leads or bell wire for connecting the jumpers.

Connect the jumpers as follows:

a. J1 determines whether or not a wait state will be introduced on memory read and memory write operations to the display RAM's and software EPROM. Though many 8080-based systems will not require a wait state, we recommend introducing one, simply to keep your board easily interchangeable. (The real-time slowdown is not noticeable.) All 4 mhz applications will require a wait state. To introduce a wait state, connect terminals 1 and 2 of J1. To run without a wait state, connect terminals 2 and 3 of J1.

b. J2,J3,J4 determine the 8K block of your system's address space into which SCREENSPLITTER is mapped. You should map the hardware into the same 8K block for which your Window Package software has been assembled. The following table specifies the connections to be made for J2, J3 and J4 for any 8K block.

Base Address	J2	J3	J4
0K (WP-i)	1-2	2-3	2-3
8K (WP-1)	1-2	1-2	2-3
16K (WP-2)	1-2	2-3	1-2
24K (WP-3)	1-2	1-2	1-2
32K (WP-4)	2-3	2-3	2-3
40K (WP-5)	2-3	1-2	2-3
48K (WP-6)	2-3	2-3	1-2
56K (WP-7)	2-3	1-2	1-2

c. J5 affects the address decoding associated with the Window Package EPROM. For the 1K byte prom provided, connect terminals 2 and 3 of J5. If you upgrade to a 2K byte 2716 EPROM in the future, switch this so that terminals 1 and 2 of J5 are connected.

d. J6, J7 determine the power supply connections to the Window Package EPROM. For the 2708 provided, connect terminals 2 and 3 of J6, and connect terminals 1 and 2 of J7. As shown on the schematic, page 3, these two jumpers will allow you to upgrade to a 2716 by switching J6 to connect terminals 1 and 2, and switching J7 to connect terminals 2 and 3.

e. J9 governs whether or not memory reads from SCREENSPLITTER's display buffer and EPROM will wait for the S-100 buss PDBIN signal, or whether they will begin immediately at SMEMR time. (Strictly speaking, an S-100 buss memory read operation occurs when SMEMR and PDBIN are both on. However, since it is safe to equate memory reads with SMEMR, many boards do this to give the memory more time to respond.) If you ignore PDBIN, you will probably be able to run on any 2 mhz machine without a wait state. If you include PDBIN, you may need a wait state, since the read window is then shortened. To ignore PDBIN, connect pins 1 and 2 of J9; to include PDBIN, connect pins 2 and 3 of J9.

A "universal" SCREENSPLITTER board, and the one we recommend, has the following combination of jumpers:

J1	1-2	(introduce wait states)
J2,J3,J4	...	(board address, up to user)
J5	2-3	(1K Window Package EPROM)
J6	2-3	(1K Window Package EPROM)
J7	1-2	(1K Window Package EPROM)
J9	2-3	(include PDBIN on read cycles)

6. Inspect, Reflect, Smoke-test

At this point, you have completed the mechanical assembly of the board. Reinspect the entire board a final time, looking for solder bridges, nicked traces, etc. When confident that all is well, subject the board to The Smoke Test before inserting the IC's. This test is simple: plug SCREENSPLITTER into your system to verify that there are no power supply problems. We recommend that you test all supply voltages with a meter or scope at this point, paying special attention to the power supply pins of the expensive chips: the 2114 RAM's, the 2708 EPROM's, and the 5320 sync generator. We also recommend performing The Smoke Test on a reduced system (i.e., remove as much of your system from the buss as possible) to minimize risk.

7. Insert IC's

After successfully completing The Smoke Test, turn the system off, remove the board, and insert all IC's. IC insertion should be done with the board

lying on a flat surface so that insertion pressure does not crack the board or its traces. All IC's are inserted with pin 1 at the top right. Note that the character generator EPROM is marked either S for scientific symbols, G for graphics symbols, or A for the APL character set, and that the Window Package is marked WP-n, where n (0,...,7) indicates the 8K block of memory for which the Window Package software has been assembled. The character generator is inserted as IC U34, the Window Package as IC U27. Pin 1 of an IC is identified either by a small indentation on the IC's top surface next to it, or by a notch in one end of the IC: when the IC is positioned pins-down, notch at the left, pin 1 is at the lower left corner. NOTE: The 5320 sync generator is extremely sensitive to static discharges at its pins. Be sure you are in a relatively high-humidity environment when you handle this chip, and keep finger contact with its pins at a minimum. Some manufacturers recommend grounding your fingers to earth (through a minimum resistance of 200K ohms to eliminate shock hazards) via a wire wrapped around them while handling such sensitive devices! (See Popular Electronics, August 1977, "How to Handle MOS Devices without Destroying Them" by Leslie Solomon.)

As you insert each IC, verify that all pins have entered the socket properly; one common class of problems stems from bent IC pins that do not correctly enter the socket.

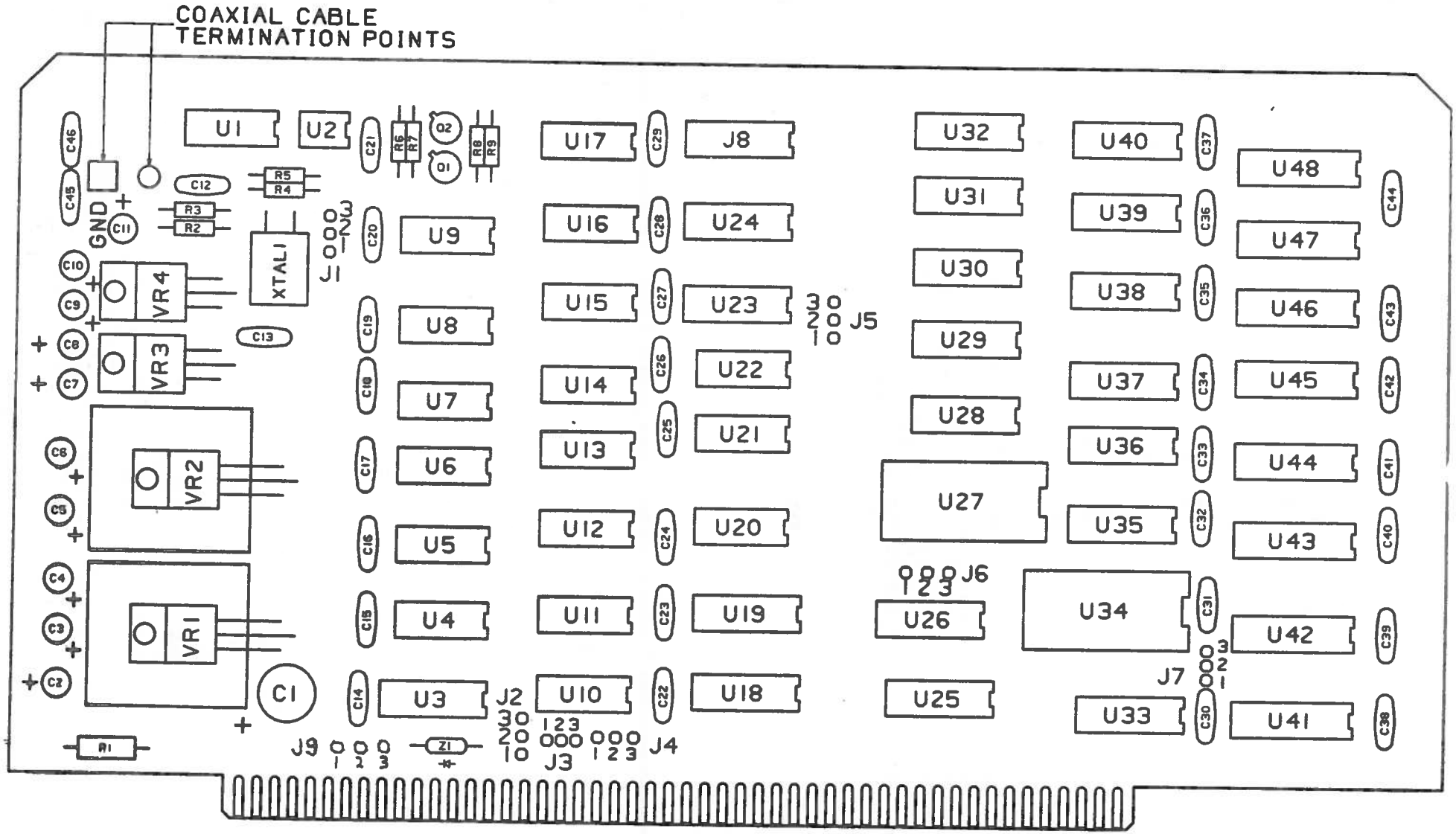
8. Plug It In

Kit assembly is complete at this point. Reinsert the board into the reduced system, connect the coaxial cable to a 10 mhz or better monitor (P39 or other high-persistance phosphor gives the best results), and turn the system on. After suitable adjustments to the monitor, you should see a screenful of random characters. If you do, chances are good that you have a completely functioning system. If you do not, refer to the section of the User's Manual entitled "Troubleshooting". If all seems well, proceed to the section entitled "Software Integration and Checkout".

SCREENSPLITTER PARTS LIST AND SYMBOL DEFINITIONS

Symbol	Part	Description
B1		SCREENSPLITTER PC board
U1	74LS04	hex inverter
U2	NE555	timer
U3	MM5320	sync generator
U4	74LS74	dual D-type flip-flop
U5	74LS74	dual D-type flip-flop
U6	74LS74	dual D-type flip-flop
U7	74LS93	4 bit ripple counter
U8	74LS74	dual D-type flip-flop
U9	74LS02	quad 2-input NOR
U10	74LS04	hex inverter
U11	74LS20	dual 4-input NAND
U12	74LS02	quad 2-input NOR
U13	74LS02	quad 2-input NOR
U14	74LS27	triple 3-input NOR
U15	74LS10	triple 3-input NAND
U16	74LS86	quad exclusive-OR
U17	74LS74	dual D-type flip-flop
U18	74367	hex tri-state driver
U19	74367	hex tri-state driver
U20	74LS125	quad tri-state buffer
U21	74LS125	quad tri-state buffer
U22	74LS00	quad 2-input NAND
U23	74LS161	4 bit synchronous counter
U24	74LS161	4 bit synchronous counter
U25	74367	hex tri-state driver
U26	74367	hex tri-state driver
U27	2708	1K UV EPROM
U28	74LS161	4 bit synchronous counter
U29	74LS174	hex D-type latch
U30	74LS161	4 bit synchronous counter
U31	74LS174	hex D-type latch
U32	74LS161	4 bit synchronous counter
U33	74166	8-bit shift register
U34	2708	1K UV EPROM
U35	74LS175	quad D-type latch
U36	74LS175	quad D-type latch
U37	74LS42	4-line to 10-line decoder
U38	74LS157	quad 2-line to 1-line selector
U39	74LS157	quad 2-line to 1-line selector
U40	74LS157	quad 2-line to 1-line selector
U41	2114	4K static RAM
U42	2114	4K static RAM
U43	2114	4K static RAM
U44	2114	4K static RAM
U45	2114	4K static RAM
U46	2114	4K static RAM
U47	2114	4K static RAM
U48	2114	4K static RAM
S1		8-pin socket
S2-S19		14-pin socket
S20-S39		16-pin socket
S40-S47		18-pin socket
S48-S49		24-pin socket
VR-1	LM340T-5	+5 volt voltage regulator (7805)
VR-2	LM340T-5	+5 volt voltage regulator (7805)
VR-3	LM320T-5	-5 volt voltage regulator (7905)
VR-4	LM340T-12	+12 volt voltage regulator (7812)
R1	220 ohm	1/2 watt resistor
R2	330 ohm	1/4 watt resistor
R3	330 ohm	1/4 watt resistor
R4	100K ohm	1/4 watt resistor
R5	68K ohm	1/4 watt resistor

R6	75 ohm	1/4 watt resistor
R7	220 ohm	1/4 watt resistor
R8	560 ohm	1/4 watt resistor
R9	110 ohm	1/4 watt resistor
C1	100 mf	electrolytic capacitor
C2	6.8 mf	tantalum capacitor
C3	6.8 mf	tantalum capacitor
C4	1.0 mf	tantalum capacitor
C5	6.8 mf	tantalum capacitor
C6	1.0 mf	tantalum capacitor
C7	6.8 mf	tantalum capacitor
C8	1.0 mf	tantalum capacitor
C9	6.8 mf	tantalum capacitor
C10	1.0 mf	tantalum capacitor
C11	2.2 mf	tantalum capacitor
C12	470 pf	disc capacitor
C13	0.1 mf	disc capacitor
C14-C46	0.01 mf	disc capacitor
Z1	1N4742	12 v. 1 watt zener diode
Q1	2N2222A	transistor
Q2	2N2222A	transistor
XTAL1	10mhz	crystal
H1,H2	291-.36H	heat sink
M1-M5	6-32x3/8"	nut-bolt-lock washers
CA1	RG-59U	8' coaxial cable
CC1	PL-259	coaxial connector with adaptor
CL1		nylon cable clamp
SP1,SP2		solder post
JP1-JP24		jumper post


COAXIAL CABLE
TERMINATION POINTS
U1
U2
U3
U4
U5
U6
U7
U8
U9
U10
U11
U12
U13
U14
U15
U16
U17
U18
U19
U20
U21
U22
U23
U24
U25
U26
U27
U28
U29
U30
U31
U32
U33
U34
U35
U36
U37
U38
U39
U40
U41
U42
U43
U44
U45
U46
U47
U48
J8
J5
J6
J7
J2
J3
J4
J9
J1
VR1
VR2
VR3
VR4
XTALI
C1
GND

ASSEMBLY

HARDWARE INTEGRATION

SCREENSPLITTER HARDWARE INTEGRATION

Hardware integration of SCREENSPLITTER consists of:

1. Deciding on which 8K memory boundary you should locate the system.

As discussed in step 5b of the Assembly Guide, jumpers J2,J3,J4 select one of eight 8K boundaries. Simply connect these three jumpers as described in step 5b.

2. Deciding whether or not you need to introduce wait states via jumper J1, as described in step 5a of the Assembly Guide.

Since you can't go wrong with a wait state, we recommend that you simply select the wait state option. That way you will be able to run SCREENSPLITTER on any 2 or 4 mhz machine with no further concern.

The virtues of running <u>without</u> a wait state are twofold:

a. The CPU runs slightly faster (but not a lot, since the wait state introduces only one additional period for every 4 or so normal periods)

b. The "snow" streaks caused by CPU accesses in the display buffer are shorter by one character's width on the average. (See the discussion in the Theory of Operation entitled "Snow Suppressor".) Although it is clearly desirable to minimze this dropout, the visible difference between running with and without a wait state is small.

Running without wait states gives the onboard EPROM and display RAM about 500 ns to respond. Because the EPROM requires 450 ns to settle, and the display RAM slightly less, but still in excess of 300 ns, the data on the data-in buss during memory reads will be valid only for the last 50-150 ns of this 500 ns window. You should attempt to run without a wait state only if your system samples the data-in buss no more than about 50 ns in advance of the end of the 500 ns read window.

It is possible to run 500 ns memory chips without wait states. To do so,

it is necessary to activate the memories in advance of the actual 500 ns read window by responding to SMEMR, and ignoring PDBIN. (Jumper J9 governs whether or not PDBIN will be involved in memory reads.) Since doing this would lock out the video logic for a slightly longer period of time, PDBIN is used in SCREENSPLITTER. However, if you have to introduce a wait state because of delay problems in your system, it might be to your advantage to omit the wait state, and ignore PDBIN. This will permit the CPU to run at full speed, and will produce "snow" streaks of about the same length as with the original wait state.

All 4 mhz applications will require installation of wait states. Because it may be convenient to be able to run on any system without jumper reconfiguration, we strongly recommend that you follow the "universal" setup described in the Assembly section.

Running Only During Blanking

If you have a demanding application where the small visible effects of the black "snow" caused by CPU accesses within the display RAM (see Theory of Operation, "Snow Suppressor") are unacceptable, or if you prefer to run most of the time with black text on white background (where the dropout is more noticeable), there is a way to "sanitize" the display. Since the dropout is caused by CPU accesses to the display buffer during visible periods of each TV field, if you restrict CPU accesses so that they occur only during non-visible periods of each field, the dropouts will not occur. Since non-visible periods are precisely those periods during which either or both of HBLANK and VBLANK (available externally via J8(pin 5) and J8(pin 7), respectively) are high, you must restrict the CPU display memory accesses to only HBLANK and VBLANK periods. (Note, however, that doing so will substantially reduce the speed of the display software - perhaps by up to a factor of ten!)

You can implement restricted access in either of two ways:

1. Use the externally accessible HBLANK and VBLANK signals to generate both start and stop interrupts to the CPU, alerting it when it may begin accessing the display buffer, and terminating it when its time is up.

2. Use HBLANK and VBLANK to introduce wait states. Since you may already

be introducing wait states via jumper J1, you may have to modify the board to implement this strategy.

We recommend the first strategy for two reasons: it requires no modification of the board, and it allows the CPU to proceed with other tasks while it waits for blanking. However, with either strategy, you will have to build some small amount of additional hardware which is mounted elsewhere in your system.

External Signals

The following 14 signals are made available externally via J8 (16 pin DIP socket) at the top edge of the board:

J8 Pin	Signal	Description
PIN 1	U1(pin 8)	10 mhz clock
PIN 2	U1(pin 4)	10 mhz clock, inverted
PIN 3	U4(pin 6)	Field
PIN 4	U16(pin 6)	Composite Sync, inverted
PIN 5	U4(pin 9)	Horizontal Blanking
PIN 6	U8(pin 8)	Horizontal Blanking, inverted
PIN 7	U6(pin 9)	Vertical Blanking
PIN 8	U6(pin 8)	Vertical Blanking, inverted
PIN 9	U14(pin 6)	Final Video, TTL level (less sync)
PIN 10	Q1(emitter)	Composite Video, 2 volts p-p
PIN 11	U34(pin 16)	Character Generator, bit 7
PIN 12	U34(pin 17)	Character Generator, bit 8
PIN 13	U2(pin 3)	Character Winker
PIN 14	U5(pin 8)	Video Suppresor
PIN 15	nc	unconnected (user definable)
PIN 16	nc	unconnected (user definable)

SOFTWARE INTEGRATION

SCREENSPLITTER SOFTWARE INTEGRATION

Introduction

The onboard Window Package defines 20 user-callable functions for controlling and writing to the display buffer. The basic unit of control is the window. Virtually any number of windows (rectangular subregions of the screen) can be defined and independently controlled. All status information for each window is maintained in the window's window descriptor block (WDB), a 9 or 11 byte block of RAM supplied by the user at the time the window is opened.

In all transactions with window functions, the user loads the address of the target window's WDB into the HL register, loads any additional parameters into registers B, C, D and E, then CALLs the desired window function. Window Package functions generally destroy all registers.

Windows and Window Descriptor Blocks

Each window defines a rectangular region of the display screen from size 1 by 1 up to 40 by 86. Each window can be manipulated independently from all other windows. Overlapping windows are permitted, but operations on a window take place without regard for possible effects on any overlapping windows.

Each window has its own set of parameters governing:

1. whether the window's figure/ground is reversed (black text on white background
2. whether the window has a frame (visible border)
3. whether the window's cursor is visible

4. what character is to be used as the window's cursor

5. whether the cursor displaying technique is to print the cursor character at the cursor location, or simply to reverse the figure/ground of the character at the cursor location

6. whether the window is to be held (by calling a user-specified "hold processor") at scroll time during output bursts

7. the type and degree of scrolling (pop-up or wraparound) the window will perform at window-full time.

The format of a window descriptor block is shown in Figure 1. The user is responsible for allocating this block of either 9 or 11 bytes (depending on whether or not the HOLD feature will be used) of free memory. Possible strategies for memory management of WDB's are described in a section below.

Ordinarily, the user will interact directly with a window's WDB only when (1) setting bits VC, CM, HO and OB in the STATUS byte to engage or disengage different modes of control, and when (2) loading the address of the hold processor in the last two WDB bytes when using the hold feature. Otherwise, the Window Package functions are responsible for maintaining and updating the parameters in the WDB at all times.

Window Package Interface

There are 20 user-level functions in the Window Package. Generally speaking, all functions perform error checking before modifying either the WDB or the visible display, and the error checking is intended to be thorough (i.e., it is not possible to clobber the display). Erroneous requests (e.g., window out of bounds, labeling with a string that will not fit on the top border, trying to frame a window that is not at least 3 by 3, and so forth) are simply ignored.

A call to a Window Package function typically comprises:

1. Loading any required parameters into registers B,C,D,E

2. Loading the window descriptor block pointer of the desired window

into the HL register

3. CALLing the appropriate Window Package function.

Generally, Window Package functions will destroy all registers except SP.

The Window Package uses the system stack, which must have been properly initialized prior to any Window Package interaction. All functions run in less than 64 bytes of stack space (FRAME is probably the greediest, yet probably runs in considerably fewer than 64 bytes). The Window Package guarantees that SP is always valid (i.e., there are no moments during which SP is not an accurate reflection of the top of the stack), so that any Window Package function may be safely interrupted in an interrupt-driven system.

The relative entry addresses of the 20 user-level Window Package functions are listed in Table 1. The complete Window Package Symbol Table is included at the end of this section as Table 2. Note in Table 2 that there are several internal functions that might be of interest for direct use.

Table 1 also contains the calling register conventions for the 20 functions, showing what information each function expects to find in each register at call time. The parameter mnemonics in the table refer to the mnemonics in the function descriptions below.

The Window Package Functions

The display screen is thought of as a 40 by 86 byte array. Screen rows are counted from row 0 (topmost) to row 39 (bottommost). Screen columns are counted from 0 (leftmost) to 85 (rightmost). Except for functions which affect a window's position on the screen, positions are expressed relative to the interior of a window. The topmost print line of a window's interior is line 0; the leftmost print column is column 0. Framing a window causes the interior to shrink by two cells in each dimension, with the new topmost row and leftmost column becoming line 0 and column 0, respectively. See the Window Package Source Code Listing for more details.

INIT(ch)

Clears the entire display buffer to character CH (typically ASCII blank, octal 040).

OPEN(wdb,x,dx,y,dy)

Opens a window of size DX rows by DY columns with top-left corner at screen row X, screen column Y; initializes the window's window descriptor block, WDB; does not clear the region.

CLEAR(wdb)

Clears the window and resets the cursor to the top left interior cell.

FRAME(wdb,hc,vc,cc)

Frames the window, using horizontal character HC for the top and bottom borders, vertical character VC for the left and right borders, and corner character CC for the four corners; if a frame is not already present, reduces the window's interior region by two characters in each dimension, and clears the window; if frame is already present, interior is unaffected.

UNFRAME(wdb)

Removes the window's frame (if any); if removed, increases the window's interior by two characters in each dimension and clears the window.

LABEL(wdb,str,len)

If the window has a frame, and if the string will fit, prints the string pointed to by STR, of length LEN, centered on the window's top border.

LABELS(wdb,str)

Behaves identically to LABEL, except that the string is assumed to be terminated by the string terminator character (octal 377), so that a length parameter is not required.

CURSORCH(wdb,ch)

Defines CH to be the window's cursor character; automatically complements CH if window's figure/ground is reversed.

SCROLL(wbd,n)

Sets the window's scroll parameter to N (0 for wraparound, greater than zero for N-line pop-up).

PRINT(wdb,str,len)

Prints the string pointed to by STR, of length LEN, to the window, starting at the current cursor position; LEN may be from 0 (no characters printed) to 65,535; the cursor is forced to the beginning of a fresh line after the

print.

PRIN(wdb,str,len)

Behaves identically to PRINT, except that the cursor is not forced to a fresh line after the print.

PRINTS(wdb,str)

Behaves identically to PRINT, except that the string is assumed to be terminated by the string terminator character (octal 377), so that a length parameter is not required.

PRINS(wdb,str)

Behaves identically to PRIN, except that the string is assumed to be terminated by the string terminator character (octal 377), so that a length parameter is not required.

FRESHLINE(wdb)

Forces the window's cursor to the beginning of a fresh line if it is not already there.

CLEARLINE(wdb)

Clears the window's cursor line, resetting the cursor to the first character of the line.

BACKSPACE(wdb)

Backs the window's cursor up one character if not already at the window's leftmost column; erases one character as it does so.

COMPLEMENT(wdb)

Reverses the figure/ground of the window's interior.

FLASH(wdb)

Reverses the figure/ground of the window's interior momentarily (about a third of a second for 2 mhz CPU).

PLOT(wdb,x,y,ch)

Prints character CH at window line X, column Y; does not affect the cursor position, and cannot invoke scrolling.

MOVEWINDOW(wdb,x,y,ch)

Moves the window so that the window's top left corner is on screen row X,

screen column Y; fills any vacated region of the screen with character CH (typically ASCII blank).

The Hold Feature

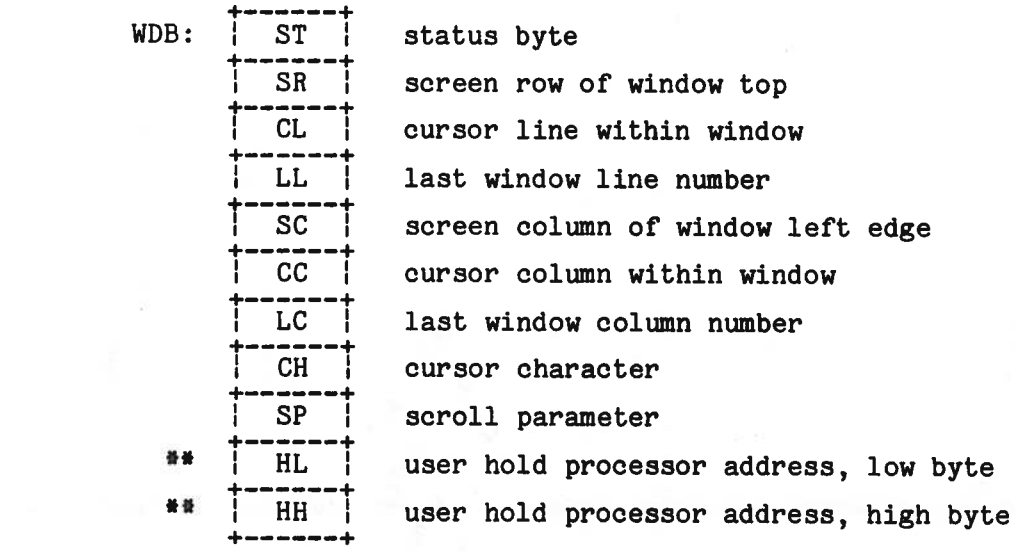
During rapid bursts of output which might cause a window to be entirely filled and begin scrolling, information can fly past the user faster than he can process it. By controlling two bits in a window's STATUS byte, and by preloading an address into WDB bytes HL and HH, you can introduce pauses during rapid output bursts.

When set, bit HO in the STATUS byte (see Figure 1) indicates that there is a legitimate "hold processor" address present in WDB bytes HL and HH. (This address should be loaded into the WDB before setting bit HO. You must do the loading yourself, since there is no Window Package function for this purpose.) The hold processor is a user-defined subroutine which will be called (immediately before scrolling or rolling) at window-full time during output operations to the window, whenever STATUS bit OB is on. The HO bit therefore indicates the presence of a hold processor, while the OB bit enables hold processing at scroll times.

When both HO and OB have been set and a window is about to scroll, the user-defined hold processor is called, with the HL register pair pointing at byte HL of the window's WDB. (The hold processor can infer the identity of the window being held from this pointer in HL.) The hold processor is free to take any appropriate action (e.g., flash or complement the window and wait for the user to type a go-ahead key), and it may destroy any registers as needed. The hold processor then returns to the Window Package, and the scrolling finally proceeds. Naturally, the hold processor should not attempt to write to the window being held!

Since you generally do not want the hold processor to be engaged (e.g., you do not want the window held when it is scrolling because of low-speed echoing of user-typein), OB should be set only at the beginning of long output bursts, and cleared immediately afterward. On the other hand, the HO bit will generally remain on, once set.

Figure 1. Window descriptor block format.



fg figure/ground bit
fr frame bit
vc visible cursor bit
cm cursor mode bit
ho hold bit
ob output burst bit
nc unused

* bit must be directly controlled by user

** byte must be directly loaded by user prior to setting status bit HO

Table 1. Window Package function summary.

Function	Relative Entry Address (octal)		Calling Parameters B	C	D	E	HL
C0-19-INIT	0	031	ch				
C0-28-OPEN	0	050	x	dx	y	dy	wdb
C0-AD-CLEAR	0	255					wdb
C1-EB-FRAME	1	353	hc	vc	cc		wdb
C2-6A-UNFRAME	2	152					wdb
C2-8D-LABEL	2	215		str		len	wdb
C2-8A-LABELS	2	212		str			wdb
C2-CE-CURSORCH	2	316	ch				wdb
C1-E3-SCROLL	1	343	n				wdb
C1-9A-PRINT	1	232		str		len	wdb
C1-96-PRIN	1	226		str		len	wdb
C1-8F-PRINTS	1	217		str			wdb
C1-88-PRINS	1	210		str			wdb
C3-4E-FRESHLINE	3	116					wdb
C0-C9-CLEARLINE	0	311					wdb
C3-3B-BACKSPACE	3	073					wdb
C2-DB-COMPLEMENT	2	333					wdb
C3-0A-FLASH	3	012					wdb
C3-23-PLOT	3	043	x	y	ch		wdb
C3-65-MOVEWINDOW	3	145	x	y	ch		wdb

NOTE: STR and LEN parameters occupy register pairs, as indicated above.

Table 2. Window Package Symbol Table (Relative to Location 0).

	Symbol	Page	Byte
*	CURSOR	0	000
*	CURSORAD	0	125
	UP7	0	104
	CUR1	0	027
**	INIT	0	031
	INI1	0	037
**	OPEN	0	050
	OKOPEN	3	326
	UP9	0	102
	UP8	0	103
	UP6	0	105
	UP5	0	106
	UP4	0	107
	UP3	0	110
	DOWN8	0	114
	DOWN7	0	115
	DOWN6	0	116
	DOWN5	0	117
	DOWN4	0	120
	DOWN3	0	121
*	CURSORLAD	0	143
*	LINEADDR	0	154
*	CLEARLIN	0	214
	CLL1	0	234
	CLEARCL	0	244
**	CLEAR	0	255
	CLR	0	263
	CLR1	0	277
**	CLEARLINE	0	311
	NEWLINE	0	326
	GLITCH	0	360
	HOLD	1	105
	GLT1	1	005
	GLT0	1	001
*	XYMOVE	1	123
	GLT2	1	071
	XYM1	1	130
	XYM2	1	145
	XYM3	1	171
**	PRINS	1	210
*	SLENGTH	2	273
	PRT	1	233
**	PRINTS	1	217
**	PRIN	1	226
**	PRINT	1	232
	PRT1	1	234
	PRT2	1	256
*	PRI	1	265
	FRLINE	3	124
*	CPLT	1	305
	PLT	1	317
**	SCROLL	1	343
**	FRAME	1	353
	FRM1	2	000
	TAKEIN	2	121
	FRM2	2	056
	FRM3	2	104
	FRM4	2	110
**	UNFRAME	2	152
	LETOUT	2	167
	TKO1	2	176
**	LABELS	2	212
**	LABEL	2	215
	LAB1	2	262
	SLN1	2	277

* Denotes useful internal function.

** Denotes user entry point.

	SLN2	2	314
**	CURSORCH	2	316
**	COMPL	2	333
	CPL0	2	337
	CPL1	2	367
	CPL2	2	371
**	FLASH	3	012
*	DELAY	3	031
	DLY1	3	034
**	PLOT	3	043
**	BACKSPACE	3	073
	BSP1	3	112
**	FRESHLINE	3	116
**	MOVWIN	3	145
	MVW0	3	263
	MVW1	3	272
	MVW2	3	304
	OK01	3	370

Software Integration

Interfacing to the Window Package functions from assembly language routines is straightforward. As an illustration, the code sequence below clears the display screen, opens a 10 by 15 window at row 5, column 40, frames it, labels it "Moving Window", prints a message to it, then moves it to location 10, 30.

```
mvi   b,040          clear display screen to blanks
call  init
lxi   hl,wdb         open the window
lxi   bc,10d/5d      at row 5, height 10
lxi   de,15d/40d     at column 40, width 15
call  open
lxi   hl,wdb         frame and clear it
lxi   bc,174/055     with dashes, vertical bars
mvi   d,000          and corner boxes
call  frame
lxi   hl,wdb         label it
lxi   bc,lab         with string lab
call  labels
lxi   hl,wdb         print "Hello"
lxi   bc,mes
call  prints
lxi   hl,wdb         move it to 10,30
lxi   bc,30d/10d
mvi   d,040          filling in with blanks
call  movwin
```

In many applications, however, you will want to interface SCREENSPLITTER to higher level programming languages and operating systems as the primary output device. The sections below describe integration techniques to higher level systems. Complete example interfaces are included at the end of the discussion.

Interfacing to BASIC and Other Systems

Most BASIC systems have an I/O hook: a block of memory into which the user inserts his own I/O routines when he integrates the BASIC system. Generally, all I/O is focussed down through two routines, INPUT and OUTPUT.

SCREENSPLITTER has nothing to do with the INPUT routine, since other system software is responsible for interacting with terminals, etc. However, you can modify the OUTPUT hook so that all output from BASIC is routed to SCREENSPLITTER.

The general strategy for connecting the OUTPUT hook to SCREENSPLITTER is

as follows. When you power up and first call BASIC, there must be some provision for opening a main window to which all "standard" BASIC output will be printed via the hook. If your BASIC has an initialization routine which it calls once at the beginning, that is an ideal place for the calls to INIT, OPEN, FRAME, CLEAR, or whatever you need to set up the kind of main window you want. If your BASIC has no initialization facility, you will probably have to set up the main window by inserting code in the BASIC OUTPUT hook that notices when it has been called for the first time, initializes the screen and main window, then disables itself for subsequent output calls.

In any event, setting up the main output window for BASIC will require that you set aside a 9 or 11 byte block of RAM for that window's WDB. An example initialization sequence in the BASIC OUTPUT hook might be coded as follows:

```
;       output the character in register A to the output device
;
output  <jump to OUT if not first time here>
        push  psw               save the output character
        mvi   b,040             clear display screen to all blanks
        call  init
        lxi   bc,30d,5d         open main window at row 5, column 7
        lxi   de,72d,7d         of size 30 by 72
        lxi   hl,mainwdb        point to main window's wdb
        call  open
        lxi   bc,174/055        frame and clear it
        mvi   d,000
        lxi   hl,mainwdb
        call  frame
        lxi   bc,lab            label it
        lxi   hl,mainwdb
        call  labels
        mvi   b,10d             set scroll parameter to 10-line pop-up
        lxi   hl,mainwdb
        call  scroll
        pop   psw               restore the output character
out     <remainder of output routine>
        ...
        ret                     return from OUTPUT routine
mainwdb block 11d               11 byte block for main window wdb
lab     str   "Main Window"     the main window label string
        d8    377               string terminator byte
```

After the main window has been opened, directing output to it is straightforward. Since BASIC will probably pass the OUTPUT hook one character at a time in register A, all you have to do is pass this along to PRIN or PRINS, keeping a lookout for cursor control characters such as carriage-return (octal 015), line-feed (octal 012), and whatever your BASIC's backspace character is. You will want to translate these position-controlling characters into calls on Window Package functions other than PRIN or PRINS (i.e.,

FRESHLINE, BACKSPACE).

A prototype single-character output interface is shown below. This interface assumes the output character is in register A, and that the system backspace character is ASCII 177. It ignores line-feeds, causing new lines to be begun by carriage-returns. Note that this code jumps to the Window Package, letting the Window Package return directly to the caller.

```
;         continuation of the OUTPUT hook
;
out       lxi   hl,mainwdb      load the main window's WDB
          cpi   012             test for line-feed
          rz                    line-feed, ignore
          cpi   015             test for carriage-return
          jz    freshline       carriage-return, call freshline
          cpi   177             test for backspace character
          jz    backspace       backspace, call backspace
          sta   char            store character as string for PRIN
          lxi   bc,char         point at it
          lxi   de,1            length= 1
          jmp   prin            print it
          ...
char      block 1               one free byte for output char string
```

With these two strategies for the OUTPUT hook, you should be able to use SCREENSPLITTER as a standard video display module via one main window of whatever size you choose, with or without a frame, etc. Since all BASIC print routines are funneled through the OUTPUT hook, not only will BASIC system prompts and error messages pass through to SCREENSPLITTER, but so will all output generated by PRINT statements in your BASIC programs.

Similar remarks apply to any system with a single OUTPUT hook.

Advanced Window Control

To enable BASIC to interact with more than one window, and to make use of the Window Package functions other than those referenced from the standard OUTPUT hook, you will need to establish facilities for managing window descriptor blocks and for switching the standard output hook from window to window.

Most BASICs have facilities for (a) storing or fetching a byte in the system's address space by its address, and (b) calling an assembly language routine at a fixed address in the system. If you do not have one or both of these facilities, you may have difficulties interfacing to the advanced

features of the Window Package. For the following discussion, we will assume that the following functions or their equivalents exist:

DEP(val,adr) - deposit byte VAL in memory address ADR

EXMN(adr) - fetch the byte at memory address ADR

CALL(adr) - call assembly language routine at memory address ADR

The general strategy for interacting with windows other than the main window is:

1. OPEN a new window, then frame, label and clear it as desired
2. Call a BASIC or assembly language routine (which you must write), FOCUS(wdb), which switches all subsequent standard output to window WDB
3. Print information to the selected window for a while
4. Switch to another window via another call to FOCUS.

and so forth.

The routine FOCUS(wdb) is either a BASIC or assembly language routine that loads X (a window descriptor block address) into a two byte storage area CURRENT-WINDOW somewhere in memory (e.g., in the OUTPUT hook). Then, if instead of always loading the HL register with MAINWDB (via the LXI instruction in the scheme above), the OUTPUT hook loads the HL register from CURRENT-WINDOW at each output, then the OUTPUT hook will be switchable from window to window.

Each window will be known to BASIC via the 16 bit integer that is the address of the window's WDB. Hence, to code FOCUS(wdb) in BASIC using DEP is nearly trivial: DEPosit the low order 8 bits of X in CURRENT-WINDOW, the high order 8 bits in CURRENT-WINDOW+1. Then, if you replace the LXI HL,MAINWDB instruction in the above OUTPUT hook by a LHLD CURRENT-WINDOW instruction, all basic I/O will be directed to the currently-selected window. CURRENT-WINDOW should be initially set to MAINWDB.

Invoking Other Window Package Functions

You must play all these games for basic I/O because you want to intercept all output at the single focal point of the OUTPUT hook. However, interfacing to Window Package functions from points other than the OUTPUT hook (e.g., from BASIC user programs) is more straightforward. As you will see, if your BASIC permits you to call an assembly language routine with parameters, the solution is trivial. We will assume here that you have only the DEP, EXMN and CALL primitives defined above.

Using DEP and CALL, the strategy for calling any Window Package function is the following:

1. DEP the calling parameters, the WDB address, and the entry address of the Window Package function you wish to invoke in a special block of memory PARAMS (for example, a reserved block within the OUTPUT hook); the PARAMS block would be 8 bytes long to accommodate the parameters for registers b,c,d,e,h,l and the entry address of the Window Package function about to be called

2. CALL an assembly language interface routine LINK (which you must write), which simply loads the parameters from the PARAM block, then jumps to the specified Window Package entry point.

(Note again, however, that this is the strategy for the worst case assumption that your BASIC has no mechanism for passing an assembly language subroutine parameters; the interface can be faster and simpler in BASICs with better linkage mechanisms.)

You will probably want to implement a BASIC function

INVOKE(fct,wdb,p1,p2,p3,p4)

which other BASIC functions call to invoke Window Package function FCT on window WDB, passing parameters P1,P2,P3,P4 in registers B,C,D,E. INVOKE would simply do the DEP's as described, then CALL the assembly routine LINK that loads the registers and branches to the appropriate Window Package function, as described.

Memory Management of WDB's within BASIC

For a general window control facility, you will require some method of allocating WDB's as you open new windows, and some mechanism for returning WDB's when windows are no longer needed. This calls for some sort of memory management of a pool of potential WDB's.

Within BASIC, you will probably want a function NEW-WINDOW() which produces a block of memory to accommodate a new window's WDB, and returns the memory address for this block. You would then be able to write sequences like:

```
LET W1= NEW-WINDOW()
CALL INVOKE(OPEN,W1,10,20,10,40)
CALL INVOKE(FRAME,W1,DASH,BAR,BOX)
...
```

where the BASIC variable W1 would remember the WDB address of the newly acquired WDB, and where the variables OPEN, FRAME, DASH, BAR and BOX would have been previously defined to their appropriate values. (We use descriptive mnemonics and syntax here; in most BASICs, you will have to use simpler mnemonics and different syntax.)

You will probably also want a companion function to NEW-WINDOW, say, RELEASE(wdb), which returns the window descriptor block WDB to the free storage pool of WDB's for reuse.

NEW-WINDOW(), RELEASE(wdb), and WDB memory management can be implemented entirely in BASIC as follows. You first decide on a reasonable size for the WDB pool (i.e., what is the expected maximum number of open windows), and where in the address space (probably outside of BASIC) you will situate the WDB pool. This pool will be of length (9 or 11) times the maximum number of windows. BASIC will know of the pool by assigning a variable, say WDBP, the starting address of the pool, and another variable, say WDBN, the address of the last WDB in the pool.

You will need some mechanism for determining of a WDB in the pool whether or not it is currently in use decribing a window on the screen. Since the Window Package does not use the last two bits in a WDB's status word (Figure 1), you can turn one of these bits on to denote that a block is in use. A BASIC WDB pool initialization routine must then be run once at log-on time to

clear the status byte of each WDB in the pool to zero (e.g., via a DEP(I,0), where I varies over the pool WDB's status byte addresses).

To acquire a WDB from the pool to describe a new screen window, NEW-WINDOW() can then simply scan the WDB pool looking for the first free block, turn on its busy bit, then return its address (or, in case there are no remaining free blocks, return a negative number to indicate a "WDB pool exhausted" condition). Conversely, RELEASE(wdb) simply clears the busy bit in block WDB.

Summary

A reasonable set of BASIC interface functions that follows from the discussion above is:

1. The OUTPUT hook, as described, for standard BASIC I/O to a main window
2. NEW-WINDOW(), which acquires a free WDB block from the pool and returns its address
3. FOCUS(wdb), which sets cells CURRENT-WINDOW in the output hook to direct all standard BASIC I/O to the window represented by WDB
4. RELEASE(wdb), which returns a WDB to the WDB pool

From these lowest level functions, you should be able to write all higher-level window control entirely in BASIC.

Final illustrative implementations for these four functions are shown on the following pages.

Switchable Assembly Language OUTPUT Hook for Standard I/O

```
;          output the character in register A to the current window
;
output     push  psw              save the output character
           lxi   hl,flag          see if this is first time here
           xra   a
           cmp   m
           jz    out              not first time, go print
           mov   m,a              first time, clear flag and initialize
           mvi   b,040            clear display screen to all blanks
           call  init
           lxi   bc,30d,5d        open main window at row 5, column 7
           lxi   de,72d,7d        of size 30 by 72
           lxi   hl,mainwdb       point to main window's wdb
           call  open
           lxi   bc,174/055       frame and clear it
           mvi   d,000
           lxi   hl,mainwdb
           call  frame
           lxi   bc,lab           label it
           lxi   hl,mainwdb
           call  labels
           mvi   b,10             set scroll parameter for 10-line pop-up
           lxi   hl,mainwdb
           call  scroll
out        pop   psw              restore the output character
           cpi   012              test for line-feed
           rz                     line-feed, ignore
           lhld  curwin           load WDB address of current I/O window
           cpi   015              test for carriage-return
           jz    freshline         carriage return, call freshline **
           cpi   177              test for backspace character
           jz    backspace        backspace
           sta   char             store character as string for PRIN
           lxi   bc,char          point at it
           lxi   de,1             length= 1
           jmp   prin             print it
flag       d8    1                flag to determine first-time call
curwin     d16   mainwdb          current window's WDB address
mainwdb    block 11d              11 byte block for main window wdb
lab        str   "Main Window"    the main window label string
           d8    377              string terminator byte
char       block 1                one free byte for output char string
```

** FRESHLINE will not cause blank lines to be skipped on sequences of carriage returns. If you want blank line skip control, replace this call with a PRINT of a single ASCII blank.

The BASIC Function FOCUS

```
100 REM This is the BASIC subroutine FOCUS(w) which sets the
200 REM window of current focus to W.
300 DEF FNF(W)
400 DEP(W-INT(W/256)*256,CURWIN)
500 DEP(INT(W/256),CURWIN+1)
600 RETURN
700 FNEND
```

INVOKE and LINK for Direct Window Package Calls from BASIC

```
100   REM This is the BASIC function INVOKE(fct,wdb,p1,p2,p3,p4) that
200   REM causes Window Package function FCT to be invoked with parameters
300   REM P1,P2,P3,P4 on window WDB.
400   DEF FNI(F,W,P1,P2,P3,P4)
500   DEP(F-INT(F/256)*256,PARAM)
600   DEP(INT(F/256),PARAM+1)
700   DEP(W-INT(W/256)*256,PARAM+2)
800   DEP(INT(W/256),PARAM+3)
900   DEP(P1,PARAM+4)
1000  DEP(P2,PAPAM+5)
1100  DEP(P3,PARAM+6)
1200  DEP(P4,PARAM+7)
1300  CALL(LINK)
1400  RETURN
1500  FNEND
```

```
;           LINK: link to Window Package function from PARAM block.
;
link        lxi   hl,param+4      load registers b,c,d,e
            mov   b,m
            inx   hl
            mov   c,m
            inx   hl
            mov   d,m
            inx   hl
            mov   e,m
            lhld  param+2         load the window descriptor block
            d8    303             jump to function entry address
param       block 8d              parameter block
```

WDB Management Functions in BASIC

The WDB pool is known to BASIC by the integer variables WDBP and WDBN, representing the start address of the WDB pool (probably outside BASIC) and the address of the last WDB in the pool, respectively. Each WDB will be 11 bytes long, with the lowest order bit of the STATUS word of each WDB used to indicate whether or not the WDB is busy (0: not busy, 1: busy). Initially, all pool WDB STATUS bytes should be set "not busy":

```
100 FOR I=WDBP TO WDBN STEP 11
200 DEP(0,I)
300 NEXT I
```

The function NEW-WINDOW(), which returns either the address of a free WDB or -1 (to indicate there are no free WDB's left) would then be as shown below. In this code, we assume the existence of the function EXMN(adr) which returns the byte stored at memory address ADR (the inverse of DEP).

```
100 DEF FNW()
200 FOR I= WDBP TO WDBN STEP 11
300 IF EXMN(I)=0 GOTO 600
400 NEXT I
500 RETURN -1
600 DEP(1,I)
700 RETURN I
800 FNEND
```

RELEASE(wdb) would be coded as:

```
100 DEF FNR(W)
200 DEP(0,W)
300 RETURN
400 FNEND
```

TROUBLESHOOTING

SCREENSPLITTER TROUBLESHOOTING GUIDE

Hardware problems will almost always stem either from solder bridges or faulty IC's. Solder bridges usually result from inadvertantly dropping molten solder across the board during assembly. (SCREENSPLITTER's solder mask on both sides minimizes the risk of creating solder bridges. However, even if you are an extremely careful solderer, you should expect a solder bridge once in a while!) The other source of bridges is the manufacturing process itself, where etching fails to remove enough metal in critical places, or where the solder reflow process leaves behind tiny splatters. Although quality control usually prevents bridges of manufacturing origin, some faulty boards inevitably sneak through. (Naturally, bad boards are our fault, and we will gladly fix them for you at no charge!)

If you have trouble bringing up SCREENSPLITTER, or if it breaks in the future, first scan the Theory of Operation to get an idea of how the system works. If you feel able to cope with the problem yourself, read the Theory in detail, then follow the diagnostic table below. (You will need a scope, preferrably with two channels.) If you would rather have Micro Diversions repair your system, by all means, send it back with all parts; we can usually offer two-week turnaround.

Problems will typically appear on the scope in one of three forms:

1. A line that should be changing seems to be permanently high or low (this could be either a bad chip or a solder bridge);

2. A line that seems to be changing properly, but that does not undergo a full voltage swing (from 0 to better then 3 volts is normal); this is usually the result of a solder bridge;

3. A line with more signal on it than expected; this is usually also the result of a solder bridge, where two signals become mixed.

For suspected solder bridge problems underneath sockets (where you can't see

the traces), refer to the copies of the artwork supplied in this section. If you find bridges, they can usually be removed either by reheating, or by scraping with a sharp implement of some sort. When you suspect a solder bridge underneath a socket, you will either have to remove the socket (unsoldering tools are available), or attempt to sever traces underneath by drilling through the socket and board with a small bit, then jumpering around the problem.

If you suspect a faulty component, there are three ways of isolating the fault:

1. If most of the board will still run without the suspect chip, remove it and see what happens. This is an appropriate strategy for the memory chips, and the counters and latches in the final stages.

2. Switch components. If there are more than two of the same type of chip and you suspect one of them, switch them and see if anything changes.

3. Replace the suspect chip.

Since you generally cannot damage the logic by removing chips, one general strategy is to remove progressively more chips in an attempt to localize the problem. (For example, many problems can be diagnosed without the memory chips, which tend to obscure the display raster.)

If your system seems to work, but you experience jitter, and random noise on the screen, (either constantly, or after the system warms up), the chances are good that the sync generator is at fault. If you suspect the sync generator, try a little mechanical pressure on it in its socket; if things happen on the screen, the problem is almost certainly the sync generator. If you have followed the precautions described in the Assembly manual, and you suspect the sync generator, send it back for a replacement, at no charge.

Signal Table

This first set of measurements can be made with a single-channel scope, since they are not synchronized to a second signal. The notation indicates the chip or component first, pin number in parentheses. All times and frequencies are approximate. Measurements should be made with a static display (i.e., without the CPU running). You should first verify that all voltage regulators are working properly, and that the coaxial cable connector is not shorted. (With the system powered down, you should measure about 75 ohms across the coaxial connector's inner and outer conductors.)

1. U1(8), basic clock: 10 mhz "square" wave (actually, not very square!)
2. U7(11), input to sync generator: 1.25 mhz square wave
3. U6(9), vertical blanking: high 1.4 ms every 16.7 ms
4. U4(6), field: 60 hz square wave
5. U4(9), horizontal blanking: high 11 us every 63.5 us (you will see multiple traces or jitter, since you are not synchronized to vertical blanking)
6. U3(16), composite sync: low 5 us every 63.5 us (same comment as measurement 5 - you might also see longer low periods, corresponding to vertical sync)
7. U15(2), snow suppressor: constant high
8. U2(3), winker: one-half hz

The remaining measurements should be synchronized to the falling edge of vertical blanking, U6(9). If you do not have a two-channel scope, give them a try anyway. First, remeasure items 5 and 6 above, synchronized to the falling edge of vertical blanking. You should see stable displays of these signals.

9. U24(14), scan line counter, bit 0: 63.5 us square wave
10. U24(13), scan line counter, bit 1: high 63.5 us, low 127 us, high 127 us, low 63.5 us, ...
11. U24(12), scan line counter, bit 2: high 63.5 us, low 254 us, high

127 us, then repeat low 254 us, high 127 us

12. U24(11), scan line counter, bit 3: high 63.5 us every 381 us

13. U24(9), scan line counter reset: low 63.5 us every 381 us

14. U9(1), display address latch advance: high 11 us every 381 us

15. U23(14), pixel counter, bit 0: 5 mhz square wave

16. U23(12), pixel counter, bit 2: high 200 ns every 600 ns

17. U22(6), last-pixel-in-character signal, low 100 ns every 600 ns

18. U28(14), display buffer address, bit 0: 1.2 us square wave

19. U30(14), display buffer address, bit 4: 19.4 us square wave

20. U32(14), display buffer address, bit 8: low 800 us, then five spikes, then high

21. U5(5), first-character-in-line suppressor: low 600 ns, then low 12 us every 63.5 us

22. U15(1), character video: low 63.5 us, then four 50 us bands of video, separated by 13.5 us low (assumes that the display memory has powered up randomly, so that there are random characters across the top character row of the screen)

23. U17(6), figure/ground flip flop: high/low levels (minimum 600ns) corresponding to the figure/ground sequence of characters on the top row of the screen

24. Q1(e), emitter of final output transistor Q1: 2 volt peak-to-peak final video, corresponding to first row's figure/ground for first 63.5 us, sync pulse every 63.5 us, subsequent 63.5 us periods containing character video

APPLICATIONS NOTES

SCREENSPLITTER APPLICATIONS NOTES

SCREENSPLITTER's 40 by 86 upper lower case display and logical segmentation into windows make practical TV interfaces that are not realistic on smaller, single window displays. For openers, here are 5 example applications that rely on SCREENSPLITTER's large display size and Window Package:

1. As an interface to a high level language, such as BASIC

   In this application, each important subroutine and function is assigned its own private window. As it runs, each subroutine or function can output trace information, status indications, or user prompts through its own window. The visual effect would be flurries of activity from window to window, each independently scrolling, flashing, etc. This provides a very effective way to see "in two dimensions" exactly what's going on inside your programs. User typein can be directed through individual windows, giving the illusion of making it possible for the user to converse with components of a large system independently.

2. As a Debugging Display System

   In this application, relevant status information concerning the execution of a program (as output by a debugging package) is displayed through numerous windows. One window, for example, could display the register and accumulator contents; another could display the top N items on the system call stack; another could flash up interrupts as they were serviced; another could display a selected portion of central memory, such as a critical array; another could display a real-time/run-time clock. You

could even do all these things independently for a number of subroutines, presenting and recording, say, the registers and accumulator as they were at subroutine exit time. You could also arrange to have the normal I/O of your program appear in one window, with debugging information popping up through another window beside it when requested.

3. As a Basis for Controlling Several Keyboards

In this application, there are several keyboards, each with its own window, as might be useful in multiple player computer games. Each player would have his own area on the screen, into which all input typed by him would be echoed, and to which all output directed at him would be written. Under certain circumstances, a player might be given access to another player's window, or "party" windows might be established to combine the inputs from several users into a single window.

4. As a Basis for Advanced Page-Oriented Text Editing

In this application, entire pages of text are displayed on the screen in one relatively large window. As you decide to move paragraphs or lines around, you issue commands to pick up a paragraph or line and place it in a smaller holding window. As you rummage through the main window looking for the spot at which to insert the line or paragraph, the holding window remains fixed. Finally, you issue a command to insert the contents of the holding window at a selected point in the main one. The editor would perhaps be capable of directing its editing powers at the text in any window, so that you could also modify the line or paragraph in the holding window before reinserting it into the main window. Also, using the MOVEWINDOW function, you could actually lay out a screen of text (as it is about to be printed on a hard-copy device) by moving paragraph-size chunks of text around. Meanwhile, of course, there could be a very small window up in the corner containing a real time clock ticking away!

5. As a Basis for Networking and Concurrent Processing

In this application, you might wish to be doing local computing, but

remain connected to some external computer (over the phone lines) or to a computer network of other personal machines similar to yours. One window would then be reserved for all I/O in your local computation, with additional windows through which communication with the external computers could occur. You might allocate one window for each other personal computer attached to the network. This would enable you to keep all the I/O to the various communicating computers separate from each other, and separate from your local computations. Again, party windows could be established.

Window Package Control: Some Pointers

1. Window Opening, Framing and Labeling

Note that windows can be defined either with or without a frame, and that a frame can be removed or altered after being defined. For visible frames, we suggest using the following three border characters:

```
horizontal character:  ASCII 055  (dash)
vertical character:    ASCII 174  (vertical bar)
corner character:      ASCII 000  (centered box)
```

Note that window labeling can be done only after framing a window. Note also that, since the FRAME function does not notice the presence of a label, if you reframe a labelled window, you will also have to rewrite the label. To create a labeled window with no visible frame, frame the window with ASCII blanks (octal 040's), then LABEL it.

You can cause frames to flash by calling FRAME with alternating parameters several times, rewriting the label (if present) each time. (FRAME and LABEL run fast enough for the visual effect to be an instantaneous border change.) Momentary reframing of a window with blanks, or with complemented current frame characters give the effects of a winking and flashing frame, respectively.

2. Sneak Windows: Moving Text Around

Since opening a window does not affect the visible display, it is quite

possible to open a window on top of an existing window or windows. If the newly opened window is then moved, any text it encompasses will follow along. This provides the basis for a powerful text editing facility, in which parts of paragraphs can surrounded by a "sneak window" which is then moved. Entire paragraphs can be moved in this fashion for final layout work, or the technique can be used at the line and character levels: to insert a line at line J of an existing paragraph N lines long, surround lines J through N-1 with a sneak window, then move the sneak window down one row! Similarly, to insert a character in the middle of an N character long existing line at position J, open a sneak window over characters J through N-1, then move the long skinny window right one character. Symmetric operations can be used for deleting a line or character from existing text.

3. Sneak Windows: Highlighting

Using sneak windows, portions of text can be highlighted (figure/ground reversed) or flashed. When you want to highlight an entire window, COMPL and FLASH are appropriate functions. Often, however, it is useful to focus attention to some portion of a window by highlighting or flashing only several lines or characters. By opening, then complementing or flashing sneak windows, you can effectively highlight or flash any rectangular subregion of text in a larger window. For irregularly-shaped subregions, you can open and control several sneak windows in parallel. (Again, Window Package operations are fast enough so that all control will appear to be simultaneous.)

4. Cursor Control

Note that there are three types of control over each window's cursor:

a. What ASCII character is to be used as the cursor character.

The cursor character is initialized to the caret (octal 037). By calls to CURSORCH, you can redefine the cursor character to any other ASCII code. For example, you may wish to call attention to one window by changing the cursor to the winking caret (octal 177) during some editing operation, or during periods of output activity to the window.

b. Whether or not the cursor is visible.

The default, established by OPEN when a window is opened, is for a visible cursor. However, there are times when you wish cursor control to have no visible display effects (when plotting, or posting status information to small windows, for example). By altering bit VC of the window's STATUS word (you must do this manually - see the Software Integration section), you can turn the cursor off and on.

c. What the cursor-displaying technique is, if the cursor is visible.

Bit CM of the window's STATUS word governs the type of cursor control. When this bit is 0, the cursor is represented by displaying the cursor character at the cursor position; when the CM bit is 1, the cursor is represented by reversing the figure/ground of the character at the cursor position. This latter cursor mode is useful in page-oriented text editing applications where it is undesirable for the cursor to occupy its own separate character position.

5. Manual Control of Window Descriptor Blocks

When opening a window, you are responsible for giving OPEN a pointer to a block of storage to use as the new window's descriptor block (either 9 or 11 bytes, depending on whether or not you will be using the HOLD feature). Although you will ordinarily address the window simply by passing this window descriptor block pointer to the Window Package on each subsequent transaction with the window, you obviously have access to the descriptor block for manual access and control. As described in the Window Package and Software Integration section, you must rely on this manual access to the block for operations that require the setting of bits VC, CM, HO and OB in the block's STATUS byte. However, other useful information is also accessible: the window's current screen position, the cursor's current position, etc. Naturally, you are free to sample and alter a window's descriptor block information any time you choose; simply take care not to clobber anything!

6. Closing Windows

There is no close function for "closing a window". This is unnecessary, since the Window Package itself has no memory of the window in the first place! (The window's descriptor block is somewhere else in the system.)

However, it is often necessary to close a window visually, removing it from the screen to make room for another window, etc. This can always be accomplished by: (1) either setting the cursor character to ASCII blank or turning off the cursor as described above, (2) UNFRAMEing the window , then (3) CLEARing the window. If there is a frame, it is removed, then the window is cleared, leaving no visible cursor. All visible traces of the window thus vanish. If memory management of the window descriptor block storage space is necessary, you would also mark the closed window's WDB free at that time. (See the Software Integration section.)

7. Block Lettering

If you have applications where large block lettering must be displayed for viewing at a distance, note that adjacent figure/ground-reversed blanks (octal 240) will appear as uninterrupted broad bands of white on the black background. You can use these for large lettering, coarse graphics, computer games, etc.

8. Window Shadowing

One technique for highlighting a window is to move it one or two cells diagonally on the screen (e.g., down one cell, left one cell), filling in the vacated region with figure/ground-reversed blanks. On a black background, there is a three-dimensional effect that makes the window appear to have depth. When finished, move the window back, storing uncomplemented blanks in the vacated region to restore the window to its original condition.

9. Scrolling and Rolling

Note that you may set the scroll parameter to either rolling wraparound, or N-line pop-up. N can be anything from 1 to the full height of the window interior. Full-height pop-up is equivalent to an automatic window clear each time the window fills up. We have found a pop-up of about a third of the window's height to be pleasing.

10. Skipping Blank Lines

Window Package function FRESHLINE will force a window's cursor to the

beginning of a fresh line if it is not already there. However, it will not advance the cursor if it is already at the beginning of a fresh line. When you wish to skip blank lines during print operations, call FRESHLINE, then PRINT of a single ASCII blank. This will always cause exactly one blank line to be skipped.

11. Displaying Stacks: Pushing and Popping

You may have occasion to display a system stack that exhibits last-in, first-out behavior. For this, you will probably want to open a tall narrow window into which you print the names of functions or data objects on the stack. When a new item is pushed onto the stack, you will want its mnemonic to appear on the next line of the stack window. When the item is popped from the stack, you want the cursor to back up a line, "popping" visually.

If you wish to display the stack with most recent entry on the bottom of the display, pushing corresponds to a regular PRINT to the window. But, since there is no line-backup function, you must implement pop operations manually by decrementing the CL (current cursor line) byte in the stack window's WDB. Naturally, you should note when the cursor line is already 0 at pop time so that a wraparound or null action (whichever you prefer) occurs for that special case. Implementing a stack that grows visually from the bottom of the window up follows the same ideas.

12. Using Window Package Internal Functions

Note that there are several internal Window Package functions (designated by single asterisks in the Window Package Symbol Table in the Software Integration Section) that might be of use. PRI, XYMOVE, CURSORAD, and CLEARLIN are perhaps the four most interesting. Consult the Window Package Source Listing for detailed documentation and calling conventions.

THEORY OF OPERATION

SCREENSPLITTER THEORY OF OPERATION

Overview

SCREENSPLITTER consists of 14 logical modules, which together implement all hardware and software functions:

1. The Clock Module, which generates the basic 10 mhz square wave that drives and synchronizes the system

2. The Sync Generator Module, which generates all the signals required for driving a TV monitor, and for synchronizing the display logic counters and shift register with the movement of the electron beam in the TV monitor

3. the Scan Line Counter Module, which keeps track of which scan line of the current character row is being scanned

4. the Horizontal Pixel Counter Module, which keeps track of which pixel (bit) of the current character is being displayed

5. the Display Buffer Address Counter Module, which does the addressing of the display buffer, causing the appropriate character to be fetched for display at each moment

6. the Final Pipeline Buffer Module, which holds fetched characters while the character generator works on them

7. the Character Generator Module, which generates the pixel grid for each fetched character

8. the Final TTL Video Module, which shifts character pixels onto the TV display, and which controls winking and figure/ground reversal

9. the TV Interface Module, which mixes the TTL video with the TV sync, and which shifts voltage levels and impedance to values suitable for

driving the TV monitor

10. the Display Buffer Memory Module, which stores the displayed text, and which controls both host CPU and TV logic access to the text

11. the Snow Suppressor Module, which suppresses white-on-black snow caused by CPU accesses to the display buffer

12. the CPU Wait Logic Module, which introduces a user-selectable wait state into host CPU memory requests to the display buffer and Window Package EPROM

13. the Software EPROM Module, which stores the Window Package, and controls its interface to the host system's address and data busses

14. the Power Supply Interface Module, which regulates and filters the power supply inputs to the system.

Clock

U1(units 2,3,4,5), XTAL1, R2, R3 and C12 implement the 10 mhz square wave clock module. The 10mhz signal is available at U1(pin 8), 10 mhz-inverted at U1(pin 4). This clock synchronizes all display-related events on the board. Specifically, it controls the shifting and latching of data in the final shift register U33, the counting of the synchronous pixel and display memory counters U23, U28, U30, U32, and the fetching and decoding of bytes from the display memory RAM.

Sync Generator

U7, U3, U4, U6, U10(unit 2), and U16(unit 2) implement the sync generator module. U7, used as a divide by 8 counter, reduces the 10 mhz clock to 1.25 mhz for driving the MM5320 sync generator U3. Five 5320 signals are used: composite sync (CSYNC-inverted) at U3(pin 16), horizontal drive (HDRIVE-inverted) at U3(pin 15), composite blanking (CBLANK-inverted) at U3(pin 14), vertical drive (VDRIVE-inverted) at U3(pin 11), and field index (FINDEX-inverted) at U3(pin 9). Because the 5320 blanking signals are not precisely matched to the needs of the display logic, the 4 D-type flip flops of U4, ~~U5~~ U6 assist in the generation of these signals.

U4(unit 2), responsible for HBLANK, is set at the falling edge of HDRIVE-inverted, and is cleared by rising edges of CBLANK-inverted. The resulting outputs, HBLANK and HBLANK-inverted at U4(pins 9,8), define the period during which the display video is off between each scan line (for horizontal retrace), approximately 10.5 microseconds. This latch essentially extracts the horizontal blanking from the composite blanking emitted by the 5320 (and would be unnecessary if the 5320 generated separate horizontal and vertical blanking signals).

U6(units 1,2) serve to transform the VDRIVE-inverted signal from the 5320 into a vertical blanking signal; like U4(unit2), they essentially extract vertical blanking from the composite blanking provided by the 5320. U6(unit 1) is set at the falling edge of VDRIVE-inverted, and persists until the next rising edge of CBLANK-inverted. When U6(unit 1) is set, since its output controls the preset line of U6(unit 2), the change is immediately reflected through U6(unit 2) to become the vertical blanking signals VBLANK and VBLANK-inverted, U6(pins 9,8). U6(unit 2) ensures that, for the odd field, video does not turn on halfway through the top scan line (as directed by the 5320), since this would throw the memory address counters out of synchronization. On the first rising edge of CBLANK-inverted after VDRIVE-inverted at U6(pin 4) releases U6(unit 1) from preset mode, U6(unit 1) is cleared by strobing in the logical 0 at its pin 2. Clearing U6(unit 1) releases U6(unit 2) from preset mode. The <u>next</u> rising edge of CBLANK-inverted marks the beginning of the first full scan line of the odd field, and it is this rising edge that finally clears U6(unit 2) and causes VBLANK to fall.

U4(unit 2) is used to transform the FINDEX-inverted pulse, emitted momentarily at the beginning of each even field, into a level that persists for the duration of each field. U4(unit 2) is hence set by the falling edge of FINDEX-inverted at U4(pin 4), and cleared by the falling edge of the next VDRIVE-inverted, which denotes the end of the even field and beginning of the odd field. FIELD at U4(pin 6) is the generated output.

U16(unit 2) is used as a buffer to shift the 5320's CSYNC-inverted output from a +5 to -12 volt swing to TTL level U16(pin 6).

The outputs of the sync generator circuitry are therefore:

U4(pin 9) HBLANK

U4(pin 8) HBLANK-inverted
U6(pin 9) VBLANK
U6(pin 8) VBLANK-inverted
U4(pin 6) FIELD
U16(pin 6) CSYNC-inverted

Scan Line Counter

U24, U9(units 2,4), and U1(unit 6) implement the scan line counter module. Synchronous 4 bit counter U24 counts scan lines within character rows, via HBLANK-inverted at pin 2. Each row of characters consists of two dead scan lines at the top, 8 visible scan lines, and two dead scan lines at the bottom, making each row of text 12 scan lines in height. Since on each field of a frame (there are 30 frames/sec, two fields per frame) only the 6 even scan lines, or the 6 odd scan lines are displayed, U24 is a 6 state counter that counts 15-0-1-2-3-4, changing state on the falling edge of HBLANK (i.e., at the beginning of each scan line, just as the video is turned on for the scan line). Count 15 corresponds to the top dead line on each field, counts 0,1,2,3 correspond to the 4 even or odd visible scan lines of the row, and count 4 corresponds to the bottom dead line on each field. U24 is forced to 15 by VBLANK, so that the first scan line of each field (top of the screen) is count 15. U9(units 2,4) and U1(unit 6) perform the modulus reset function for U24.

Since the 4 even or 4 odd visible scan lines of each character row are represented by counts 0,1,2,3 of U24, the two low-order bits of U24 directly feed the address bits of the character generator that determine which scan line of a character is to be fetched, U34(pins 6,7). The third, high order scan line bit input to the character generator (three scan line address bits are required to identify which of the eight visible scan lines is being displayed) is FIELD, the slowly-toggling (60 hz) signal that denotes whether it is the even or odd field that is currently being scanned. This use of FIELD accounts for interlace.

Since only the scan lines corresponding to counts 0,1,2,3 of U24 are to be visible, bit 3 of the counter U24(pin 12), which is on only for counts 15 and 4, is fed to U9(pin 9) to suppress loading of video data during the dead line at the top and bottom of each character row on each field.

Horizontal Pixel Counter

U23, U22(unit 2), and U10(unit 1) implement the horizontal pixel counter. This 6 state counter, reset to 0 by HBLANK's, keeps track of which of the 6 pixels horizontally across the current character's current scan line is being displayed. U23 counts at the 10 mhz rate, resulting in a pixel width of 100 ns.

Characters are 5 pixels wide, with one dead pixel between characters. Counts 0,1,2,3,4 correspond to the visible pixels of the character, count 5 to the dead pixel between characters. Since HBLANK allows approximately 53 microseconds of visible video on each scan line, U23 will cycle 0,1,2,3,4,5 88 and a fraction times for each video scan line. It is this counter that dictates the number of characters on each line.

U22(unit 2) decodes count 5 of U23 and generates U23's synchronous reset: U22(pin 6), LASTCOL-inverted, goes low for the 100 ns of count 5 of U23, the count that corresponds to the last pixel of the current character (the dead pixel between this and the next character). U10(unit 1) generates the true form of this signal, LASTCOL. Thus, LASTCOL and LASTCOL-inverted are the outputs of the horizontal pixel counter module, lasting 100 ns and having a period of 600 ns, the width of a character.

Display Buffer Address Counter

U28, U29, U30, U31, U32, and U9(unit1) implement the display buffer address counter. It is this counter that generates memory addresses in the range 0-4095 for fetching the ASCII codes from the buffer for display at the proper moments. (Actually, the count is terminated before 4095 by VBLANK; only the first 3550 or so bytes are ever referenced.)

Since each row of 88 and a fraction characters across the screen actually consists of six even field scan lines, interlaced with six odd field scan lines, in order to generate each row, the same 88 and a fraction bytes of display memory must be fetched six times, each pass being used to generate a progressively lower screen scan line of the character row. (In reality, only four passes, for the four visible scan lines of each row on each field, would be necessary; but the circuit is simpler if the two dead lines are treated as though they are important also.)

Generation of the first row of characters proceeds as follows:

1. count from 0 to 88 for the first scan line, then reset to 0
2. count from 0 to 88 for the second scan line, then reset to 0
3. count from 0 to 88 for the third scan line, then reset to 0
4. count from 0 to 88 for the fourth scan line, then reset to 0
5. count from 0 to 88 for the fifth scan line, then reset to 0
6. count from 0 to 88 for the sixth scan line

But after reaching 88 at the end of the sixth scan line, the counter should not reset, but rather "fall through" to the next block of 88 characters in the display buffer, the block between addresses 88 and 175, inclusive. Then it must cycle through this block 6 times, resetting to address 88 on each cycle, and so forth. Synchronous 4 bit counters U28, U30, and U32 do this memory address counting, advancing on the falling edge of LASTCOL each 600 ns. The result is that a new display buffer memory address comes up on V11-V0 every 600 ns, beginning on the falling edge of LASTCOL.

Latches U29 and U31 keep track of the base address of the block of 88 characters in the display buffer corresponding to the currently scanned character row. So that they begin at count 0 at the top of the screen, they are cleared by VBLANK at the end of each field. During the course of a complete field scan, they will hold base addresses 0, 88, 176, ... 3432 (the base address of the 40th character row). Each base address is held for the first five of the six scan lines of the character row; then, at the rising edge of HBLANK for the sixth scan line, as signaled by U9(pin 1), the latches immediately latch in the current count of the address counter (V11-V0), which at that moment holds a count which has "fallen through" to the next block of 88 bytes in the display buffer.

At the end of each scan line HBLANK-inverted, connected to U28,U30,U32(pin 9), causes the counter to stop counting and to load in the value contained in the U29, U31 latch. Five times out of six, this will force the counter back to the beginning of the current block of 88 bytes. But on the last line, the latches will have quickly grabbed the "fall-through" value of the counters, so that when the counters load from them (many times during HBLANK), the counters will simply receive their present value, i.e., they will not reset to the old base address. (The latches latch on the rising edge of HBLANK, whereas the counters load on the falling edge of HBLANK.)

The outputs of the display buffer address counter module are therefore

V11-V0, an address in the range 0-4095. This address is sent to the display buffer RAM as the address of the character currently being fetched.

Final Pipeline Buffer

U35, U36, and U5(unit 1) implement the final pipeline buffer. This is primarily an 8 bit latch that latches in the byte from the display buffer whose address has been present on V11-V0 during the preceding 600 ns. The latched byte, which enters on lines C7-C0 from the display RAM on the falling edges of LASTCOL, is interpreted as 7 bits of ASCII (C6-C0), and one bit of figure/ground (C7). Each fetched byte resides in the latch for 600 ns while the character generator is working on it.

All latching in the U35, U36 buffer, and in subsequent video steps is synchronized to the falling edge of LASTCOL (i.e., the rising edge of the first pixel of the next character). The U35, U36 latch is the first step in a three-step pipeline, with the character generator U34 being the second step, and the final shift register U33 the third. As one character is being shifted out of the shift register for display, another character is residing in the U35, U36 latch (being worked on by the character generator), and yet another character is being fetched from memory in preparation for latching into U35, U36. Thus, there is a two character lag (1200 ns) between the time a character's address first comes on V11-V0, and the time its first pixel becomes visible on the screen.

Because of this pipeline, the first two character periods of each scan line are not meaningful, and should not be displayed. To suppress the video during these first two 600 ns time periods, D-type flip-flop U5(unit 1) is cleared by HBLANK, so that each new scan line begins with U5(unit 1) in a logic 0 state. This suppresses, via U22(pin 9), loading of video information into the final shift register. On the falling edge of the first LASTCOL of the new scan line, U5(unit 1) restores itself to its preferred state of logic 1. At the end of the second 600 ns character period, this logic 1 signal will be present at U22(pin 9) to enable loading of the shift register on the falling edge of the second LASTCOL. Thus, on all but the first LASTCOL falling edge, U22(pin 8) will request the final shift register U33 to load video data. (Because of U9(unit 3), the requests get through only during the visible scan lines of each character row.) The result is suppression of the video display

for the first two character periods of each scan line.

Character Generator

2708 EPROM U34, a 450 ns device, serves as the character generator. It responds to two sources of information: the scan line number on its address pins 8,7,6, and the ASCII code of the character under generation. The ASCII code comes from the U35, U36 latch, feeding 2708 address pins 22,23,1,2,3,4,5.

At the end of each 600 ns character period, the character generator will have produced an eight bit byte of scan line information. The high order two bits are not used, but are routed to the 16 pin external jumper socket for user extensions. The sixth bit of the generator output is interpreted as the wink bit, and is latched into D-type flip-flop U17(unit 2) at the end of each 600 ns character period. The remaining 5 bits of generator output are latched into the final shift register U33, also at the end of each 600 ns character period. At that time, a sixth bit, a logical 0, is also latched into the final shift register to generate the one pixel inter-character space.

Final TTL Video

U33, U17(units 1,2), U22(unit 1), U13(unit 3), U15(unit 1), U16(unit 3), U14(unit 2), U2, R4, R5 and C11 implement the final TTL video module. At the falling edge of each LASTCOL during which video loading into U33 is enabled via U9(unit 3), three pieces of information enter the final video section:

1. The five character scan line pixels, plus one dead pixel, enter the final shift register U33
2. the wink bit for the scan line enters flip-flop U17(unit 2)
3. the character's figure/ground bit, being held separately in latch U36 while the character generator is working on the other seven ASCII bits, enters U17(unit 1), the figure/ground flip-flop.

The wink bit in U17(unit 2), and the figure/ground bit in U17(unit 1) will prevail for the entire 600ns period during which the six video pixels of the current character are being shifted out of U33.

Final shift register U33 shifts on the falling edge of the 10 mhz clock,

presenting each pixel to U15(pin 1) for 100 ns. At the end of six pixels, the shift register loads the next scan line from the character generator and thus begins the next character, when enabled by U9(unit 3).

To ensure that the final shift register shifts out only zeroes during the first two meaningless character periods of each scan line, U33 is cleared to all zeroes by the HBLANK-inverted line at U33(pin 9). Since U33 will not latch data until the end of the second character period of each new line, the first two character periods will appear as two characters of zero pixels.

Likewise, to ensure that the accompanying figure/ground bit for the first two character periods is zero (uncomplemented), U17(unit 1) is also cleared by HBLANK.

A 555 timer U2, running in astable mode, generates the wink signal. When enabled, via U22(unit 1), by the wink bit having been set in U17, this signal is ANDed with the final video by U15(unit1). The visual result is that all character scan lines with wink bits appear to wink simultaneously on the screen.

When not suppressed by U15(pin 2) (suppression is discussed below), possibly winking video emerges inverted, a pixel at a time, out of U15(pin 12). If the figure/ground flip-flop has been set, and if the suppressor circuitry is not active at U13(pin 9), the video's figure/ground is reversed as it passes through U16(unit 3). Thus, possibly winking, possibly figure/ground reversed video emerges inverted, a pixel at a time from U16(pin 8).

To ensure that no visible video signal occurs during either horizontal or vertical blanking, the video is blocked by U14(unit 2) during these periods. Final video, in true form, issues from U14(pin 6).

TV Interface

Discrete components Q1, Q2, R6, R7, R8 and R9 implement the TV interface, whose purpose is twofold: to mix the final video at U14(pin 6) with the composite sync at U16(pin 6), and to shift voltage and impedance levels from TTL values to 2 volts peak-to-peak, 75 ohms.

The composite video required by TV monitors is such that absolute white

is defined at 2 volts, absolute black as 0.5 volt, and sync as 0 volts. To realize the sync portion of this standard, the CSYNC-inverted from U16(pin 6) is used to control transistor Q1. When CSYNC is low (active), Q1 is off, forcing FINAL VIDEO on the coaxial cable to ground through the 75 ohm resistor R6. When CSYNC-inverted is high (inactive), Q1 is on, connecting the FINAL VIDEO to the output of the Q2, R8, R9 network which, in conjunction with R6, establishes a voltage divider.

When the TTL level final video of U14(pin 6) is high (white), transistor Q2 is on, shorting out R8. As a result, FINAL VIDEO sees voltage divider R9-R6 dividing 5 volts. Under this circumstance, the FINAL VIDEO output is 5 volts times the ratio of R6 to R9, or about 2 volts. On the other hand, when the TTL video at U14(pin 6) is off, Q2 is off, and does not short out R8. FINAL VIDEO then sees a voltage which is 5 volts times the ratio of R6 to (R8+R9), or about 0.5 volt. The resulting FINAL VIDEO signal is sent to a 10 mhz or better TV monitor via the 8' coaxial cable.

Display Buffer Memory

Page two of the schematic is the display buffer memory module and related circuitry. The display buffer itself is eight 1024 byte by 4 bit 2114 static random access memory chips, U41-U48, having access and cycle time of 450 ns or better. Their address lines are pins 15,16,17,1,2,3,4,7,6,5, and their bidirectional data lines are pins 11,12,13,14; pin 8 is CHIP SELECT-inverted, and pin 10 is READ WRITE-inverted (logic 0 for write, logic 1 for read).

The 2114 address pins are driven by quad 1-out-of-2 selectors U38, U39 and U40. Two addresses compete for the 2114's: the TV display logic lines V11-V0, and the host system's address buss A11-A0. Ordinarily, the selectors allow V11-V0 to feed through to the 2114 address pins, causing them to respond to fetch requests from the video circuitry via lines C7-C0 which feed the final pipeline buffer latch U35, U36. For the periods during which the video logic's address prevails, the 2114 READ/WRITE-inverted lines (pin 10) are left at logic 1 level, indicating read operations, the only type the video circuitry demands.

The low order ten address lines from the selectors U38, U39, U40 are routed directly to the display RAM's to select one of 1024 bytes. U45 stores

the high order 4 bits of each of the first 1024 bytes in the 4096 byte buffer; U44 stores the low order 4 bits of this first 1K of the 4K. In a similar way, the U46-U43 pair stores the second 1K bytes, the U47-U42 pair the third 1K, and the U48-U41 pair the fourth 1K. 4 line to 10 line decoder U37, fed by the two high order address bits from U40(pins 12,9), selects exactly one of the four pairs by bringing the CHIP SELECT-inverted pin 8 low for that pair. As a result, exactly one of the four pairs of 2114 memories is always active, and normally posting its fetched data on C7-C0 for the display logic.

From time to time, however, the host CPU will post a read or write signal to the display buffer memory in order to print or examine a visible character on the screen. U10(units 3,5,6) and U14(unit 3), in conjunction with the user-settable jumpers J2,J4,J3, decode the high order three bits of the host CPU's address buss, and define the 8K boundary at which the board is mapped. When a read or write reference to any address in this 8K block of the host's address space occurs, VSELECT at U14(pin 8) goes high. If A12 is also high, then the display buffer memory, mapped as the upper 4K of the decoded 8K, is being addressed. If the host is performing a write operation to the display buffer, when the MWRITE signal is emitted U15(unit 3) goes active low. If the host is performing a read operation from the display buffer, when PDBIN and SMEMR go high (the host's way of indicating a read), U11(pin 8) goes active low.

Under either of these circumstances, the U38,U39,U40 selector, via U22(unit 4), immediately and unconditionally switches in the host system's address, switching out the the video address V11-V0 which is in progress. In the case of a host write request, after a suitable delay through U26(units 1,2,3), the display buffer memory READ/WRITE-inverted is also brought low, putting the 2114's in write mode. Simultaneously, the host system's data-out buss DO7-DO0 is made available to the 2114's via tri-state buffers U20 and U21. In the case of a host read request, the 2114 READ/WRITE-inverted lines are left high, and U11(pin 8) directs the tri-state data buss drivers U18, U19 to turn on, connecting the output of the 2114's to the host system's data-in buss DI7-DI0. In either case, the host CPU's address locks out the video circuitry for the duration of the active read or write request from the host CPU. At the instant the host CPU releases the buffer, the U38,U39,U40 address selector switches video address V11-V0 back in, probably in mid-cycle.

Snow Suppressor

Giving the CPU instantaneous priority over the buffer simplifies the circuitry and allows the CPU to run at full speed; but it disrupts the orderly fetching of data bytes by the TV logic. Since characters are being fetched for display at the rate of one each 600 ns, if the CPU's read or write request lasts 500 ns, then either one or two 600 ns video fetch cycles will be partially damaged. In particular, the TV logic is likely to latch in the data byte involved in the CPU transaction, rather than the byte it would normally have latched in its orderly scan. Thus, for one or two display character periods, the character generator is likely to generate the scan lines of some characters other than the characters scheduled to be displayed at those one or two positions on the screen. The visual result would be "snow", or brief moments where one scan line of one or two adjacent characters is erroneous (i.e., some other ASCII code's scan line). Even an occasional erroneous segment on a mostly-black screen is quite visible to the human eye, and undesirable.

To suppress snow, a three-stage pipeline U8(units 1,2), U5(unit 2) is used to model the three-stage pipeline inherent in the video logic. Ordinarily, this three bit pipeline of D-type flip-flops is filled with logic 1's, which are constantly shifting out at U5(pin 9) on the falling edge of LASTCOL. However, at the instant a CPU request begins interfering with the video address, U8(unit 1) is immediately cleared to logic 0 to indicate that the cycle represented by this step in the pipeline has been destroyed. As long as the CPU has control, U8(unit 1) remains in the forced clear state, causing one logic 0 to enter the suppressor pipeline on each falling edge of LASTCOL. When the CPU finally releases the circuitry, the forced clear at U8(pin 1) vanishes, and on the next LASTCOL falling edge, logic 1's will again begin entering the suppressor pipeline.

A logic 0 in the suppressor pipeline indicates that the corresponding byte in the final video pipeline finally comes up for display in the output shift register U33, it will probably be invalid, and should be suppressed. At that time, the suppressor's logic 0 will prevail at U5(unit 2)'s outputs, and, via U15(unit 1) and U13(unit 3), kill the output video to black for the duration of the invalidated character's scan line. As many cycles of suppression as were shifted into the sppressor pipeline will eventually

emerge, delayed by three cycles, to suppress as many characters' scan line segments as required, typically one or two without a memory wait cycle, two or three with a memory wait cycle. (If the host CPU is based on a hardware front panel that functions by putting the CPU into long wait states, then when an examine or deposit to the display buffer occurs, the host CPU will actively lock out the video for long periods of time. As a result, the display goes entirely dead.)

Since CPU access to the display buffer during horizontal blanking (vertical blanking as well) cannot cause visible snow, the suppressor pipeline is preset to logic 1 by HBLANK-inverted at U8(pin 4) so that accesses during HBLANK time do not cause erroneous suppression of the first characters of lines.

Buffers U26(units 1,2,3) are necessary to remedy a slight timing problem. Without them, tri-state buffers U20, U21 turn on at host CPU write time after one gate delay (their own) from the write signal at U15(pin 8). At that instant, they start interfering with the fetched display byte at C7-C0. Most of the time, the suppressor circuitry catches this. However, since there are two gate delays, via U22(unit 4) and U1(unit 1), between the write signal and the suppressor logic at U8(pin 1), there is a two-gate delay window (about 20 ns at the end of each 600 ns cycle) during which the video data might be interfered with, but whose interference will be recorded by the suppressor pipeline as belonging with the <u>next</u> character. In other words, because of the difference in gate delays between the tri-state buffers and the suppressor pipeline, there is a 20 ns window at the end of each 600 ns cycle during which suppression will be thrown off by one character. Interposition of the three buffers of U26 guarantee that U20,U21 will not turn on in advance of the suppression logic. (There is then a 10 ns window at the <u>beginning</u> of each 600 ns cycle where suppression can miss interference; but since 590 ns of valid fetch time will remain, there is no need for suppression during this window anyway.)

<u>CPU Wait Logic</u>

U12, U13(units 1,2,4), U16(units 1,4), U15(unit 2) and part of U19 implement the user selectable CPU wait circuitry. This circuitry, when enabled by jumper J1, will cause any memory read or write cycle to either the display

RAM or 2708 software EPROM to persist for an additional CPU minor cycle (500 ns for a 2 mhz CPU, 250 ns for a 4 mhz CPU).

U12(unit 2) and U13(units 1,2,4) generate the wait signal itself; U12(units 1,3,4), U16(units 1,4) and U15(unit 2) determine when this signal is to be made available to the host CPU via its PRDY line. Each PSYNC at U13(pin 2), emitted by the CPU at the beginning of each major cycle, causes the RS flip-flop implemented by U13(units 1,2) to be cleared, sending a logic 0 to pin 2 of tri-state buss driver U19. When activated by U15(unit 2), the U19 will become active and reflect this logic 0 through to the system PRDY line, indicating a "not ready" condition. At the coincidence of either a MWRITE or PDBIN (active for memory reads) from U12(unit 2) and PHI-2 (the system's second phase clock) at U13(unit 4), the U13(units 1,2) RS flip-flop is set back to logic 1, causing the system's PRDY line to switch to the "ready" condition. The result is a memory wait of one clock period.

Whether or not the "not ready" signal (which is constantly being generated) is actually reflected through to the system's PRDY buss is controlled by jumper J1. When wired to +5, the tri-state gate controlling the PRDY buss line at U19(pins 1,2,3) is held in tri-state mode, so that no wait states are reflected through to the system buss. However, when wait states are selected by connecting U19(pin 1) to U15(pin 6), the tri-state buffer is controlled by U15(unit 2). When any address in the 8K block of host address space into which the TV circuitry has been mapped is referenced, VSELECT at U15(pin 5) goes high. If either host address line A12 is high, indicating a display buffer reference, or ROM ADDR is high, indicating a reference to the software EPROM (from the third page of the schematic), U16(pin 3) goes high also. If the reference is determined to be either a memory read or a memory write (as opposed to an I/O operation) by U12(units 3,4), then U16(pin 11) also goes high. At the coincidence of these three signals, U15(pin 6) goes low, turning on the PRDY tri-state buffer until the referenced address is taken off the host address lines.

Software EPROM

U27, U25, part of U26, U11(unit 1) and U14(unit 1) implement the onboard 1K or 2K byte software module. U27 is either a 2708 or a 2716 EPROM, with jumper selectable power supply and address lines. For the 2708 provided, J7

connects U27(pin 21) to -5 power, and J6 connects U27(pin 19) to +12 power. Other style 2708's or 2716's can be accommodated by altering J6 and J7.

U25 and part of U26 connect the EPROM with the host system's data-in buss. When a memory read request causes PDBIN and SMEMR both to be high at U11(pins 2,4) and when an address in the software EPROM is detected. U11(unit 1) goes active low, enabling the EPROM and connecting it via the tri-state buffers to the host's data-in buss. References to the software EPROM are detected by VSELECT at U11(pin 5) and by U14(unit 1). VSELECT goes high for any reference within the display circuitry's 8K memory block. When A10 is connected to U14(pin 1) via jumper J5, U14(pin 12) goes high for references to the first 1K block within the 8K block. When U14(pin 1) is tied to ground via jumper J5, U14(pin 12) goes high for references within the first 2K block of the 8K block, accommodating a 2K 2716 EPROM. Host address buss lines A9-A0 feed the EPROM address lines directly to determine which byte is fetched from the EPROM on a memory read cycle.

Power Supply Interface

VR1-VR4, C1-C10, C13, R1 and Z1 implement the power supply interface circuitry, responsible for regulation and filtration of the +8, +18 and -18 volt power supplies provided by the host system. VR1 and VR2 regulate the +8 supply down to +5, with VR2 supplying +5 to the eight 2114 display RAMS, VR1 supplying all other components. Electrolytic capacitor C1 filters the +8 supply at entry to the board, with tantalum capacitors C3,C4,C5,C6, together with the 0.01 microfarad disc despiking capacitors distributed around the board, provide further filtration for the +5 supply.

VR3 regulates the -18 supply down to -5 for the two 2708 EPROM's, with C7, C8 and C13 providing filtration and stabilization of this -5 supply. VR4 regulates the +18 supply down to +12 for the two 2708 EPROM's, with C9 and C10 providing the filtration. Resistor R1 and zener diode Z1 regulate the -18 down to -12 for the small amount of current required by the 5320 sync generator, with capacitor C2 providing filtration for the -12 supply.

The outputs of the power supply interface circuitry are regulated and filtered 2 amps at +5 volts, 1 amp at -5 and +12 volts, and about 50 ma at -12 volts. Only about 1.5 amp of +5, 150 ma of +12, 100 ma of -5, and 25 ma of -12 are actually used.

HARDWARE HACKS--

AT YOUR OWN RISK!!

HARDWARE HACKS - AT YOUR OWN RISK!

Described below are three SCREENSPLITTER extensions that spring from our basic human nature: we always want more! We do not necessarily endorse any of these ideas; they just look good on paper!

HACK 1: Squeezing more lines onto the screen

The standard display is 40 and a fraction character lines of 86 and a fraction characters each. There are two blank scan lines at the top and two blank scan lines at the bottom of each character line, for a total of four blank scan lines between character lines. The blank lines are generated by U9(unit 3), which blocks the load signals for the final shift register on counts 15 and 4 of scan line counter U24's count sequence 15,0,1,2,3,4.

Since only about the first three-quarters of the display buffer is used in the standard configuration, you have room to expand. By altering the blank scan line generating circuitry, you can cut the number of blank scan lines between character rows to either 2 or 0. This alteration probably only involves cutting U24's count sequence down by one or two counts by removing one or both of counts 15 and 4, depending on whether you want 2 or 0 blank lines between character lines, respectively. Removing 15 from the count sequence is probably as simple as tying the parallel load inputs of U24 to ground, instead of +5 as they currently are. Removing 4 from the count probably only involves the substitution of U1(unit 6) with a nand gate to decode the condition U24(13)=U24(14)=1. Whatever you do, counts 0,1,2,3 must remain, since these are used to tell the character generator which scan line of the current character to generate.

There are about 484 visible scan lines. Because U24 counts 15,0,1,2,3,4 in the standard configuration, each character line occupies 12 scan lines (6

even field counts, 6 odd field counts), and there are hence 40 and a fraction visible character lines. If you alter the count sequence to remove either 15 or 4 (but not both), each character line will then occupy 10 scan lines, meaning you will see 48 and a fraction lines of text. Since each line is 88 characters wide, you will then require a display buffer of size 88*48, or 4224. Since the display buffer is only 4096 bytes, the display address counter will wrap around to 0 somewhere on the 46th character line, redisplaying the first and part of the second text lines. Since this is undesirable, you will have to adopt the software convention of keeping the top two character lines blank. Although you then have only 44 character lines (48 minus the first and last two), you still come out 4 lines ahead. Naturally, a similar (and more acute) problem occurs when you reduce the count to 0,1,2,3, dropping both counts 4 and 15. For this reason, 0-line spacing via this technique is probably not very useful.

HACK 2: Squeezing more characters on each line

The magic number 88 characters per line (86 and a fraction visible, 1 and a fraction invisible) stems from the basic 10 mhz clock rate. Since the sync generator allows 53 and a fraction microseconds visible time per scan line, and since each pixel, defined by the basic clock, is 100 ns wide, 530 some-odd pixels can be displayed across each scan line at the 10 mhz rate. Since characters are 6 pixels wide, the result is 88 characters per line.

If you increase the basic clock frequency, say to the 11 to 12 mhz range, you can pack more characters per line, since the counters will simply count faster across the screen, calling up more text. At 11 mhz, you would probably get 97 characters per line, with 95 whole, visible ones. 40 lines of 97 characters each requires a display buffer of under 4096 bytes, so all 40 lines would still be usable. At 12 mhz, you would probably get 105 characters per line. However, then you would exceed the display buffer, and, to suppress the wraparound visually, would have to keep several of the top lines blank, as in the first hack.

The basic clock can be modified simply by substituting crystals. However, since the sync generator still needs a 1.25 mhz input, you will have to replace U7 (which divides by 8) with some other divide strategy. With an 11.25 mhz crystal, for example, you could divide by 9, keeping the sync

generator happy, and getting 98 or 99 characters per line.

HACK 3: Making characters contiguous

In the standard design, inter-line and inter-character spaces (4 scan lines, and one pixel, respectively) are generated by the hardware. The inter-line spacing hardware is that mentioned above (and described in more detail in the Theory of Operation section). The inter-character pixel space is generated simply by loading a trailing zero into the final shift register U33 along with the five bits of character generator output.

There are many applications where it is desirable to have character contiguity, i.e., to have control over whether or not there are spaces between characters and lines. The simplest and most convenient way of having this control is to expand the character generator, implanting vertical and horizontal spacing explicitly around each character. Then, for characters that should be contiguous, you simply turn on bits in the character generator clear out to the edges of the character. In SCREENSPLITTER's case, each character will have to become 12 scan lines high and 6 pixels wide to achieve full contiguity.

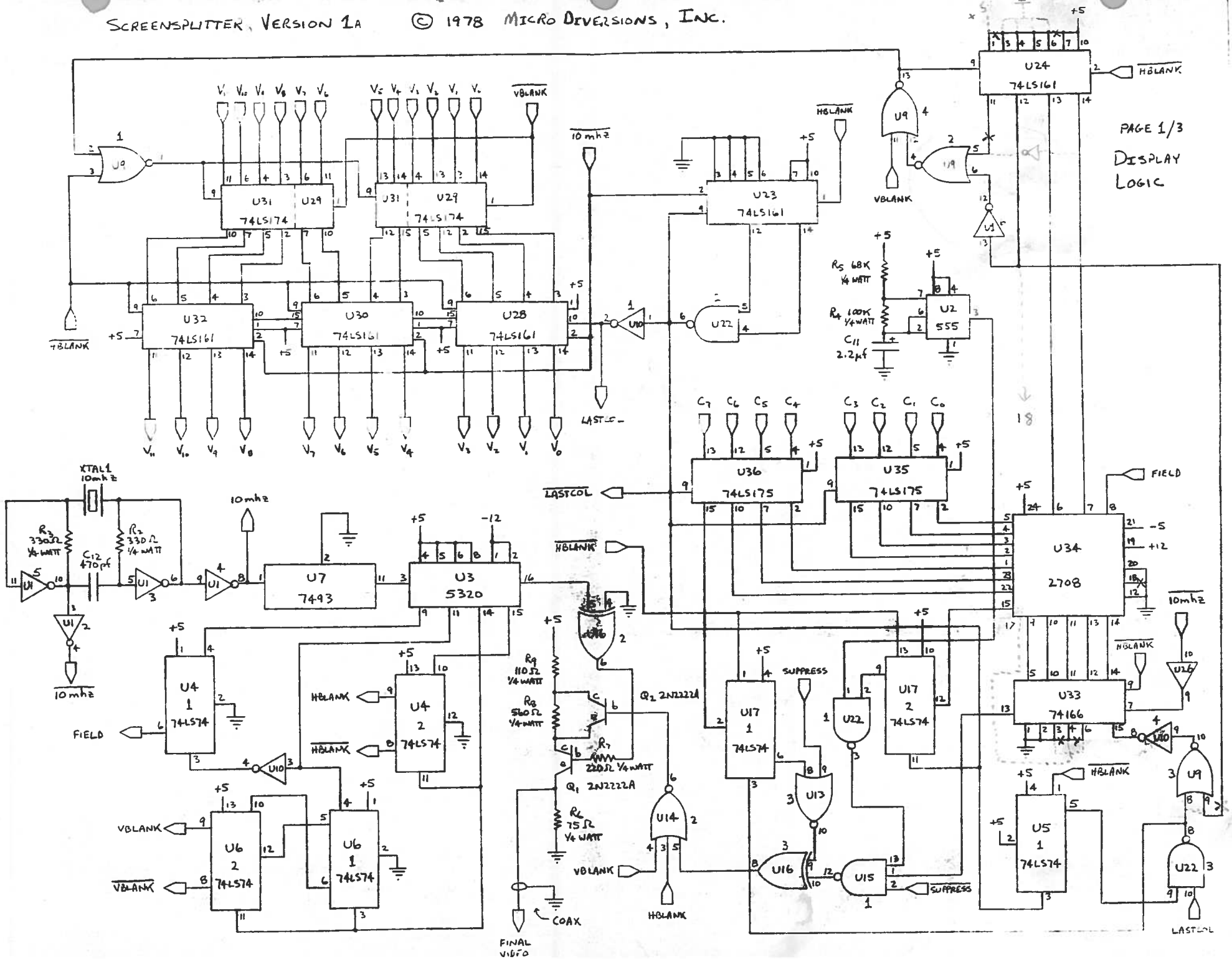
Transferring the inter-character hard-wired zero pixel to the character generator should be straightforward. In the standard configuration, bits 0-4 of the character generator are used to encode character line data, bit 5 is used to denote winking, and bits 6 and 7 are unused (and programmed to logic zero in the supplied generator). Hence, by removing the hard connection to ground, and connecting the space pixel to bit 6 or 7 of the character generator, inter-character spacing will be derived from the character generator. This has the advantage of allowing characters to be horizontally contiguous, as might be desirable in graphics applications.

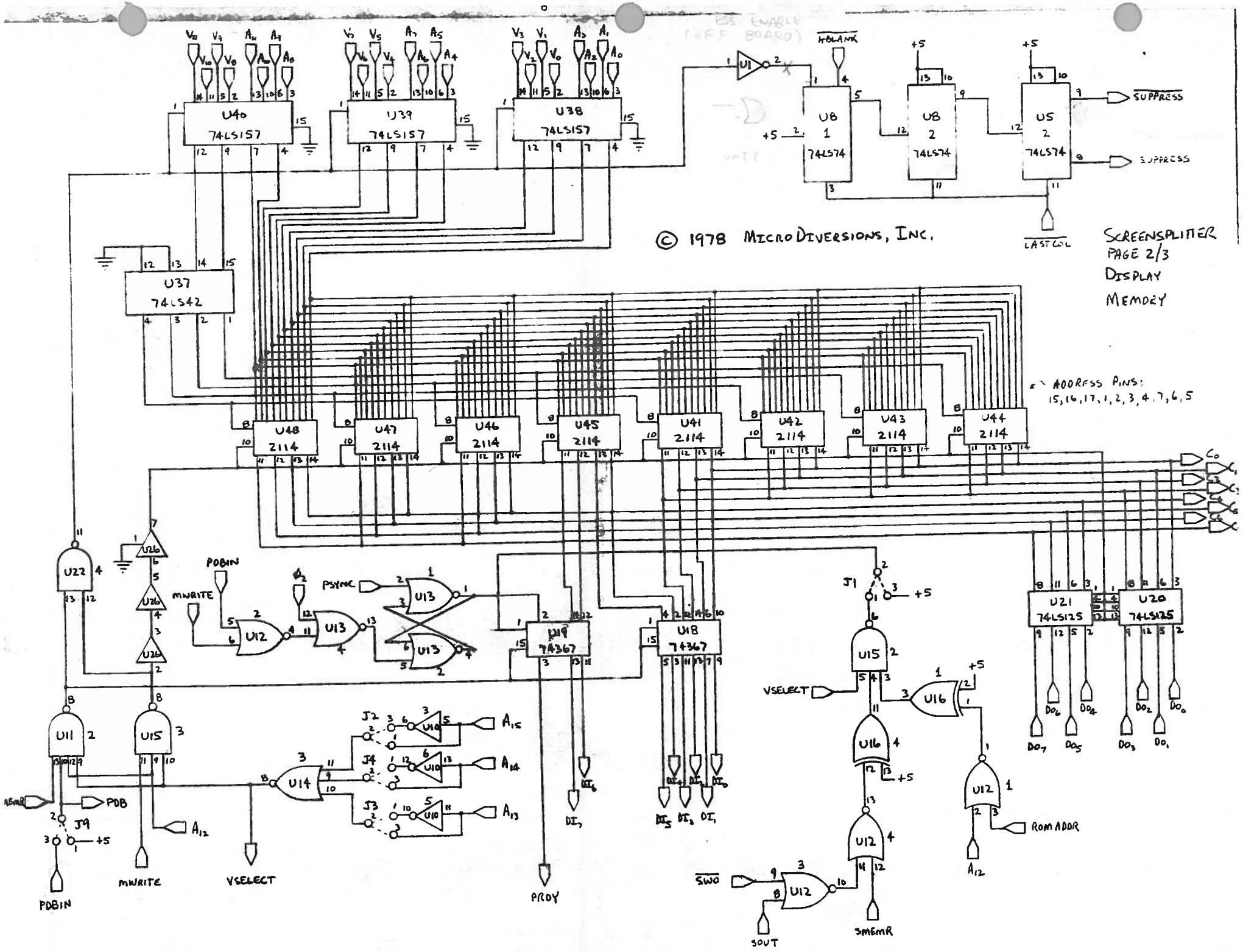
Vertical contiguity is somewhat more involved, but still not too much work. The basic idea is to replace to 2708 generator with a 2716, which is twice the size. Then, instead of storing only the 8 visible scan lines of each character, you store all 12 scan lines, including what are now the two dead lines at the top and bottom of each character row, directly in the generator. You will have to feed the new 2716 one more scan line address bit from U24(11), and you will have to defeat the hardware that generates the blank

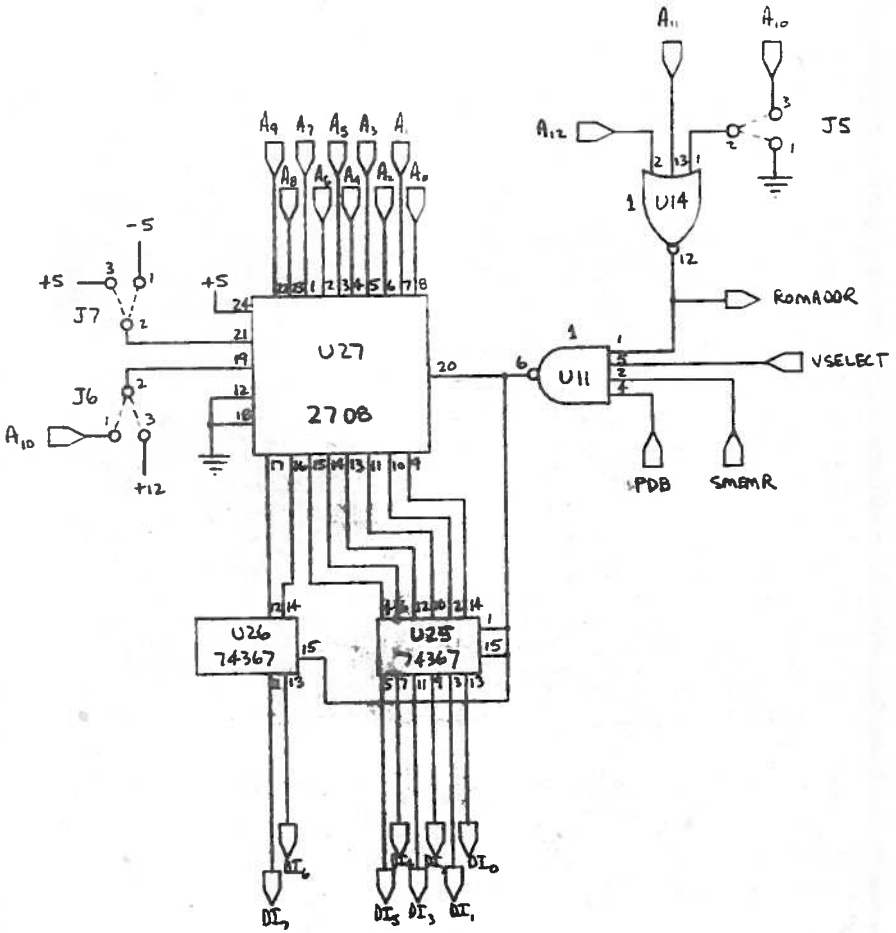
scan lines in the standard configuration. (This probably amounts to nothing more than severing U9(9) and wiring this pin permanently low.) Naturally, you will have to rewire the power connections to accommodate the 2716, and you will have to program the new character generator yourself. But you will then have a completely contiguous character raster for better graphics.

SCHEMATIC

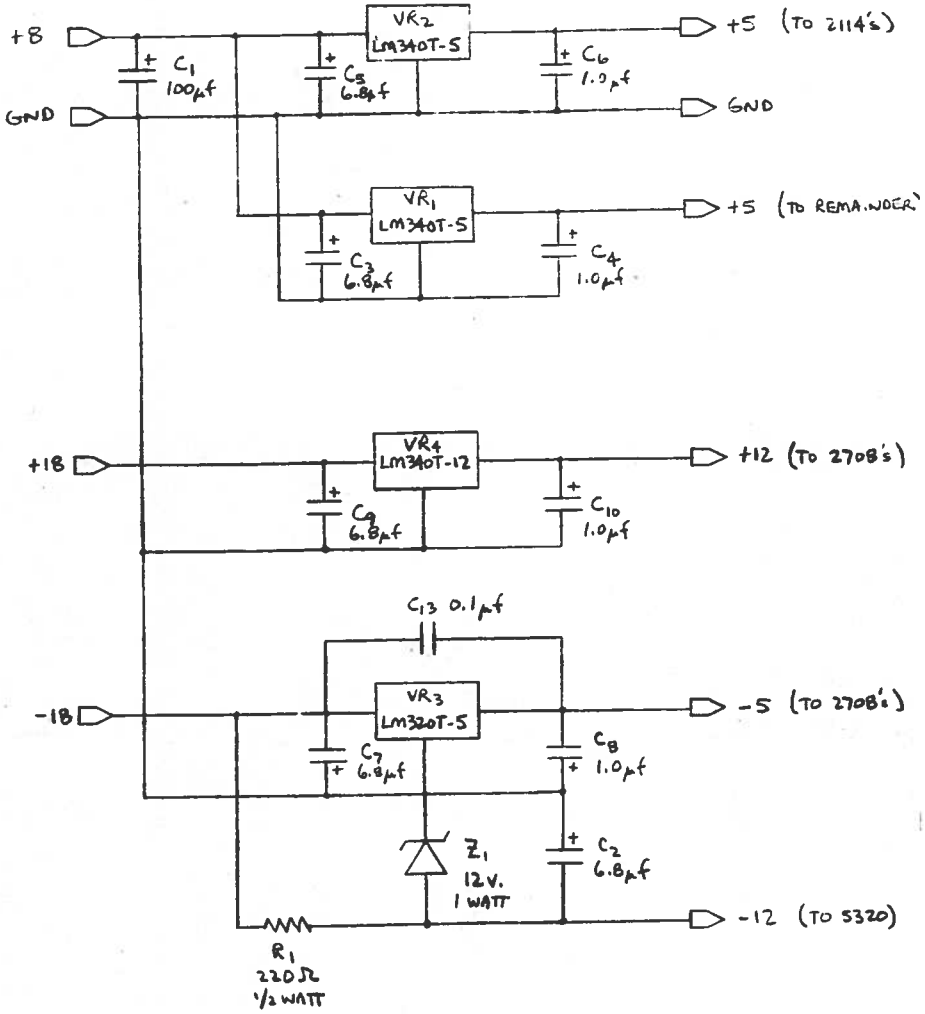
SCREENSPLITTER, VERSION 1A © 1978 MICRO DIVERSIONS, INC.

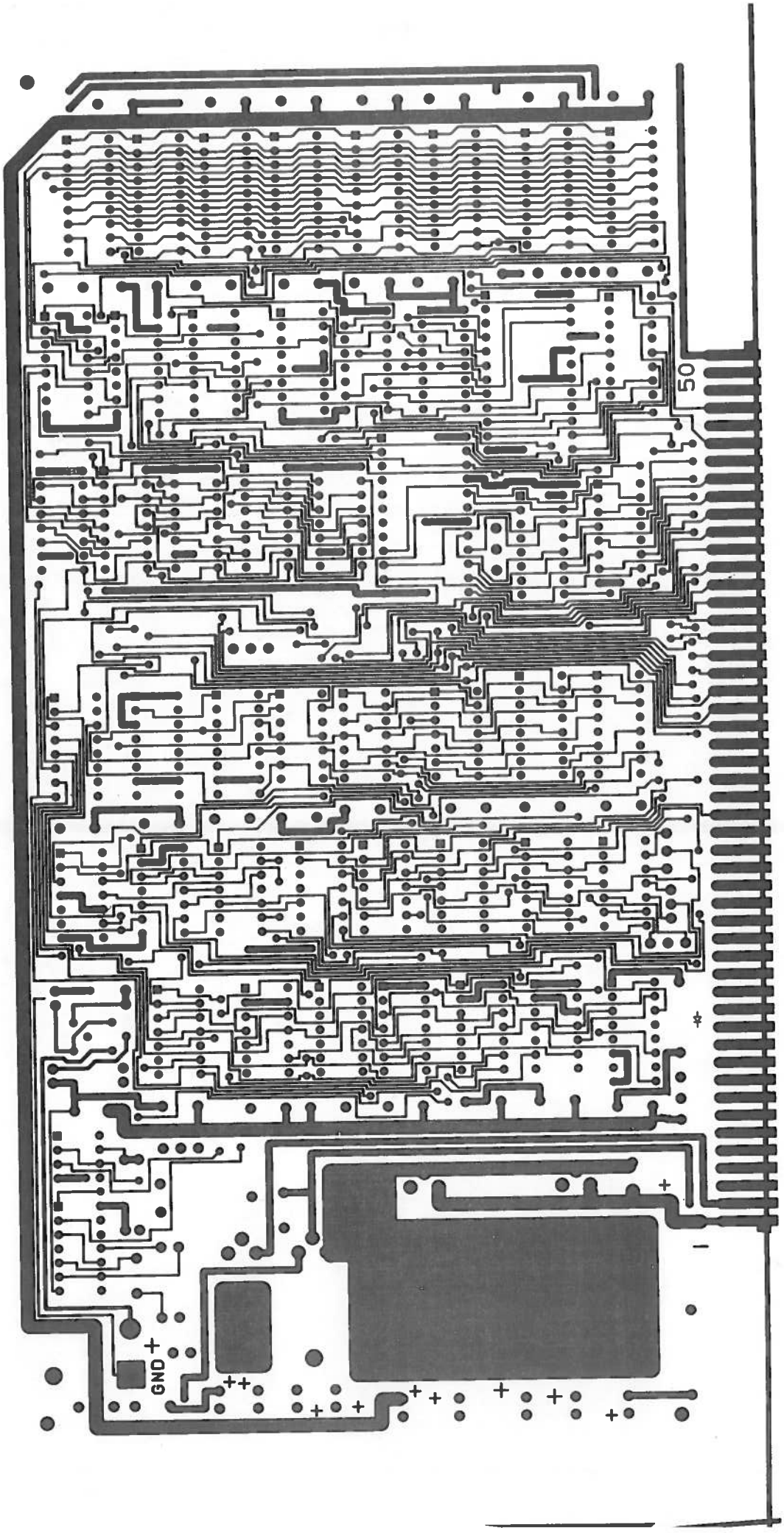


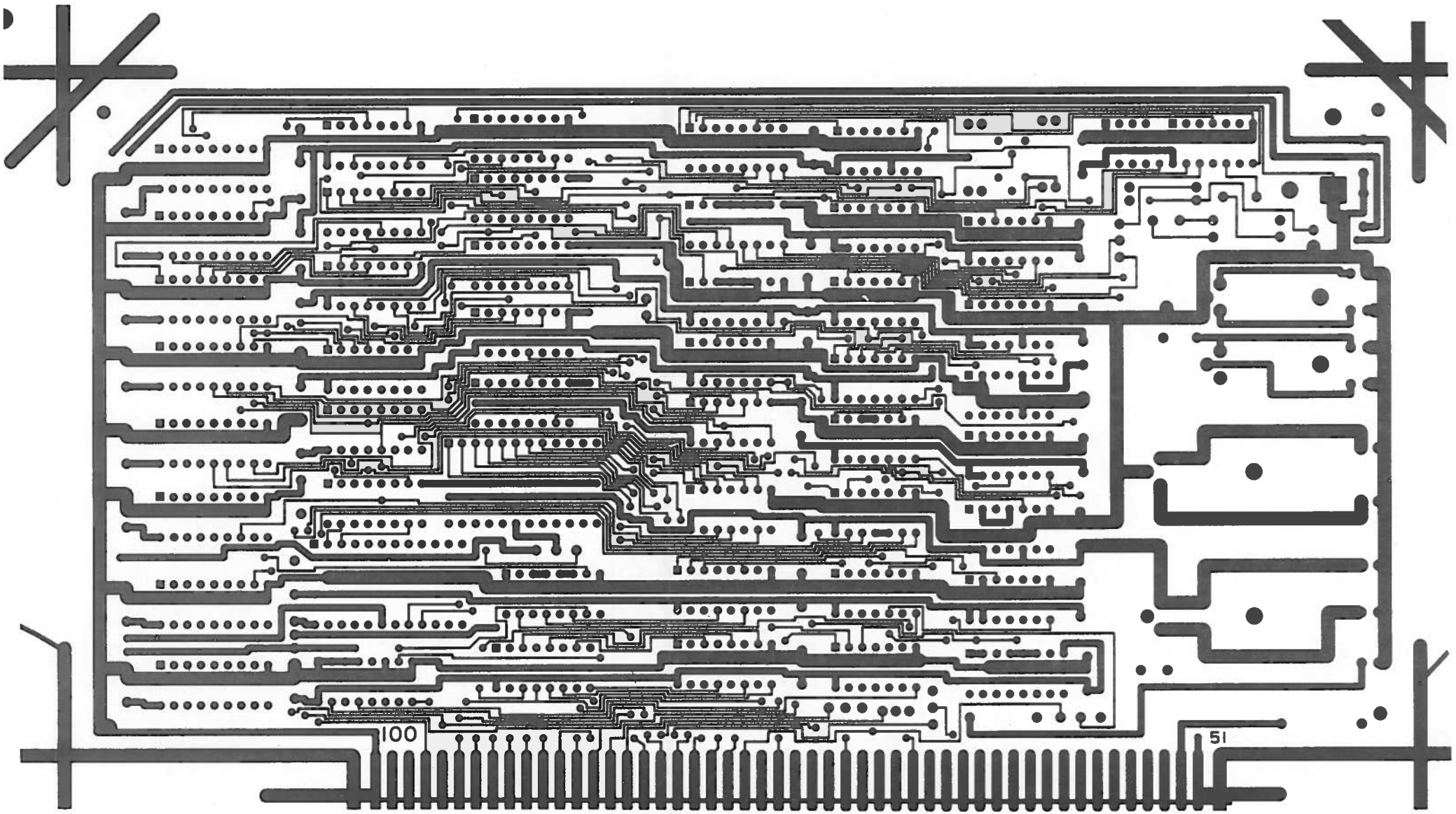

U40 74LS157
U39 74LS157
U38 74LS157
U37 74LS42
U1
HBLANK
U8 1 74LS74
U8 2 74LS74
U8 2 74LS74
SUPPRESS
SUPPRESS
LASTCOL
+5
© 1978 MICRODIVERSIONS, INC.
SCREENSPLITTER
PAGE 2/3
DISPLAY
MEMORY
* = ADDRESS PINS:
15,16,17,1,2,3,4,7,6,5
U48 2114
U47 2114
U46 2114
U45 2114
U41 2114
U42 2114
U43 2114
U44 2114
U22
U26
U26
U26
PDBIN
MWRITE
U12
U13
U13
U13
PSYNC
U19 74367
U18 74367
U21 74LS125
U20 74LS125
J1
U15
U16
U16
U12
U12
U12
VSELECT
ROMADDR
SWO
SOUT
SMEMR
U11
U15
U14
J2
J4
J3
U10
U10
U10
J9
PDB
PDBIN
MWRITE
VSELECT
PROY
A12
A13
A14
A15



SCREENSPLITTER
PAGE 3/3
PROGRAM MEMORY
&
POWER SUPPLY CIRCUITRY




50
GND


100
51

WINDOW PACKAGE SOURCE LISTING

```
1:      ; SCREENSPLITTER Window Package, from Micro Diversions
2:      ;
3:      ; (c) 1978 Micro Diversions, Inc.
4:      ;
5:      ;
6:      ; GENERAL DOCUMENTATION
7:      ;
8:      ; CONVENTIONS: The screen buffer is 4096 bytes of static RAM, known as
9:      ; SCREEN to the Window Package. SCRSIZE (4096) is the screen size, but only
10:     ; the first three-quarters is visible, with the rest unused. SCRWIDE (88)
11:     ; and SCRHIGH (40) define the height and width of the screen; VISWIDE (86)
12:     ; defines the visible width. (Two characters are lost in the hardware pipeline.)
13:     ; Screen rows are numbered 0-39 from top to
14:     ; bottom; screen columns 0-87 from left to right. A
15:     ; "screen address" is a 16 bit address that points to
16:     ; some byte in the visible screen in the host system's address space.
17:     ;
18:     ; A window's location on the screen is specified by naming the screen
19:     ; row of the window's first line and the screen column of the window's
20:     ; leftmost column. Within a window, lines are numbered starting from
21:     ; 0 (the window's first line), and likewise for the columns (the
22:     ; window's leftmost column is column 0). A window's vertical extent
23:     ; is specified by naming the window line number of the window's last
24:     ; (bottom) line. A window's horizontal extent is defined by naming the
25:     ; window column number of the window's last (rightmost) column.
26:     ;
27:     ; Windows are controlled by "window descriptor blocks" (WDB's), one WDB
28:     ; per open window on the screen. With the exception of INIT, all
29:     ; interactions with the Window Package must include (in the HL register) the
30:     ; pointer to the WDB of the window that is to be serviced.
31:     ;
32:     ; A WDB is 9 bytes long (with optional tenth and eleventh bytes), and has
33:     ; the following format:
34:     ;
35:     ;    BYTE 0: ST  (the window's status byte)
36:     ;    BYTE 1: SR  (the screen row number of the window's first line)
37:     ;    BYTE 2: CL  (the window line number of the window's cursor)
38:     ;    BYTE 3: LL  (the window line number of the window's last line)
39:     ;    BYTE 4: SC  (the screen column number of the window's left column)
40:     ;    BYTE 5: CC  (the window column number of the window's cursor)
41:     ;    BYTE 6: LC  (the window column number of the window's last column)
42:     ;    BYTE 7: CH  (the window's cursor character)
43:     ;    BYTE 8: SP  (the window's scroll parameter)
44:     ;    BYTE 9: HL  (optional, the low address byte of a hold processor)
45:     ;    BYTE 10: HH  (optional, the high address byte of a hold processor)
46:     ;
47:     ; A WDB status word contains the following information bits:
48:     ;
49:     ;    BIT 7: COMPLBIT     (complement bit for window's interior)
50:     ;    BIT 6: FRAMEBIT     (whether the window has a frame)
51:     ;    BIT 5: VCURBIT      (whether the window's cursor is to be visible)
52:     ;    BIT 4: CMODEBIT     (cursor mode bit, 0 for caret, 1 for fig. and. rev.)
53:     ;    BIT 3: HOLDBIT      (whether the window is to be held at scroll time)
54:     ;    BIT 2: OUTBIT       (set by user to denote output burst)
55:     ;    BIT 1: (unused)
56:     ;    BIT 0: (unused)
57:     ;
58:     ; The scroll parameter (SP) governs the type of scrolling that will
59:     ; occur when an output operation causes a window to overflow. SP=0
60:     ; causes one-line wraparound; SP greater than zero causes the window
61:     ; to pop up SP blank lines at overflow time.
62:     ;
63:     ; OPEN supplies the following default status when a window is first opened:
64:     ;
65:     ;     COMPLBIT: 0   (window is not complemented)
66:     ;     FRAMEBIT: 0   (window has no frame)
67:     ;     VCURBIT:  1   (cursor is visible)
68:     ;     CMODEBIT: 0   (CC is used as the cursor character)
69:     ;     HOLDBIT:  0   (the window is not to be held at scroll time)
70:     ;     OUTBIT:   0   (the window is not set for output bursts)
71:     ;
72:     ; Additionally, the default cursor character (the caret, 037 octal) is
```

```
73:       ; supplied, and the default scroll parameter (one-line pop-up) is set.
74:       ; The user-callable functions COMPL, FRAME and UNFRAME change the status
75:       ; of the COMPLBIT and FRAMEBIT. After OPENing a window, the user alters the
76:       ; VCURBIT, CMODEBIT, HOLDBIT, and OUTBIT by direct manipulation of the status
77:       ; byte. The scroll parameter is altered by calls to the function SCROLL.
78:       ; (SCROLL verifies only that the parameter is non-negative, deferring
79:       ; error checking for SP too large to the scrolling mechanism; if
80:       ; SP is too large, complete clearing of the window occurs at each scroll
81:       ; time.) The cursor character is altered by the user-callable function
82:       ; CURSORCH.
83:       ;
84:       ; The CMODEBIT governs the type of cursor control that occurs if the VCURBIT
85:       ; is on. CMODEBIT=0 causes the cursor character (CC) to be displayed at the
86:       ; cursor position on the window; CMODEBIT=1 causes the character at the cursor
87:       ; position to have its figure-ground reversed. (The latter mode is useful for
88:       ; text operations where it is not desirable for the cursor to occupy any
89:       ; space of its own.)
90:       ;
91:       ; WDB positions and extents always refer to the window's interior
92:       ; region; thus, when given a frame, SR, LL, SC, and LC are all adjusted;
93:       ; if the frame is removed, they are restored. To illustrate the WDB format,
94:       ; the following WDB would be created by OPEN in response to
95:       ;                   OPEN(1000,5,10,15,20)
96:       ; (assume all numbers except ST and CH are decimal):
97:       ;
98:       ;     1000: ST: 40 (octal)
99:       ;           SR: 5
100:      ;           CL: 0
101:      ;           LL: 9
102:      ;           SC: 15
103:      ;           CC: 0
104:      ;           LC: 19
105:      ;           CC: 37 (octal, the default caret)
106:      ;           SP: 1
107:      ;
108:      ; i.e., the WDB of this new window is now present at memory location
109:      ; 1000. All future transactions with this window will reference this WDB.
110:      ;
111:      ; If the hold status bit is set, bytes 9 and 10 of the WDB must be present,
112:      ; and must contain the address of a hold processor for the window. The
113:      ; hold processor will be called whenever the rolling or scrolling mechanism
114:      ; causes the window to roll or scroll during an output operation during
115:      ; which the (user-settable) OUTBIT is on, denoting output burst mode.
116:      ;
117:      ; WP initialization (INIT) causes the screen to be
118:      ; cleared to the specified character, and must be called once before
119:      ; the Window Package is used for window control.
120:      ;
121:      ;
122:      ; END OF GENERAL DOCUMENTATION
123:
124:
125:
126:            equ   caret,037q     default cursor character
127:            equ   blank,40q      ascii blank
128:            equ   strend,377q    string terminator character
129:            equ   dstatus,40q    default status byte for new windows
130:            equ   complbit,200q  status bit 7: window is complemented
131:            equ   framebit,100q  status bit 6: window has a frame
132:            equ   vcurbit,40q    status bit 5: window's cursor is visible
133:            equ   cmodebit,20q   status bit 4: cursor is via fig./gnd. reversal
134:            equ   holdbit,10q    status bit 3: window is to be held
135:            equ   outbit,4q      status bit 2: window is receiving output
136:            equ   holdout,14q    hold bit + output bit
137:            equ   nfrmbit,277q   frame bit complement
138:            equ   scrsize,4096   screen size
139:            equ   scrhigh,40     screen height in lines
140:            equ   scrwide,88     screen width in columns
141:            equ   viswide,86     visible screen width
142:            equ   nsw,177650q    negative of scrwide (1's complement)
143:            equ   nswmo,177647q  negative of scrwide minus one (1's complement)
144:            equ   neg1,177777q   negative one, 1's complement
145:            equ screen,10000q    this version assembled for block 0
146:
147:
148:      ; CURSOR (internal function) - display window's cursor
```

```
149:      ; ENTRY: (hl)= window descriptor block address
150:      ; EXIT: (no value returned)
151:      ; DESTROYS: all registers
152:      ; DESCRIPTION: displays the cursor for the specified window if the
153:      ;              visible-cursor bit in the window's status byte is on
154:
155:      cursor    mov   a,m           (a)= status byte
156:                ani   vcurbit       isolate visible-cursor bit
157:                rz                  invisible cursor, return
158:                push  hl            save WDB pointer
159:                call  cursorad      (hl)= cursor screen address
160:                xchg                (de)= cursor screen address
161:                pop   hl            restore WDB pointer
162:                mov   a,m           fetch status byte
163:                ani   cmodebit      isolate cursor mode bit
164:                call  up7           move to CC
165:                mov   a,m           (a)= cursor character
166:                jz    cur1          jump if caret type cursor (CMODEBIT=0)
167:                ldax  de            otherwise, fetch cursed char for fig./gnd. rev.
168:                xri   200q          reverse figure ground
169:      cur1      stax  de            store cursor
170:                ret                 return
171:
172:
173:      ; INIT (user entry point) - initialize Window Package
174:      ; ENTRY: (b)= screen blanking character
175:      ; EXIT: (no value returned)
176:      ; DESTROYS: all registers
177:      ; DESCRIPTION: clears display screen to the blanking character
178:
179:      init      lxi   hl,screen     clear visible screen to blanks
180:                lxi   de,scrsize    (de) is screen length
181:      ini1      mov   m,b           store next blanking character
182:                inx   hl
183:                dcx   de
184:                mov   a,d           test for end
185:                ora   e
186:                jnz   ini1
187:                ret                 return
188:
189:
190:      ; OPEN (user entry point) - open a window
191:      ; ENTRY: (hl)= window descriptor block address
192:      ;         (b)= screen row of window's top border
193:      ;         (c)= height of window, in rows
194:      ;         (d)= screen column of window's left column
195:      ;         (e)= width of window, in columns
196:      ; EXIT: (no value returned)
197:      ; DESTROYS: all registers
198:      ; DESCRIPTION: defines and opens a new window; does not affect the
199:      ;              visible display (e.g., window is not cleared)
200:
201:      open      call  okopen        ensure ok to open
202:                rnc                 return if problem
203:                mvi   m,dstatus     store default status
204:                inx   hl            move to SR
205:                mov   m,b           store screen row
206:                inx   hl            move to CL
207:                xra   a             set cursor line to zero
208:                mov   m,a
209:                inx   hl            move to LL
210:                mov   m,c           store last line index
211:                inx   hl            move to SC
212:                mov   m,d           store screen column
213:                inx   hl            move to CC
214:                mov   m,a           set cursor column to zero
215:                inx   hl            move to LC
216:                mov   m,e           store last column index
217:                inx   hl            move to CH
218:                mvi   m,caret       set default cursor character
219:                inx   hl            move to SP
220:                mvi   m,1           set default scroll parameter
221:                ret                 return to user
222:
223:
224:      ; UP3,...,9, DOWN3,...,8 (internal functions)
```

```
225:      ; ENTRY: (no entry conditions)
226:      ; EXIT: (hl) incremented or decremented
227:      ; DESTROYS: (no registers destroyed)
228:      ; DESCRIPTION: these functions are used for moving around within
229:      ;                a window descriptor block
230:
231:      up9       inx   hl
232:      up8       inx   hl
233:      up7       inx   hl
234:      up6       inx   hl
235:      up5       inx   hl
236:      up4       inx   hl
237:      up3       inx   hl
238:                inx   hl
239:                inx   hl
240:                ret
241:      down8     dcx   hl
242:      down7     dcx   hl
243:      down6     dcx   hl
244:      down5     dcx   hl
245:      down4     dcx   hl
246:      down3     dcx   hl
247:                dcx   hl
248:                dcx   hl
249:                ret
250:
251:
252:      ; CURSORAD (internal function) - cursor address
253:      ; ENTRY: (hl)= window descriptor block address
254:      ; EXIT: (hl)= absolute screen address of window's cursor
255:      ; DESTROYS: all registers
256:      ; DESCRIPTION: computes the absolute address of the cursor for the
257:      ;                specified window
258:
259:      cursorad  push  hl              save WDB pointer
260:                call  cursorlad       (hl)= address of first char of cur line
261:                xthl                  restore WDB, save cursor line address
262:                call  up5             move to window's CC
263:                mov   c,m             (c)= CC
264:                mvi   b,0             add CC to cursor line address
265:                pop   hl              restore cursor line address
266:                dad   bc              (hl)= cursor address
267:                ret
268:
269:
270:      ; CURSORLAD (internal function) - compute cursor line address
271:      ; ENTRY: (hl)= window descriptor block address
272:      ; EXIT: (hl)= absolute screen address of first character of window's
273:      ;               current cursor line
274:      ; DESTROYS: all registers
275:      ; DESCRIPTION: computes the absolute screen address of the first
276:      ;                character of the specified window's current cursor line
277:
278:      cursorlad inx   hl              get window's CL
279:                inx   hl
280:                mov   a,m             (a)= CL
281:                dcx   hl              restore hl
282:                dcx   hl
283:                call  lineaddr        convert to absolute address
284:                ret
285:
286:
287:      ; LINEADDR (internal function) - compute line address
288:      ; ENTRY: (hl)= window descriptor block address
289:      ;          (a)= relative line number
290:      ; EXIT: (hl)= absolute address of first character of the line
291:      ; DESTROYS: all registers
292:      ; DESCRIPTION: computes the absolute screen address of the first character
293:      ;                of the specified line of the specified window
294:      ;                NOTE: LINEADDR does a hard-wired multiply by 88, under
295:      ;                      the assumption that SCRWIDE= 88
296:
297:      lineaddr  push  hl              save WDB pointer
298:                inx   hl              access window's SR
299:                add   m               (a) now has screen line number
300:                mov   l,a             move multiplicand to hl
```

```
                mvi   h,0
                dad   hl              hl times 8
                dad   hl
                dad   hl
                mov   d,h             remember
                mov   e,l
                dad   hl              times 2
                mov   b,h             remember
                mov   c,l
                dad   hl              times 4
                dad   hl
                dad   bc              add powers 3,4 and 6
                dad   de              (hl)= (a) times 88
                lxi   de,screen       add in screen base address
                dad   de
                pop   de              (de)= WDB pointer
                xchg                  (hl)= WBD pointer, (de)= line addr
                call  up4             point at window's SC
                mov   l,m             add window's SC to line addr
                mvi   h,0
                dad   de              (hl)= address of line's first character
                ret


;  CLEARLIN (internal function) - clear a window line
;  ENTRY: (hl)= window descriptor block address
;         (a)= window line number
;  EXIT: (no value returned)
;  DESTROYS: all registers except hl
;  DESCRIPTION: clears the specified line of the specified window to
;               all blanks

clearlin  push  hl              save WDB pointer
          push  hl              save a copy of hl
          call  lineaddr        (hl) now points at line's first char
          xthl                  save addr, recall WDP pointer
          mov   a,m             get status byte
          ani   complbit        isolate complement bit
          ori   blank           (a)= clear character
          call  up6             access windgw's LC
          mov   b,m             (b)= window's LC
          pop   hl              (hl)= clear start address
cll1      mov   m,a             clear current column
          inx   hl              move to next column
          dcr   b               count down width
          jp    cll1            continue until done
          pop   hl              restore WDB pointer
          ret


;  CLEARCL (internal function) - clear window's cursor line
;  ENTRY: (hl)= window descriptor block address
;  EXIT: (no value returned)
;  DESTROYS: all registers except hl
;  DESCRIPTION: clears the cursor line of the specified window to
;               all blanks

clearcl   inx   hl              get cursor line
          inx   hl
          mov   a,m             (a)= CL
          dcx   hl              restore hl
          dcx   hl
          call  clearlin        clear line
          ret


;  CLEAR (user entry point) - clear window
;  ENTRY: (hl)= window descriptor block address
;  EXIT: (no exit conditions)
;  DESTROYS: all registers
;  DESCRIPTION: clears the interior of the specified window to
;               all blanks

clear     call  clr             clear window
          jmp   cursor          return to user
```

```
377:
378:      ; CLR (internal function) - clear a window
379:      ; ENTRY: (hl)= window descriptor block address
380:      ; EXIT: (no value returned)
381:      ; DESTROYS: all registers except hl
382:      ; DESCRIPTION: clears window's interior, resets cursor to top left
383:
384:      clr       push  hl          save WDB pointer
385:                inx   hl          move to window's CL
386:                inx   hl
387:                mvi   m,0         reset CL to zero
388:                inx   hl          move to window's LL
389:                mov   a,m         (a)= LL
390:                inx   hl          move to window's CC
391:                inx   hl
392:                mvi   m,0         reset CC to zero
393:                pop   hl          restore WDB pointer
394:      clr1      push  psw         save line counter in a
395:                call  clearlin    clear next line (bottom-to-top)
396:                pop   psw         restore line counter
397:                dcr   a           count down
398:                jp    clr1        continue through line zero
399:                ret               return
400:
401:
402:      ; CLEARLINE (user entry point) - clear and reset cursor line
403:      ; ENTRY: (hl)= window descriptor block address
404:      ; EXIT: (no value returned)
405:      ; DESTROYS: all registers
406:      ; DESCRIPTION: clears the current cursor line, and resets the cursor
407:      ;              to the left margin
408:
409:      clearline call  clearcl       clear cursor line
410:                push  hl            save WDB pointer
411:                call  up5           move to CC
412:                mvi   m,0           reset CC to zero
413:                pop   hl            restore WDB pointer
414:                jmp   cursor        return to user
415:
416:
417:      ; NEWLINE (internal function) - begin new output line
418:      ; ENTRY: (hl)= window descriptor block address
419:      ; EXIT: (no exit conditions)
420:      ; DESTROYS: all registers except hl
421:      ; DESCRIPTION: advances cursor line by one, rolling or scrolling if
422:      ;              necessary; ensures line is clear, and updates cursor
423:      ;              pointers
424:
425:      newline   push  hl            save WDB pointer
426:                inx   hl            move to window's CL
427:                inx   hl
428:                mov   a,m           (a)= current cursor line
429:                inr   m             increment CL
430:                inx   hl            move to window's LL
431:                cmp   m             test CL against LL
432:                pop   hl            restore WDB pointer
433:                cnc   glitch        off the bottom, roll or scroll
434:                push  hl            resave WDB pointer
435:                call  up5           move to window's CC
436:                xra   a             reset cursor column to zero
437:                mov   m,a
438:                call  up3           move to window's SP
439:                cmp   m             compare to zero
440:                pop   hl            restore WDB pointer
441:                cz    clearcl       in roll mode, clear new line
442:                ret
443:
444:
445:      ; GLITCH (internal function) - roll or scroll a window
446:      ; ENTRY: (hl)= window descriptor block address
447:      ; EXIT: (no exit conditions)
448:      ; DESTROYS: all registers except hl
449:      ; DESCRIPTION: if the window's scroll parameter is zero, wraparound
450:      ;              occurs; otherwise, the window's interior is popped up
451:      ;              the number of lines specified by SP (vacated lines are
452:      ;              blanked); CR is updated to point at the new cursor line
```

```
453:
454:    glitch  push   hl              save WDB pointer
455:            call   hold            hold the window if requested
456:            pop    hl              restore WDB pointer
457:            push   hl              resave it
458:            call   up8             move to window's SP
459:            mov    a,m             (a)= scroll parameter
460:            ora    a               set flags
461:            jnz    glt1            jump for scroll
462:            call   down6           it's a roll, move to CL
463:    glt0    xra    a               roll to line zero
464:            mov    m,a             reset CL to zero
465:            pop    hl              restore WDB pointer
466:            ret                    return
467:    glt1    mov    d,a             (d)= SP
468:            dcx    hl              scroll, move to window's LC
469:            dcx    hl
470:            mov    c,m             (c)= LC
471:            inr    c               (c)= window width
472:            call   down3           move to window's LL
473:            mov    a,m             (a)= LL
474:            inr    a               (a)= window height
475:            sub    d               (a)= height-SP
476:            mov    b,a             (b)= height-SP
477:            dcx    hl              move to window's CL
478:            mov    m,b             store new CL
479:            pop    hl              restore WDB pointer
480:            jm     clr             SP too large, clear entire window
481:            jz     clr             SP= window height, clear entire window
482:            push   hl              save WDB pointer
483:            push   de              save (d)= SP
484:            mov    a,m             (a)= status byte
485:            ani    complbit        isolate complement bit
486:            ori    blank           (a)= clear character
487:            push   psw             save clear character
488:            mov    a,d             (a)= SP again
489:            push   bc              save height-SP, width
490:            push   hl              make another copy of WDB
491:            call   lineaddr        (hl)= address of new first line
492:            xthl                   save address, restore WDB pointer
493:            xra    a               get address of window's top line
494:            call   lineaddr
495:            xchg                   (de)= address of first window line
496:            pop    hl              (hl)= address of new first line
497:            pop    bc              (b)= height-SP, (c)= window width
498:            pop    psw             restore clear character
499:            push   bc              save (b)= height-SP
500:            call   xymove          pop up
501:            pop    bc              restore (b)= height-SP
502:            pop    de              restore (d)= SP
503:            pop    hl              restore WDB
504:            mov    c,d             (b)= height-SP, (c)= SP
505:    glt2    push   bc              clear SP bottom lines
506:            mov    a,b             (a)= next line to clear
507:            call   clearlin        clear it
508:            pop    bc              restore (b)= height-SP, (c)= SP
509:            inr    b               increment window line
510:            dcr    c               decrement count
511:            jnz    glt2            continue until done
512:            ret                    return
513:
514:
515:    ; HOLD (internal function) - hold an output window if necessary
516:    ; ENTRY: (hl)= window descriptor block address
517:    ; EXIT: (no value returned)
518:    ; DESTROYS: all registers
519:    ; DESCRIPTION: if window's hold bit is on, a hold handler has been
520:    ;              defined, and an output operation is in progress,
521:    ;              calls the user-defined hold handler, then returns
522:
523:    hold    mov    a,m             get window's status
524:            ani    holdout         test both output and hold bit
525:            cpi    holdout
526:            rnz                    return if not both bits on
527:            call   up9             move to HH
528:            mov    e,m             load hold handler address into hl
```

```
                inx     h
                mov     d,m
                xchg
                pchl                    jump to user hold handler

;  XYMOVE (internal function) - move an XY region
;  ENTRY: (hl)= move start address
;         (de)= move destination address
;         (b)= height (low seven bits), direction (high bit)
;         (c)= width (low seven bits), direction (high bit)
;         (a)= clear character
;  EXIT:  (no value returned)
;  DESTROYS: all registers
;  DESCRIPTION:  moves XY region of given size; horizontal direction
;                bit: 0 - left-to-right, 1 - right-to-left; vertical
;                direction bit: 0 - top-to-bottom, 1 - bottom-to-top;
;                source region is cleared to the clear character

xymove  push    psw             save clear character
        push    bc              save height, width
        push    hl              save source, destination
        push    de
        mov     b,a             (b)= clear character
xym1    mov     a,m             move next byte
        mov     m,b             store clear character
        stax    de              store moved character
        dcr     c               count down width
        inx     hl              increment for left-to-right
        inx     de
        jp      xym2
        dcx     hl              decrement for right-to-left
        dcx     hl
        dcx     de
        dcx     de
xym2    mvi     a,177q          remove direction bit from width
        ana     c               (c)= true width counter
        jnz     xym1            continue until across
        pop     de              finished current line, restore pointers
        pop     hl
        pop     bc              restore counters
        dcr     b               count down line counter
        push    bc              save counters a moment
        lxi     bc,screwide     moving top-to-bottom
        jp      xym3
        lxi     bc,nsw          bottom-to-top
xym3    dad     bc              (bc)= vertical increment, advance lines
        xchg                    in both source and destination
        dad     bc
        xchg
        pop     bc              restore counters
        mvi     a,177q          isolate true vertical counter
        ana     b
        xthl                    restore clear character
        mov     a,h             (a)= clear character
        pop     hl              restore source pointer
        jnz     xymove          continue until done
        ret                     return

;  PRIN,PRINS,PRINT,PRINTS (user entry points) - print routines
;  ENTRY: (hl)= window descriptor block address
;         (bc)= string pointer
;         (de)= string length (PRIN and PRINT only)
;  EXIT:  (no value returned)
;  DESTROYS: all registers except hl
;  DESCRIPTION:  PRIN and PRINT explicitly give the string length in
;                de, and require no string terminator character; PRINS
;                and PRINTS give only the string pointer to a string
;                which is terminated by the ascii character STREND;
;                PRIN and PRINS leave the cursor at the point at which
;                the print ends (i.e., the cursor is not forced to the
;                beginning of a fresh line after the print); PRINT and
;                PRINTS force the cursor to the beginning of a fresh
;                line after the print
```

```
605:      prins     call  slength        get string length to de
606:                ora   a              clear carry flag
607:                jmp   prt            go print
608:      prints    call  slength        get string length to de
609:                stc                  set carry for fresh line after print
610:                jmp   prt
611:      prin      ora   a              clear carry
612:                jmp   prt            go print
613:      print     stc                  set carry for fresh line after print
614:
615:      prt       push  psw            save carry for prin exit
616:      prt1      mov   a,d            test for end of string
617:                ora   e
618:                jz    prt2           done, go exit
619:                ldax  bc             (a)= next character to print
620:                inx   bc             advance string pointer
621:                dcx   de             count down length
622:                push  bc             save pointer, count
623:                push  de
624:                call  pri            print the character
625:                pop   de             restore pointer, count
626:                pop   bc
627:                jmp   prt1           continue
628:      prt2      pop   psw            restore carry, set at beginning
629:                cc    frline         force new line if carry set
630:                jmp   cursor         return to user
631:
632:
633:      ; PRI (internal function) - print a character to a window
634:      ; ENTRY: (hl)= window descriptor block pointer
635:      ;        (a)= character
636:      ; EXIT: (no value returned)
637:      ; DESTROYS: all registers except hl
638:      ; DESCRIPTION: Prints the character in the window, complementing it
639:      ;              if the window's complement bit is set; advances the
640:      ;              cursor, scrolling if necessary.
641:
642:      pri       call  cplt           print the character
643:                push  hl             save WDB pointer
644:                call  up5            move to window's CC
645:                mov   a,m            (a)= CC
646:                inr   m              increment CC
647:                inx   hl             move to window's LC
648:                cmp   m              compare CC to LC
649:                pop   hl             restore WDB pointer
650:                cnc   newline        if CC=LC, line exhausted, start new line
651:                ret                  return
652:
653:
654:      ; CPLT, PLT (internal functions) - write a character to a window
655:      ; ENTRY: (hl)= window descriptor block address
656:      ;        (b)= window line number (PLT only)
657:      ;        (c)= window column number (PLT only)
658:      ;        (d)= character (PLT only)
659:      ;        (a)= character (CPLT only)
660:      ; EXIT: (no value returned)
661:      ; DESTROYS: all registers except hl
662:      ; DESCRIPTION: prints the character to the window, observing the
663:      ;              figure-ground status of the window; CPLT prints
664:      ;              the character in the a register at the current
665:      ;              cursor position
666:
667:      cplt      push  hl             save WDB pointer
668:                inx   hl             move to CL
669:                inx   hl
670:                mov   b,m            (b)= CL
671:                call  up3            move to CC
672:                mov   c,m            (c)= CC
673:                pop   hl             restore WDB pointer
674:                mov   d,a            (d)= character
675:
676:      plt       mov   a,b            (a)= line number
677:                push  hl             save registers
678:                push  bc
679:                push  de
680:                call  lineaddr       (hl)= line address of print
```

```
                pop   bc              (b)= character
                pop   de              (e)= column number of print
                mvi   d,0             (de)= column number
                dad   de              (hl)= print address
                xchg                  (de)= print address
                pop   hl              (hl)= WDB pointer
                mov   a,m             (a)= status byte
                ani   complbit        isolate complement bit
                xra   b               complement character if necessary
                stax  de              print
                ret                   return


; SCROLL (user entry point) - set a window's scroll parameter
; ENTRY: (hl)= window descriptor block address
;        (b)= scroll parameter
; EXIT: (no value returned)
; DESTROYS: all registers
; DESCRIPTION: sets the scroll parameter of the specified window;
;              zero denotes rolling, greater than zero denotes scrolling

scroll    mov   a,b             (a)= new scroll parameter
          ora   a               set flags
          rm                    new parameter is negative, error
          call  up8             move to SP
          mov   m,a             store scroll parameter
          ret                   return to user


; FRAME (user entry point) - frame a window
; ENTRY: (hl)= window descriptor block address
;        (b)= horizontal border character
;        (c)= vertical border character
;        (d)= corner character
; EXIT: (no value returned)
; DESTROYS: all registers except hl
; DESCRIPTION: draws a frame around the named window, using the three
;              specified characters; narrows the print region by
;              two in each dimension (error return if this is not
;              possible) if frame not already present, and turns on the
;              frame bit in the window's status word

frame     mov   a,m             get status word
          ani   framebit        test frame bit
          jnz   frm1            frame already present, no narrowing
          call  takein          reduce window interior by two both ways
          rm                    return if impossible
          mov   a,m             turn on frame bit
          ori   framebit
          mov   m,a
          push  bc              save frame characters
          push  de
          call  clr             clear window, reset cursor
          pop   de              restore frame characters
          pop   bc
frm1      push  hl              save WDB pointer
          mov   a,d             (a)= corner character
          call  frm2            set up frame parameters on the stack
          call  frm2            set up duplicate parameters
          push  psw             save corner character a moment
          xra   a               compute top left window char address
          call  lineaddr        (hl)= top left char address
          lxi   de,nswmo        move to top left frame corner
          dad   de              (hl)= top left frame corner address
          pop   psw             restore corner character
          lxi   bc,scrwide      draw left border
          call  frm3
          lxi   bc,1            draw bottom border
          call  frm3
          lxi   bc,nsw          draw right border
          call  frm3
          lxi   bc,neg1         draw top border
          call  frm3
          pop   hl              restore WDB pointer
          jmp   cursor          return to user
```

```
757:      frm2      call   up6             move to LC
758:                mov    d,m             (d)= LC
759:                mov    e,b             (e)= horiz character
760:                xchg                   (de)= WDB pointer, (hl)= frame info
761:                xthl                   stack frame info, get return addr
762:                push   hl              restore return address
763:                xchg                   (hl)= WDB pointer
764:                call   down3            move to LL
765:                mov    d,m             repeat for vertical parameters
766:                mov    e,c
767:                xchg
768:                xthl
769:                push   hl
770:                xchg
771:                call   down3
772:                ret                     return
773:
774:      frm3      pop    de              unstack return address a moment
775:                xchg                   (hl)= ret addr, (de)= border address
776:                xthl                   restack ret addr, (hl)= next frame segment
777:                xchg                   (de)= frame info, (hl)= border address
778:      frm4      dad    bc              move to next border position
779:                mov    m,e             store border character
780:                dcr    d               count down
781:                jp     frm4            continue until done
782:                dad    bc              move to corner point
783:                mov    m,a             store corner character
784:                ret                    return
785:
786:
787:      ; TAKEIN (internal function) - reduce window interior for frame
788:      ; ENTRY: (hl)= window descriptor block address
789:      ; EXIT: failure: sign flag set minus, success: sign flag set positive
790:      ; DESTROYS: a,e
791:      ; DESCRIPTION: reduces dimensions of window by two if such a reduction
792:      ;              leaves at least one character interior; otherwise does
793:      ;              nothing; leaves sign flag set appropriately
794:
795:      takein    call   up3             ensure wind big enough
796:                mov    a,m             (a)= LL
797:                sbi    2               height must be at least 3
798:                rm                     it's not, error
799:                call   up3             move to window's LC
800:                mov    e,m             (e)= LC
801:                dcr    e               check that window is at least 3 wide
802:                dcr    e
803:                rm                     it's not, error
804:                mov    m,e             window size ok, store new width
805:                dcx    hl              move to window's SC
806:                dcx    hl
807:                inr    m               increase SC
808:                dcx    hl              move to window's LL
809:                mov    m,a             store new LL
810:                dcx    hl              move to window's SR
811:                dcx    hl
812:                inr    m               increase SR
813:                dcx    hl              move to window's status word
814:                ret                    return
815:
816:
817:      ; UNFRAME (user entry point) - remove a window's frame
818:      ; ENTRY: (hl)= window descriptor block address
819:      ; EXIT: (no value returned)
820:      ; DESTROYS: all registers except hl
821:      ; DESCRIPTION: removes frame if present, turns off frame bit, and
822:      ;              clears window if frame actually removed
823:
824:      unframe   call   letout          increase window dimensions by two if framed
825:                mov    a,m             (a)= status byte
826:                ani    nfrmbit         turn off frame bit
827:                mov    m,a             replace status byte
828:                call   clr             clear and reset window
829:                jmp    cursor          return to user
830:
831:
832:      ; LETOUT (internal function) - increase window size to include frame
```

```
833:      ; ENTRY: (hl)= window descriptor block address
834:      ; EXIT: (no value returned)
835:      ; DESTROYS: a
836:      ; DESCRIPTION: increases window dimensions by two if frame bit is on;
837:      ;              otherwise does nothing
838:
839:      letout    mov   a,m             (a)= status byte
840:                ani   framebit        isolate frame bit
841:                rz                    no frame, return
842:                push  hl              save WDB pointer
843:                mvi   a,1             do the following twice
844:      tko1      inx   hl              move to SR (or SC on second pass)
845:                dcr   m               let out left (or top) border one char
846:                inx   hl              move to LL (or LC)
847:                inx   hl
848:                inr   m               increase LL (or LC) by two
849:                inr   m
850:                dcr   a               do again for horizontal parameters
851:                jz    tko1
852:                pop   hl              restore WDB pointer
853:                ret                   return
854:
855:
856:      ; LABELS, LABEL (user entry points) - label a window
857:      ; ENTRY: (hl)= window descriptor block address
858:      ;        (bc)= string label pointer
859:      ;        (de)= string length (LABEL only)
860:      ; EXIT: (no value returned)
861:      ; DESTROYS: all registers
862:      ; DESCRIPTION: causes the string pointed at by bc to be written
863:      ;              to the specified window's top border; if the window
864:      ;              has no frame, if the string is of zero length, or
865:      ;              if the string is too long, no action is taken; LABELS
866:      ;              supplies only the string pointer, and terminates the
867:      ;              string with the STREND character; LABEL supplies the
868:      ;              string's length in de, and needs no STREND character
869:
870:      labels    call  slength         compute label length to de
871:      label     mov   a,m             (a)= window's status byte
872:                ani   framebit        make sure window has a frame
873:                rz                    return if no frame
874:                mov   a,d             (a)= high byte of length
875:                ora   a               test for too long
876:                rnz                   return if non-zero (too large)
877:                add   e               (a)= label length
878:                rz                    null label, return
879:                call  up6             move to LC
880:                mov   a,m             (a)= LC
881:                adi   3               (a)= available width for label
882:                sub   e               (a)= window width less label length
883:                rc                    return if label too long
884:                call  down6           move back to status byte
885:                push  bc              save label pointer
886:                push  de              save label length
887:                push  psw             save difference in widths
888:                xra   a               get address of window's first line
889:                call  lineaddr        (hl)= addr of top left interior char
890:                pop   psw             restore (a)= difference in width
891:                rar                   divide by 2 (cy flag zero at this point)
892:                adi   nswmo           (a)= amount to decrement (hl)
893:                mov   c,a             add to top left addr for border point
894:                mvi   b,377q          (bc)= decrement for (hl)
895:                dad   bc              (hl)=  label start addr on top border
896:                pop   de              restore label length to e
897:                pop   bc              restore label pointer
898:      lab1      ldax  bc              copy label to border
899:                mov   m,a             store next character
900:                inx   bc              advance to next character
901:                inx   hl
902:                dcr   e               count down length
903:                jnz   lab1            continue until done
904:                ret                   return to user
905:
906:
907:      ; SLENGTH (internal function) - compute length of string
908:      ; ENTRY: (bc)= string pointer
```

```
909:      ; EXIT: (de)= string length
910:      ; DESTROYS: a
911:      ; DESCRIPTION: searches for the STREND character in the string and
912:      ;              returns the length of the string; if STREND character
913:      ;              not found after 65k bytes, 0 is returned
914:
915:      slength   push  bc            save bc
916:                lxi   de,0          de will count length
917:      sln1      ldax  bc            (a)= next string character
918:                cpi   strend        test
919:                jz    sln2          found the end, go exit
920:                inx   bc            not end, advance pointer and count
921:                inx   de
922:                mov   a,d           test for 65k wraparound
923:                ora   e
924:                jnz   sln1          not wraparound yet, continue search
925:      sln2      pop   bc            restore bc
926:                ret                 return
927:
928:
929:      ; CURSORCH (user entry point) - set a window's cursor character
930:      ; ENTRY: (hl)= window descriptor block address
931:      ;         (b)= cursor character
932:      ; EXIT: (no value returned)
933:      ; DESTROYS: all registers
934:      ; DESCRIPTION: sets the specified window's cursor character to the
935:      ;              given character
936:
937:      cursorch  mov   a,m           (a)= status byte
938:                ani   complbit      isolate complement bit
939:                xra   b             complement new CC if required
940:                push  hl            save WDB pointer
941:                call  up7           move to CH
942:                mov   m,a           store new character
943:                pop   hl            restore WDB pointer
944:                jmp   cursor        return to user
945:
946:
947:      ; COMPL (user entry point) - complement interior of window
948:      ; ENTRY: (hl)= window descriptor block address
949:      ; EXIT: (no value returned)
950:      ; DESTROYS: all registers
951:      ; DESCRIPTION: toggles window's complement bit, toggles figure-ground
952:      ;              of window's interior, and complements cursor character
953:
954:      compl     mov   a,m           toggle complement bit in status word
955:                xri   complbit
956:                mov   m,a
957:      cpl0      push  hl            save WDB pointer (cpl0 called from flash)
958:                call  up3           move to LL
959:                mov   b,m           (b)= LL
960:                call  up3           move to LC
961:                mov   c,m           (c)= LC
962:                inx   hl            move to cursor character
963:                mov   a,m           complement cursor character
964:                xri   complbit
965:                mov   m,a
966:                pop   hl            restore WDB pointer
967:                push  bc            save LL, LC
968:                xra   a             get address of top left interior char
969:                call  lineaddr      (hl)= top left char address
970:                pop   bc            restore LL, LC
971:                lxi   de,scruide    (de) will increment over lines
972:      cpl1      push  bc            complement next window line
973:                push  hl            save line address, LL, LC counters
974:      cpl2      mov   a,m           (a)= next window character
975:                xri   complbit      complement it
976:                mov   m,a           replace it
977:                inx   hl            advance to next character
978:                dcr   c             count down width
979:                jp    cpl2          until negative
980:                pop   hl            reset to beginning of this line
981:                dad   de            advance to next line (add 88)
982:                pop   bc            restore LL, LC
983:                dcr   b             count down line counter
984:                jp    cpl1          until done
```

```
                ret                     return
;
; FLASH (user entry point) - flash a window's interior
; ENTRY: (hl)= window descriptor block address
; EXIT: (no value returned)
; DESTROYS: all registers
; DESCRIPTION: causes the figure-ground of the specified window to
;               be reversed, then restored in about two fifths of a second
;               (2 mhz cpu)
;
flash   push    hl              save WDB
        call    cpl0            complement the window
        call    delay           wait about a fifth of a second
        pop     hl              restore WDB pointer
        call    cpl0            restore the window
        call    delay           wait another fifth of a second
        ret                     return to user
;
delay   lxi     bc,40000q       wait a while
dly1    dcx     bc              count down to zero
        mov     a,b
        ora     c
        jnz     dly1
        ret                     return
;
; PLOT (user entry point) - plot a character in a window
; ENTRY: (hl)= window descriptor block address
;        (b)= line number
;        (c)= column number
;        (d)= character to be plotted
; EXIT: (no value returned)
; DESTROYS: all registers
; DESCRIPTION: plots the character at the specified coordinates if
;               in bounds; does nothing if out of bounds; does not
;               affect the window's cursor
;
plot    xra     a               clear a
        add     b               (b)= plot line number
        rm                      return if negative
        call    up3             move to LL
        mov     a,m             (a)= LL
        cmp     b               test plot line for too large
        rc                      return if too large
        xra     a               clear a
        add     c               (a)= plot column number
        rm                      return if negative
        call    up3             move to LC
        mov     a,m             (a)= LC
        cmp     c               test plot column for too large
        rc                      return if too large
        call    down6           restore WDB pointer
        jmp     plt             plot character and return
;
; BACKSPACE (user entry point) - back the cursor up one space
; ENTRY: (hl)= window descriptor block address
; EXIT: (no value returned)
; DESTROYS: all registers
; DESCRIPTION: if cursor not already at leftmost column, backs it up
;               one column
;
backspace mvi   a,blank         clear current cursor
        call    cplt
        push    hl              save WDB pointer
        call    up5             move to CC
        mov     a,m             (a)= CC
        dcr     a               back up
        jm      bsp1            if already at left border
        mov     m,a             store new CC
bsp1    pop     hl              restore WDB pointer
        jmp     cursor          exit
;
; FRESHLINE (user entry point) - ensure cursor begins fresh line
```

```
1061:      ; ENTRY: (hl)= window descriptor block pointer
1062:      ; EXIT: (no value returned)
1063:      ; DESTROYS: all registers
1064:      ; DESCRIPTION: if the window's cursor is not at the left margin,
1065:      ;              a new line is begun
1066:
1067:      freshline call  frline          get fresh line
1068:                jmp   cursor          return to user
1069:
1070:      frline    push  hl              save WDB pointer
1071:                call  up5             move to CC
1072:                xra   a               test for zero
1073:                cmp   m
1074:                pop   hl              restore WDB pointer
1075:                rz                    return if already at left border
1076:                mvi   a,blank         erase current cursor (if present)
1077:                call  cplt
1078:                call  newline         move to new line
1079:                ret                   return
1080:
1081:
1082:      ; MOVWIN (user entry point) - move a window
1083:      ; ENTRY: (hl)= window descriptor block address
1084:      ;        (b)= new screen row
1085:      ;        (c)= new screen column
1086:      ;        (d)= clear character to replace moved window
1087:      ; EXIT: (no value returned)
1088:      ; DESTROYS: all registers
1089:      ; DESCRIPTION: if the window would remain entirely on the screen, the
1090:      ;              window is moved so that its top left character (frame
1091:      ;              included, if present) is situated on the specified screen
1092:      ;              row and column; if the move would shift any part of the
1093:      ;              window off the screen, no action is taken; the specified
1094:      ;              clear character (usually a blank, or a complemented
1095:      ;              blank) replaces any vacated region on the screen
1096:
1097:      movwin    call  letout          include the frame, if present
1098:                push  de              save clear character
1099:                push  hl              save WDB pointer
1100:                mov   d,c             set up call to open to check validity of move
1101:                call  up3             move to LL
1102:                mov   c,m             (c)= LL
1103:                inr   c               (b)= requested new screen row, (c)= height
1104:                call  up3             move to LC
1105:                mov   e,m             (e)= LC
1106:                inr   e               (d)= requested new screen col, (e)= width
1107:                pop   hl              restore WDB pointer
1108:                call  ok&open         ensure new position is ok
1109:                mov   c,d             restore proposed SC to c
1110:                pop   de              restore clear character to d
1111:                jnc   mvw0            window would be off screen, error
1112:                push  hl              save WDB pointer
1113:                push  de              save move clear character
1114:                push  hl              save another copy of WDB
1115:                mov   a,b             (a)= new screen row number
1116:                call  mvw1            compute vertical move direction, start row
1117:                mov   d,a             (d)= vert move dir, starting row of move
1118:                mov   a,c             do same for horizontal
1119:                call  mvw1
1120:                mov   e,a             (e)= vert move dir, starting column of move
1121:                pop   hl              restore WDB pointer
1122:                push  hl              resave WDB
1123:                call  mvw2            compute corner address of move start
1124:                xthl                  save source address, restore WDB pointer
1125:                push  hl              save another copy of WDB pointer
1126:                inx   hl              move to SR
1127:                mov   m,b             store new screen row
1128:                inx   hl              move to LL
1129:                inx   hl
1130:                mov   b,m             (b)= LL
1131:                inr   b               (b)= window height
1132:                mov   a,d             or in vertical move direction bit to height
1133:                ani   200q
1134:                ora   b
1135:                mov   b,a             (b)= move line count, vert direction bit
1136:                inx   hl              move to SC
```

```
1137:                   mov     m,c             store new SC
1138:                   inx     hl              move to LC
1139:                   inx     hl
1140:                   mov     c,m             (c)= LC
1141:                   inr     c               (c)= window width
1142:                   mov     a,e             or in horizontal move direction bit to width
1143:                   ani     200q
1144:                   ora     c
1145:                   mov     c,a             (c)= move column count, horiz direction bit
1146:                   pop     hl              restore WDB pointer
1147:                   call    mvw2            compute start (corner) address of destination
1148:                   xchg                    (de)= destination start address
1149:                   pop     hl              (hl)= source start address
1150:                   pop     psw             restore clear character
1151:                   call    xymove          move it
1152:                   pop     hl              restore WDB pointer
1153:   mvw0            mov     a,m             (a)= status byte
1154:                   ani     framebit        isolate frame bit
1155:                   cnz     takein          if frame present, take size in again
1156:                   ret                     return to user
1157:
1158:   mvw1            inx     hl              move to old SR (or SC)
1159:                   sub     m               subtract from proposed SR (or SC)
1160:                   cma                     bit 7 is now move direction (0 up, 1 down)
1161:                   ani     200q            isolate move direction bit
1162:                   inx     hl              move to LL (or LC)
1163:                   inx     hl
1164:                   rp                      if upward move, start with line 0, return
1165:                   ora     m               downward move, start with last line of window
1166:                   ret                     return
1167:
1168:   mvw2            mvi     a,177q          compute corner address of source (or dest)
1169:                   ana     d               (a)= window line number of start
1170:                   push    bc              protect registers
1171:                   push    de
1172:                   call    lineaddr        (hl)= line address
1173:                   pop     de              restore line, column numbers of move start
1174:                   mvi     a,177q          isolate col number of move start
1175:                   ana     e
1176:                   mov     c,a             (c)= column number of move start
1177:                   mvi     b,0             (bc)= column number
1178:                   dad     bc              (hl)= start address of move source (or dest)
1179:                   pop     bc              restore bc
1180:                   ret                     return
1181:
1182:
1183:   ; OKOPEN (internal function) - test whether window is within screen
1184:   ; ENTRY: (b)= proposed screen row
1185:   ;        (c)= proposed height
1186:   ;        (d)= proposed screen column
1187:   ;        (e)= proposed width
1188:   ; EXIT: carry flag set on success, cleared on failure
1189:   ; DESTROYS: a
1190:   ; DESCRIPTION: checks that proposed window will lie entirely on the
1191:   ;              visible screen
1192:
1193:   okopen          push    hl              save hl a moment
1194:                   lxi     hl,oko1         push local error return address
1195:                   xthl                    restore hl
1196:                   mov     a,b             (a)= screen row of window's top border
1197:                   ora     a               set flags
1198:                   rm                      negative row number, return
1199:                   cpi     scrhigh
1200:                   rnc                     too large, return
1201:                   dcr     c               (c)= relative index of last row
1202:                   rm                      negative, return
1203:                   mov     a,c             ensure that window fits vertically
1204:                   add     b               (a)= last row screen index
1205:                   cpi     scrhigh
1206:                   rnc                     off bottom of screen, return
1207:                   mov     a,d             (a)= window's leftmost col. screen pos.
1208:                   ora     a               certify it
1209:                   rm                      negative, return
1210:                   cpi     viswide
1211:                   rnc                     too large, return
1212:                   dcr     e               (e)= relative index of last col
```

```
1213:                   rm                      negative, return
1214:                   mov     a,e             ensure that window fits horizontally
1215:                   add     d               (d)= last column screen index
1216:                   cpi     viswide
1217:                   rnc                     off right edge of screen, return
1218:                   xthl                    success, clear off local error return address
1219:                   pop     hl
1220:                   ret                     return to caller
1221:   oko1            ora     a               error, clear carry flag
1222:                   ret                     return
1223:
1224:   nil                                     end of Window Package
```

WINDOW PACKAGE SYMBOL TABLE (RELATIVE TO LOCATION 0)

	Symbol	Page	Byte
	CURSOR	0	000
	CURSORAD	0	125
	UP7	0	104
	CUR1	0	027
**	INIT	0	031
	INI1	0	037
**	OPEN	0	050
	OKOPEN	3	326
	UP9	0	102
	UP8	0	103
	UP6	0	105
	UP5	0	106
	UP4	0	107
	UP3	0	110
	DOWN8	0	114
	DOWN7	0	115
	DOWN6	0	116
	DOWN5	0	117
	DOWN4	0	120
	DOWN3	0	121
	CURSORLAD	0	143
	LINEADDR	0	154
	CLEARLIN	0	214
	CLL1	0	234
	CLEARCL	0	244
**	CLEAR	0	255
	CLR	0	263
	CLR1	0	277
**	CLEARLINE	0	311
	NEWLINE	0	326
	GLITCH	0	360
	HOLD	1	105
	GLT1	1	005
	GLT0	1	001
	XYMOVE	1	123
	GLT2	1	071
	XYM1	1	130
	XYM2	1	145
	XYM3	1	171
**	PRINS	1	210
	SLENGTH	2	273
	PRT	1	233
**	PRINTS	1	217
**	PRIN	1	226
**	PRINT	1	232
	PRT1	1	234
	PRT2	1	256
	PRI	1	265
	FRLINE	3	124
	CPLT	1	305
	PLT	1	317
**	SCROLL	1	343
**	FRAME	1	353
	FRM1	2	000
	TAKEIN	2	121
	FRM2	2	056
	FRM3	2	104
	FRM4	2	110
**	UNFRAME	2	152
	LETOUT	2	167
	TKO1	2	176
**	LABELS	2	212
**	LABEL	2	215
	LAB1	2	262
	SLN1	2	277

** Denotes user entry point.

```
     SLN2          2       314
**   CURSORCH      2       316
**   COMPL         2       333
     CPL0          2       337
     CPL1          2       367
     CPL2          2       371
**   FLASH         3       012
     DELAY         3       031
     DLY1          3       034
**   PLOT          3       043
**   BACKSPACE     3       073
     BSP1          3       112
**   FRESHLINE     3       116
**   MOVWIN        3       145
     MVW0          3       263
     MVW1          3       272
     MVW2          3       304
     OKO1          3       370
```

(Relative to location 0)

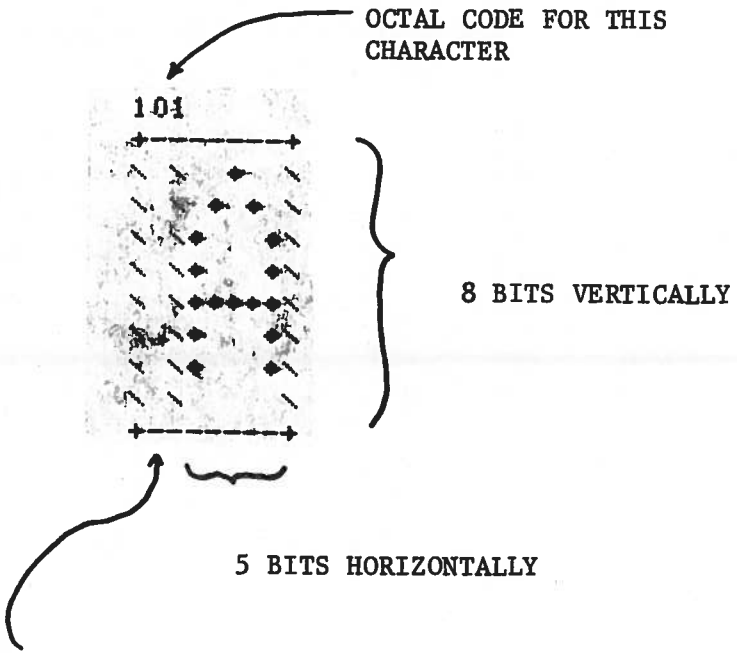
**n denotes page address. Add your 8K page boundary to n for the actual address.

sss (on lines 000 020 and 000 200) are the page address of the display buffer. Substitute as appropriate for your 8K boundary.

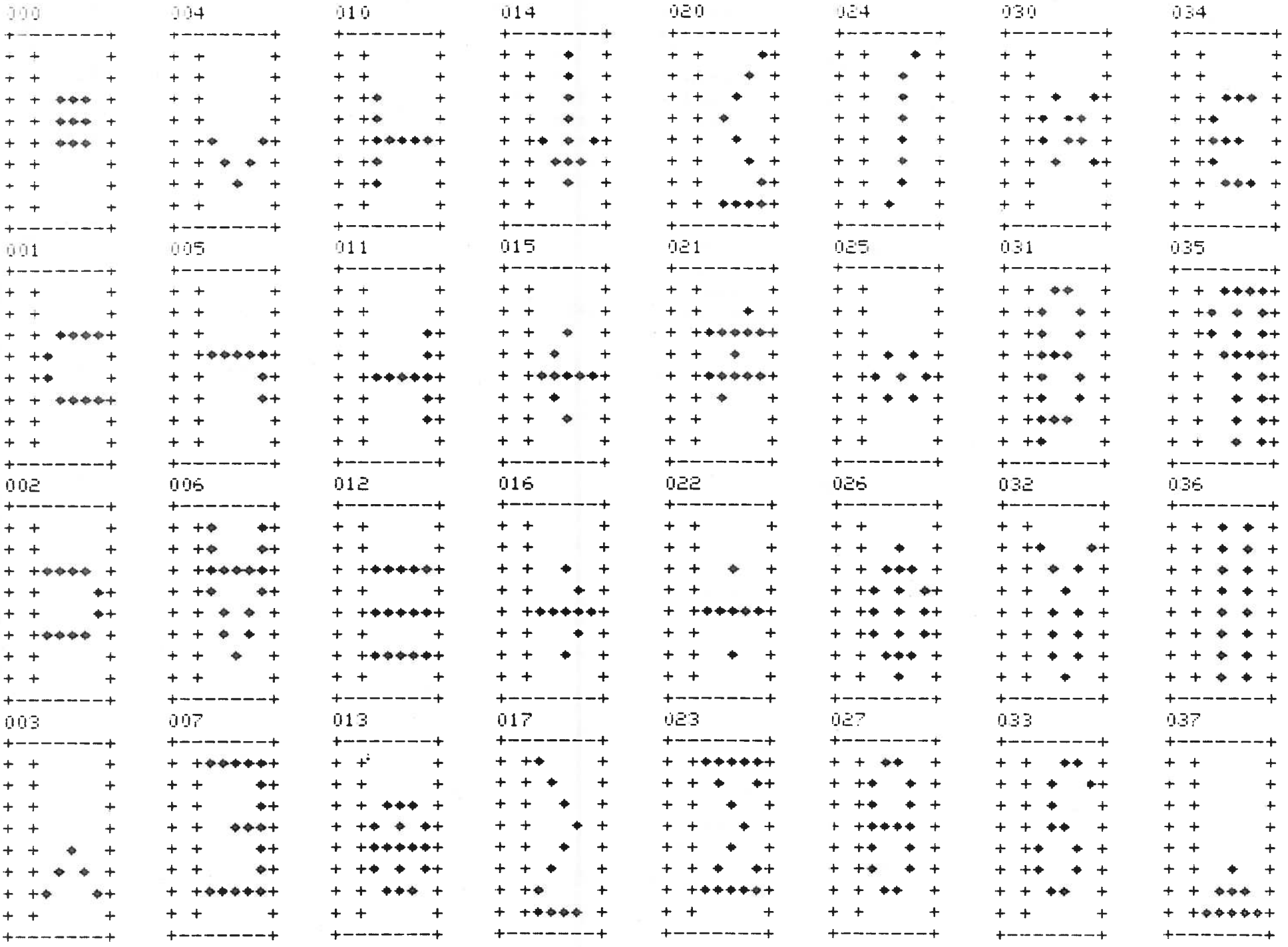
```
000 000:  176 346 040 310 345 315 125 **0 353 341 176 346 020 315 104 **0
000 020:  176 312 027 **0 032 356 200 022 311 041 000 sss 021 000 020 160
000 040:  043 033 172 263 302 037 **0 311 315 326 **3 320 066 040 043 160
000 060:  043 257 167 043 161 043 162 043 167 043 163 043 066 037 043 066
000 100:  001 311 043 043 043 043 043 043 043 043 043 311 053 053 053 053
000 120:  053 053 053 053 311 345 315 143 **0 343 315 106 **0 116 006 000
000 140:  341 011 311 043 043 176 053 053 315 154 **0 311 345 043 206 157
000 160:  046 000 051 051 051 124 135 051 104 115 051 051 011 031 021 000
000 200:  sss 031 321 353 315 107 **0 156 046 000 031 311 345 345 315 154
000 220:  **0 343 176 346 200 366 040 315 105 **0 106 341 167 043 005 362
000 240:  234 **0 341 311 043 043 176 053 053 315 214 **0 311 315 263 **0
000 260:  303 000 **0 345 043 043 066 000 043 176 043 043 066 000 341 365
000 300:  315 214 **0 361 075 362 277 **0 311 315 244 **0 345 315 106 **0
000 320:  066 000 341 303 000 **0 345 043 043 176 064 043 276 341 324 360
000 340:  **0 345 315 106 **0 257 167 315 110 **0 276 341 314 244 **0 311
000 360:  345 315 105 **1 341 345 315 103 **0 176 267 302 005 **1 315 116
001 000:  **0 257 167 341 311 127 053 053 116 014 315 121 **0 176 074 222
001 020:  107 053 160 341 372 263 **0 312 263 **0 345 325 176 346 200 366
001 040:  040 365 172 305 345 315 154 **0 343 257 315 154 **0 353 341 301
001 060:  361 305 315 123 **1 301 321 341 112 305 170 315 214 **0 301 004
001 100:  015 302 071 **1 311 176 346 014 376 014 300 315 102 **0 136 043
001 120:  126 353 351 365 305 345 325 107 176 160 022 015 043 023 362 145
001 140:  **1 053 053 033 033 076 177 241 302 130 **1 321 341 301 005 305
001 160:  001 130 000 362 171 **1 001 250 377 011 353 011 353 301 076 177
001 200:  240 343 174 341 302 123 **1 311 315 273 **2 267 303 233 **1 315
001 220:  273 **2 067 303 233 **1 267 303 233 **1 067 365 172 263 312 256
001 240:  **1 012 003 033 305 325 315 265 **1 321 301 303 234 **1 361 334
001 260:  124 **3 303 000 **0 315 305 **1 345 315 106 **0 176 064 043 276
001 300:  341 324 326 **0 311 345 043 043 106 315 110 **0 116 341 127 170
001 320:  345 305 325 315 154 **0 301 321 026 000 031 353 341 176 346 200
001 340:  250 022 311 170 267 370 315 103 **0 167 311 176 346 100 302 000
001 360:  **2 315 121 **2 370 176 366 100 167 305 325 315 263 **0 321 301
002 000:  345 172 315 056 **2 315 056 **2 365 257 315 154 **0 021 247 377
002 020:  031 361 001 130 000 315 104 **2 001 001 000 315 104 **2 001 250
002 040:  377 315 104 **2 001 377 377 315 104 **2 341 303 000 **0 315 105
002 060:  **0 126 130 353 343 345 353 315 121 **0 126 131 353 343 345 353
002 100:  315 121 **0 311 321 353 343 353 011 163 025 362 110 **2 011 167
002 120:  311 315 110 **0 176 336 002 370 315 110 **0 136 035 035 370 163
002 140:  053 053 064 053 167 053 053 064 053 311 315 167 **2 176 346 277
002 160:  167 315 263 **0 303 000 **0 176 346 100 310 345 076 001 043 065
002 200:  043 043 064 064 075 312 176 **2 341 311 315 273 **2 176 346 100
002 220:  310 172 267 300 203 310 315 105 **0 176 306 003 223 330 315 116
002 240:  **0 305 325 365 257 315 154 **0 361 037 306 247 117 006 377 011
002 260:  321 301 012 167 003 043 035 302 262 **2 311 305 021 000 000 012
002 300:  376 377 312 314 **2 003 023 172 263 302 277 **2 301 311 176 346
002 320:  200 250 345 315 104 **0 167 341 303 000 **0 176 356 200 167 345
002 340:  315 110 **0 106 315 110 **0 116 043 176 356 200 167 341 305 257
002 360:  315 154 **0 301 021 130 000 305 345 176 356 200 167 043 015 362
003 000:  371 **2 341 031 301 005 362 367 **2 311 345 315 337 **2 315 031
003 020:  **3 341 315 337 **2 315 031 **3 311 001 000 100 013 170 261 302
003 040:  034 **3 311 257 200 370 315 110 **0 176 270 330 257 201 370 315
003 060:  110 **0 176 271 330 315 116 **0 303 317 **1 076 040 315 305 **1
003 100:  345 315 106 **0 176 075 372 112 **3 167 341 303 000 **0 315 124
003 120:  **3 303 000 **0 345 315 106 **0 257 276 341 310 076 040 315 305
003 140:  **1 315 326 **0 311 315 167 **2 325 345 121 315 110 **0 116 014
003 160:  315 110 **0 136 034 341 315 326 **3 112 321 322 263 **3 345 325
003 200:  345 170 315 272 **3 127 171 315 272 **3 137 341 345 315 304 **3
003 220:  343 345 043 160 043 043 106 004 172 346 200 260 107 043 161 043
003 240:  043 116 014 173 346 200 261 117 341 315 304 **3 353 341 361 315
003 260:  123 **1 341 176 346 100 304 121 **2 311 043 226 057 346 200 043
003 300:  043 360 266 311 076 177 242 305 325 315 154 **0 321 076 177 243
003 320:  117 006 000 011 301 311 345 041 370 **3 343 170 267 370 376 050
003 340:  320 015 370 171 200 376 050 320 172 267 370 376 126 320 035 370
003 360:  173 202 376 126 320 343 341 311 267 311 000 000 000 000 000 000
```

CHARACTER SETS

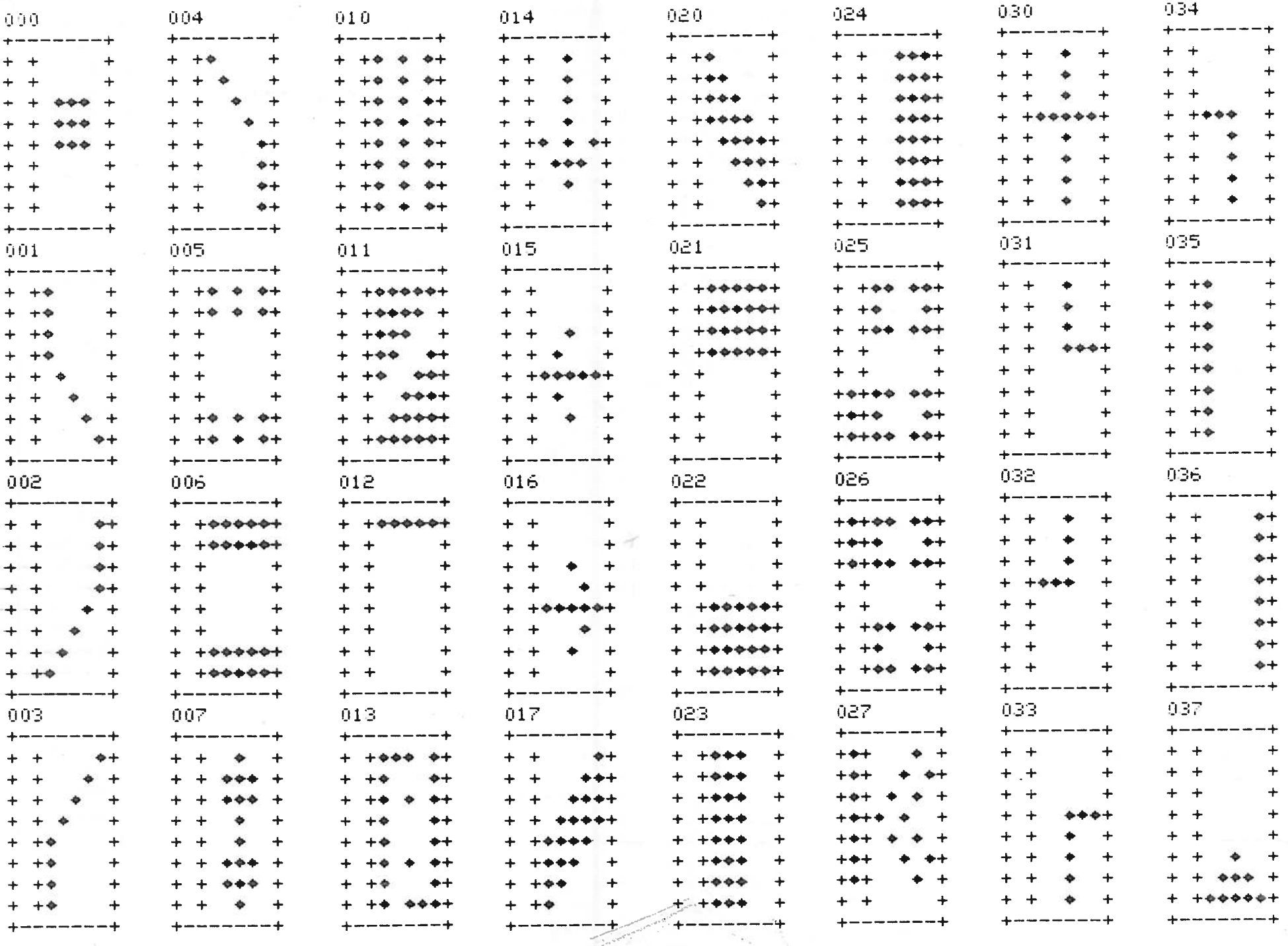
CHARACTER SET FONT SCHEMATIC EXPLANATION



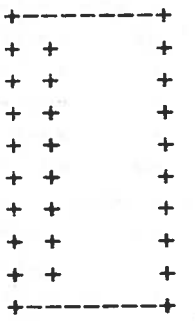
SCIENTIFIC CHARACTERS 000-037



GRAPHICS CHARACTERS 000-037



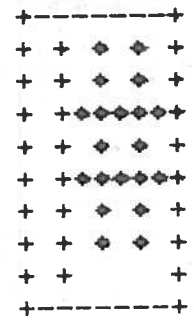
040



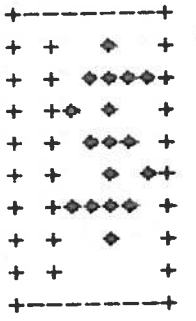
041

042

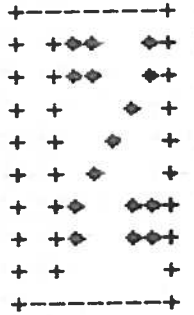
043



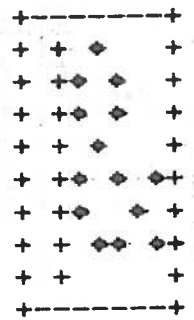
044



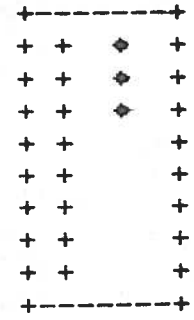
045



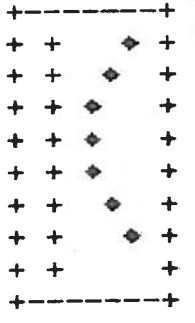
046



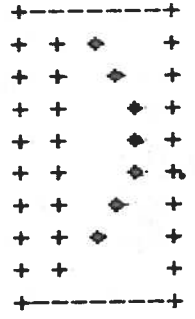
047



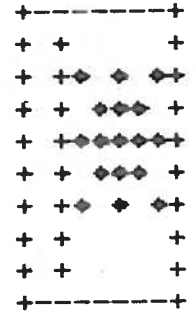
050



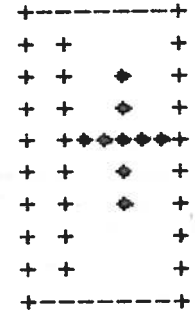
051



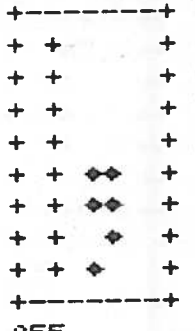
052



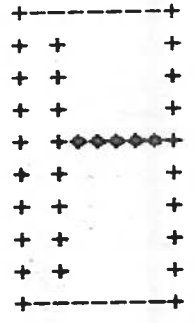
053



054

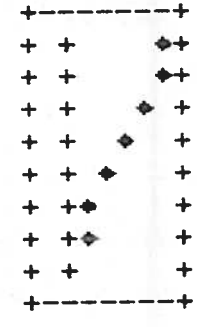


055

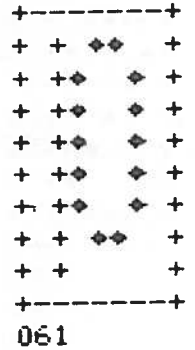


056

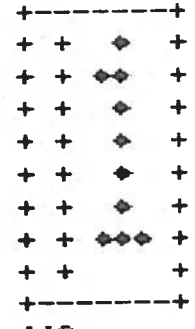
057



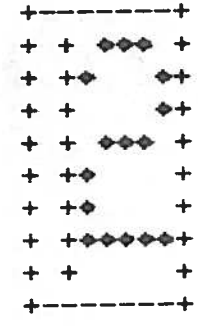
060



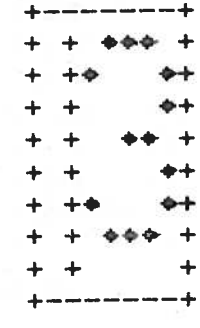
061



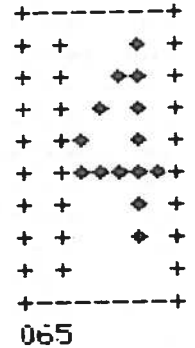
062



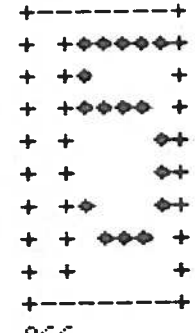
063



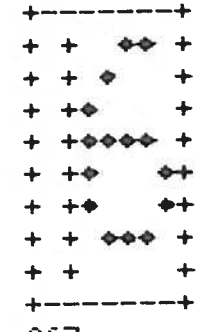
064



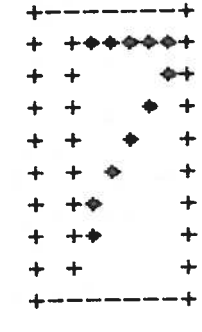
065



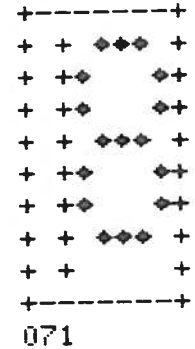
066



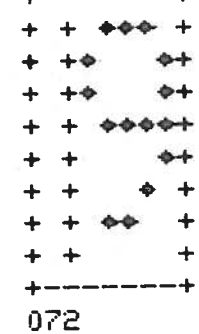
067



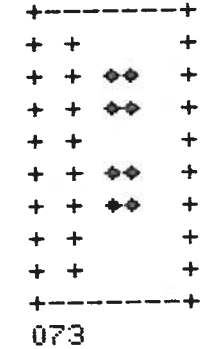
070



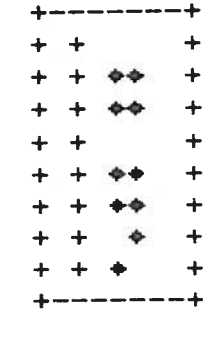
071



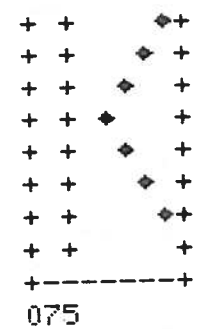
072



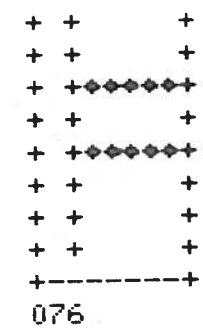
073



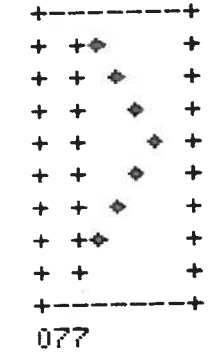
075



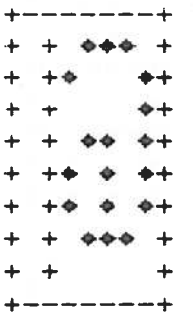
076



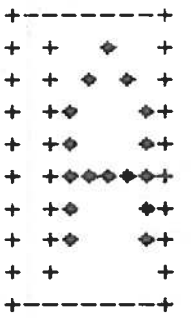
077



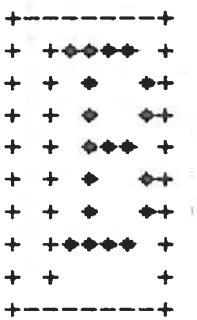
100



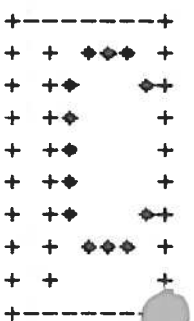
101



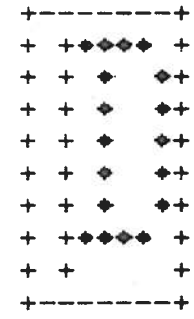
102



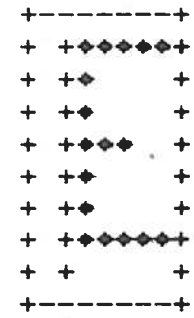
103



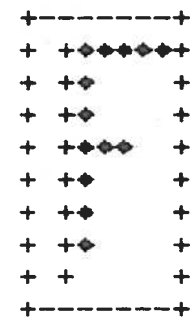
104



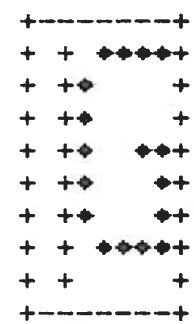
105



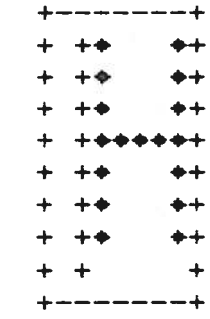
106



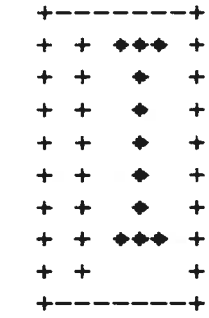
107



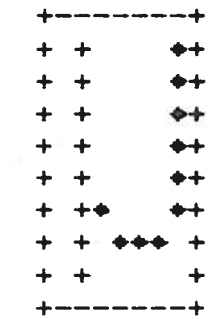
110



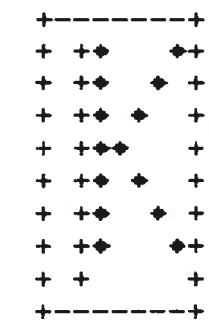
111



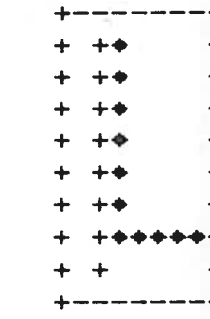
112



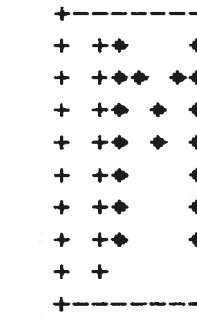
113



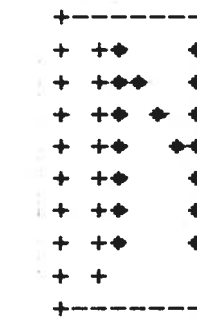
114



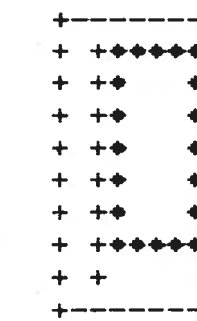
115



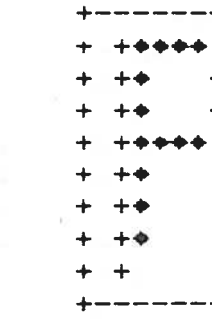
116



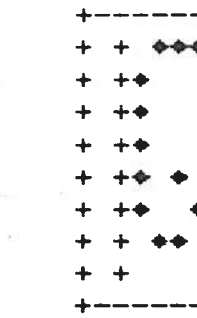
117



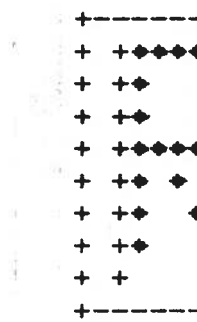
120



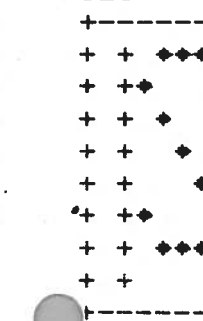
121



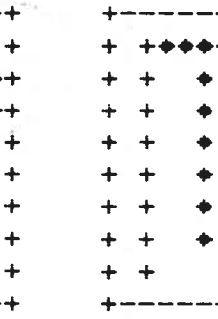
122



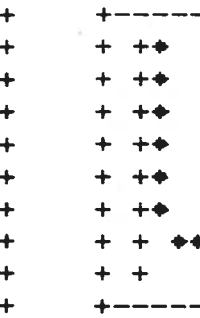
123



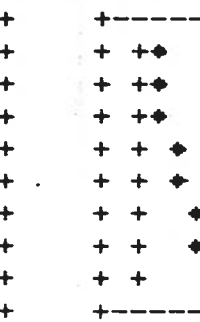
124



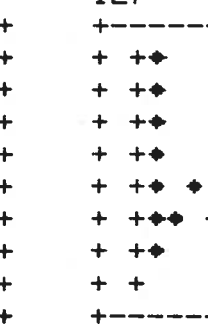
125



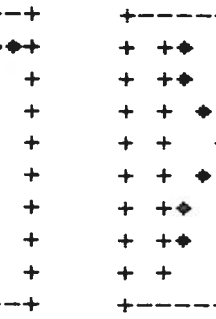
126



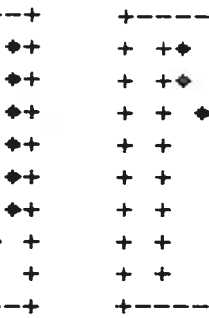
127



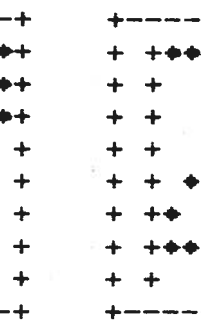
130



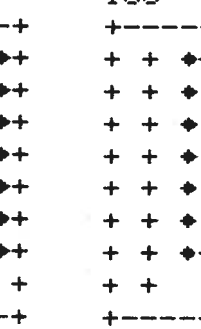
131



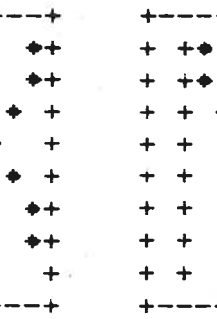
132



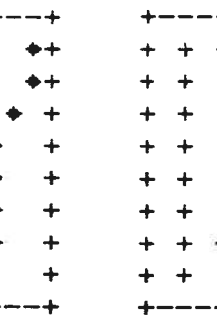
133



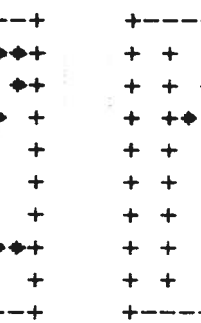
134



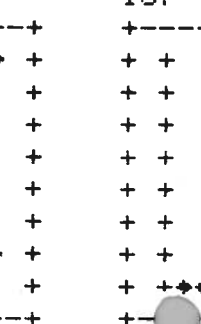
135



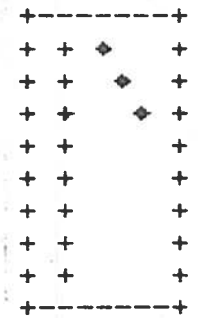
136



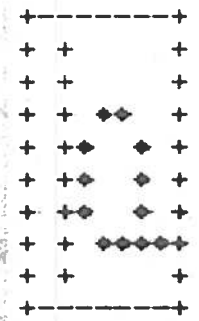
137



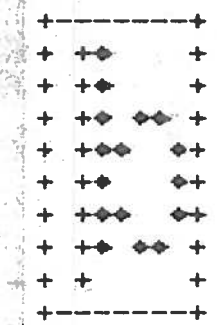
140



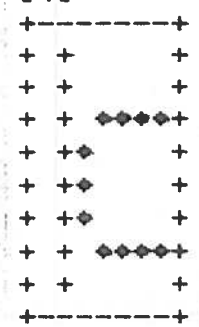
141



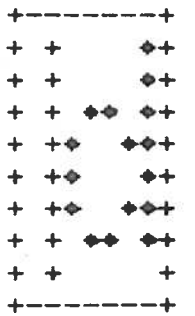
142



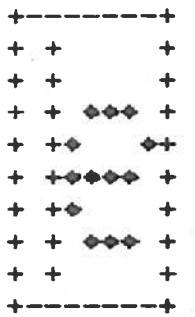
143



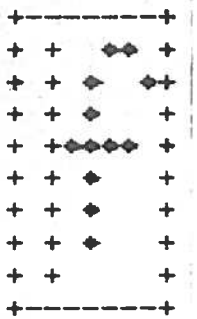
144



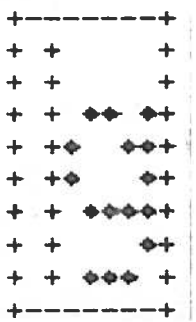
145



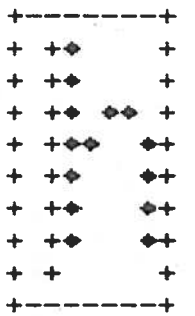
146



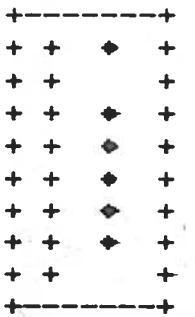
147



150

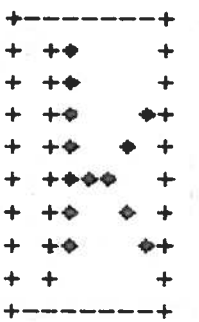


151

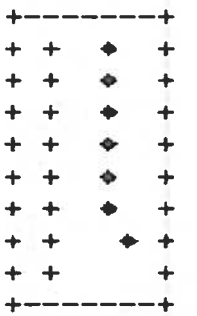


152

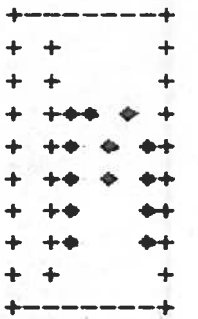
153



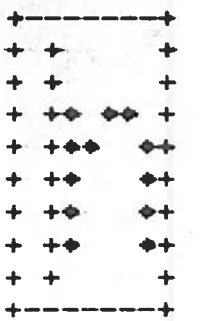
154



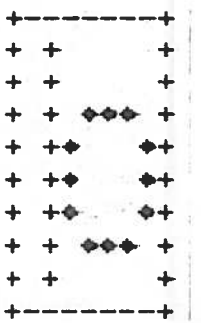
155



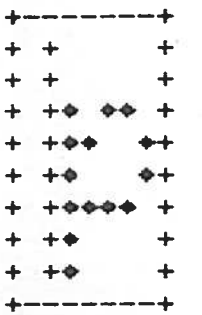
156



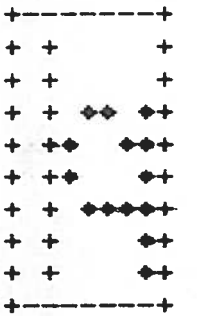
157



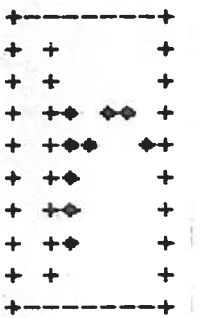
160



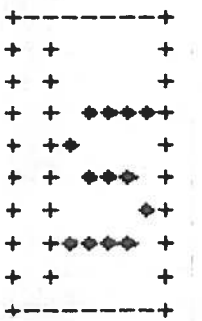
161



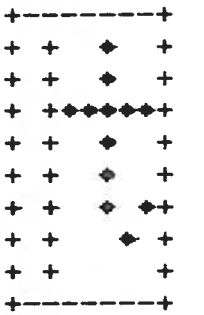
162



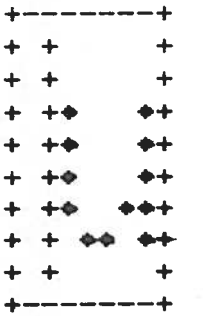
163



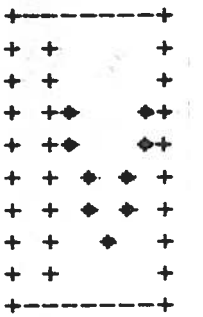
164



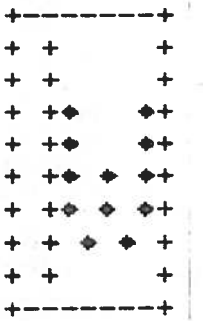
165



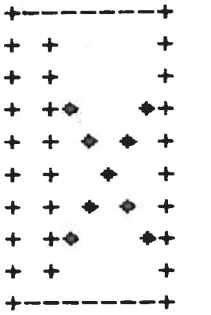
166



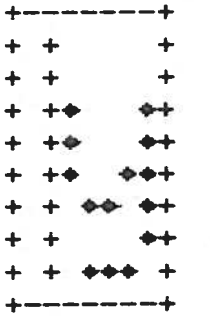
167



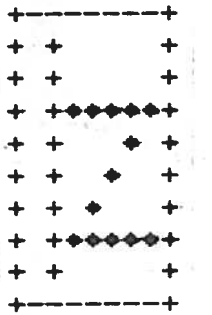
170



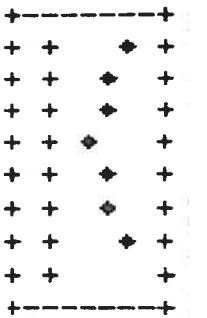
171



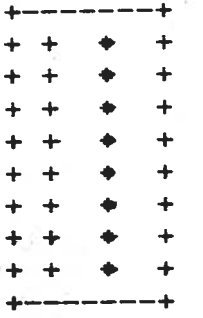
172



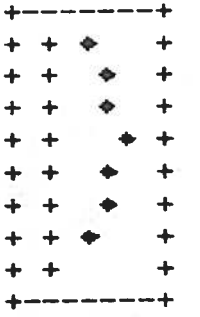
173



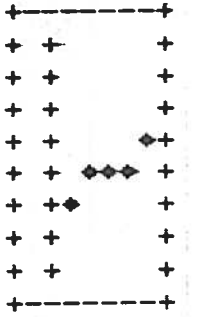
174



175



176



177

HEX LISTING: GRAPHICS CHARACTER GENERATOR EPROM

```
0000: 00 00 0E 0E 0E 00 00 00 10 10 10 10 08 04 02 01
0010: 01 01 01 01 02 04 08 10 01 02 04 08 10 10 10 10
0020: 10 08 04 02 01 01 01 01 15 15 00 00 00 00 15 15
0030: 1F 1F 00 00 00 00 1F 1F 04 0E 0E 04 04 0E 0E 04
0040: 15 15 15 15 15 15 15 15 1F 1E 1C 19 13 07 0F 1F
0050: 1F 00 00 00 00 00 00 00 1D 11 15 11 11 15 11 17
0060: 04 04 04 04 15 0E 04 00 00 00 04 08 1F 08 04 00
0070: 00 00 04 02 1F 02 04 00 01 03 07 0F 1E 1C 18 10
0080: 10 18 1C 1E 0F 07 03 01 1F 1F 1F 1F 00 00 00 00
0090: 00 00 00 00 1F 1F 1F 1F 1C 1C 1C 1C 1C 1C 1C 1C
00A0: 07 07 07 07 07 07 07 07 1B 11 1B 00 00 3B 31 3B
00B0: 3B 31 3B 00 00 1B 11 1B 22 25 2A 34 2A 25 22 00
00C0: 04 04 04 1F 04 04 04 04 04 04 04 07 00 00 00 00
00D0: 04 04 04 1C 00 00 00 00 00 00 00 07 04 04 04 04
00E0: 00 00 00 1C 04 04 04 04 10 10 10 10 10 10 10 10
00F0: 01 01 01 01 01 01 01 01 00 00 00 00 00 04 0E 1F
0100: 00 00 00 00 00 00 00 00 04 04 04 04 04 00 04 00
0110: 0A 0A 0A 00 00 00 00 00 0A 0A 1F 0A 1F 0A 0A 00
0120: 04 0F 14 0E 05 1E 04 00 19 19 02 04 08 13 13 00
0130: 08 14 14 08 15 12 0D 00 04 04 04 00 00 00 00 00
0140: 02 04 08 08 08 04 02 00 08 04 02 02 02 04 08 00
0150: 00 15 0E 1F 0E 15 00 00 00 04 04 1F 04 04 00 00
0160: 00 00 00 00 0C 0C 04 08 00 00 00 1F 00 00 00 00
0170: 00 00 00 00 00 0C 0C 00 01 01 02 04 08 10 10 00
0180: 0C 12 12 12 12 12 0C 00 04 0C 04 04 04 04 0E 00
0190: 0E 11 01 0E 10 10 1F 00 0E 11 01 06 01 11 0E 00
01A0: 02 06 0A 12 1F 02 02 00 1F 10 1E 01 01 11 0E 00
01B0: 06 08 10 1E 11 11 0E 00 1F 01 02 04 08 10 10 00
01C0: 0E 11 11 0E 11 11 0E 00 0E 11 11 0F 01 02 0C 00
01D0: 00 0C 0C 00 0C 0C 00 00 00 0C 0C 00 0C 0C 04 08
01E0: 01 02 04 08 04 02 01 00 00 00 1F 00 1F 00 00 00
01F0: 10 08 04 02 04 08 10 00 0C 12 02 04 04 00 04 00
0200: 0E 11 01 0D 15 15 0E 00 04 0A 11 11 1F 11 11 00
0210: 1E 09 09 0E 09 09 1E 00 0E 11 10 10 10 11 0E 00
0220: 1E 09 09 09 09 09 1E 00 1F 10 10 1C 10 10 1F 00
0230: 1F 10 10 1C 10 10 10 00 0F 10 10 13 11 11 0F 00
0240: 11 11 11 1F 11 11 11 00 0E 04 04 04 04 04 0E 00
0250: 01 01 01 01 01 11 0E 00 11 12 14 18 14 12 11 00
0260: 10 10 10 10 10 10 1F 00 11 1B 15 15 11 11 11 00
0270: 11 19 15 13 11 11 11 00 1F 11 11 11 11 11 1F 00
0280: 1E 11 11 1E 10 10 10 00 0E 11 11 11 15 12 0D 00
0290: 1E 11 11 1E 14 12 11 00 0E 11 08 04 02 11 0E 00
02A0: 1F 04 04 04 04 04 04 00 11 11 11 11 11 11 0E 00
02B0: 11 11 11 0A 0A 04 04 00 11 11 11 11 11 15 1B 11 00
02C0: 11 11 0A 04 0A 11 11 00 11 11 0A 04 04 04 04 00
02D0: 1F 01 02 04 08 10 1F 00 0E 08 08 08 08 08 0E 00
02E0: 10 10 08 04 02 01 01 00 0E 02 02 02 02 02 0E 00
02F0: 04 0E 15 04 04 04 04 00 00 00 00 00 00 00 00 1F
0300: 08 04 02 00 00 00 00 00 00 00 00 0C 12 12 12 0F 00
0310: 10 10 16 19 11 19 16 00 00 00 00 0F 10 10 10 0F 00
0320: 01 01 0D 13 11 13 0D 00 00 00 00 0E 11 1E 10 0E 00
0330: 06 09 08 1E 08 08 08 00 00 00 00 0D 13 11 0F 01 0E
0340: 10 10 16 19 11 11 11 00 04 00 04 04 04 04 04 00
0350: 01 00 01 01 01 01 09 06 10 10 11 12 1C 12 11 00
0360: 04 04 04 04 04 04 02 00 00 00 1A 15 15 11 11 00
0370: 00 00 16 19 11 11 11 00 00 00 0E 11 11 11 0E 00
0380: 00 00 16 19 11 1E 10 10 00 00 0D 13 11 0F 01 01
0390: 00 00 16 19 10 10 10 00 00 00 0F 10 0E 01 1E 00
03A0: 04 04 1F 04 04 05 02 00 00 00 11 11 11 13 0D 00
03B0: 00 00 11 11 0A 0A 04 00 00 00 11 11 15 15 0A 00
03C0: 00 00 11 0A 04 0A 11 00 00 00 11 11 13 0D 01 0E
03D0: 00 00 1F 02 04 08 1F 00 02 04 04 08 04 04 02 00
03E0: 04 04 04 04 04 04 04 04 08 04 04 02 04 04 08 00
03F0: 00 00 00 01 0E 10 00 00 20 20 20 20 20 24 2E 3F
```

HEX LISTING: SCIENTIFIC CHARACTER GENERATOR EPROM

```
0000: 00 00 0E 0E 0E 00 00 00 00 00 0F 10 10 0F 00 00
0010: 00 00 1E 01 01 1E 00 00 00 00 00 00 04 0A 11 00
0020: 00 00 00 00 11 0A 04 00 00 00 00 1F 01 01 00 00
0030: 11 11 1F 11 0A 0A 04 00 1F 01 01 07 01 01 1F 00
0040: 00 00 10 10 1F 10 10 00 00 00 01 01 1F 01 01 00
0050: 00 00 1F 00 1F 00 1F 00 00 00 0E 15 1F 15 0E 00
0060: 04 04 04 04 15 0E 04 00 00 00 04 08 1F 08 04 00
0070: 00 00 04 02 1F 02 04 00 10 08 04 02 04 08 10 1E
0080: 01 02 04 08 04 02 01 0F 00 02 1F 04 1F 08 00 00
0090: 00 00 04 00 1F 00 04 00 1F 09 04 02 04 09 1F 00
00A0: 02 04 04 04 04 04 04 08 00 00 00 0A 15 0A 00 00
00B0: 00 04 0E 15 15 15 0E 04 0C 12 12 1E 12 12 0C 00
00C0: 00 00 09 16 16 09 00 00 0C 12 12 1C 12 12 1C 10
00D0: 00 11 0A 04 0A 0A 0A 04 06 09 08 0C 12 12 0C 00
00E0: 00 00 0E 10 1C 10 0E 00 0F 15 15 0F 05 05 05 05
00F0: 0A 0A 0A 0A 0A 0A 0A 0A 00 00 00 00 00 04 0E 1F
0100: 00 00 00 00 00 00 00 00 04 04 04 04 04 00 04 00
0110: 0A 0A 0A 00 00 00 00 00 0A 0A 1F 0A 1F 0A 0A 00
0120: 04 0F 14 0E 05 1E 04 00 19 19 02 04 08 13 13 00
0130: 08 14 14 08 15 12 0D 00 04 04 04 00 00 00 00 00
0140: 02 04 08 08 08 04 02 00 08 04 02 02 02 04 08 00
0150: 00 15 0E 1F 0E 15 00 00 00 04 04 1F 04 04 00 00
0160: 00 00 00 00 0C 0C 04 08 00 00 00 1F 00 00 00 00
0170: 00 00 00 00 00 0C 0C 00 01 01 02 04 08 10 10 00
0180: 0C 12 12 12 12 12 0C 00 04 0C 04 04 04 04 0E 00
0190: 0E 11 01 0E 10 10 1F 00 0E 11 01 06 01 11 0E 00
01A0: 02 06 0A 12 1F 02 02 00 1F 10 1E 01 01 11 0E 00
01B0: 06 08 10 1E 11 11 0E 00 1F 01 02 04 08 10 10 00
01C0: 0E 11 11 0E 11 11 0E 00 0E 11 11 0F 01 02 0C 00
01D0: 00 0C 0C 00 0C 0C 00 00 00 0C 0C 00 0C 0C 04 08
01E0: 01 02 04 08 04 02 01 00 00 00 1F 00 1F 00 00 00
01F0: 10 08 04 02 04 08 10 00 0C 12 02 04 04 00 04 00
0200: 0E 11 01 0D 15 15 0E 00 04 0A 11 11 1F 11 11 00
0210: 1E 09 09 0E 09 09 1E 00 0E 11 10 10 10 11 0E 00
0220: 1E 09 09 09 09 09 1E 00 1F 10 10 1C 10 10 1F 00
0230: 1F 10 10 1C 10 10 10 00 0F 10 10 13 11 11 0F 00
0240: 11 11 11 1F 11 11 11 00 0E 04 04 04 04 04 0E 00
0250: 01 01 01 01 01 11 0E 00 11 12 14 18 14 12 11 00
0260: 10 10 10 10 10 10 1F 00 11 1B 15 15 11 11 11 00
0270: 11 19 15 13 11 11 11 00 1F 11 11 11 11 11 1F 00
0280: 1E 11 11 1E 10 10 10 00 0E 11 11 11 15 12 0D 00
0290: 1E 11 11 1E 14 12 11 00 0E 11 08 04 02 11 0E 00
02A0: 1F 04 04 04 04 04 04 00 11 11 11 11 11 11 0E 00
02B0: 11 11 11 0A 0A 04 04 00 11 11 11 11 15 1B 11 00
02C0: 11 11 0A 04 0A 11 11 00 11 11 0A 04 04 04 04 00
02D0: 1F 01 02 04 08 10 1F 00 0E 08 08 08 08 08 0E 00
02E0: 10 10 08 04 02 01 01 00 0E 02 02 02 02 02 0E 00
02F0: 04 0E 15 04 04 04 04 00 00 00 00 00 00 00 00 1F
0300: 08 04 02 00 00 00 00 00 00 00 0C 12 12 12 0F 00
0310: 10 10 16 19 11 19 16 00 00 00 0F 10 10 10 0F 00
0320: 01 01 0D 13 11 13 0D 00 00 00 0E 11 1E 10 0E 00
0330: 06 09 08 1E 08 08 08 00 00 00 0D 13 11 0F 01 0E
0340: 10 10 16 19 11 11 11 00 04 00 04 04 04 04 04 00
0350: 01 00 01 01 01 01 09 06 10 10 11 12 1C 12 11 00
0360: 04 04 04 04 04 04 02 00 00 00 1A 15 15 11 11 00
0370: 00 00 16 19 11 11 11 00 00 00 0E 11 11 11 0E 00
0380: 00 00 16 19 11 1E 10 10 00 00 0D 13 11 0F 01 01
0390: 00 00 16 19 10 10 10 00 00 00 0F 10 0E 01 1E 00
03A0: 04 04 1F 04 04 05 02 00 00 00 11 11 11 13 0D 00
03B0: 00 00 11 11 0A 0A 04 00 00 00 11 11 15 15 0A 00
03C0: 00 00 11 0A 04 0A 11 00 00 00 11 11 13 0D 01 0E
03D0: 00 00 1F 02 04 08 1F 00 02 04 04 08 04 04 02 00
03E0: 04 04 04 04 04 04 04 04 08 04 04 02 04 04 08 00
03F0: 00 00 00 01 0E 10 00 00 20 20 20 20 20 24 2E 3F
```

WARRANTY

The SCREENSPLITTER Video Display System is warranteed for 90 days. During this period any part which fails because of defects in material or workmanship will be repaired or replaced at no charge, if in the opinion of Micro Diversions, Inc., the module has not been subjected to electrical or mechanical abuse and the cautions as stated in the Assembly Guide have been followed. The defective module must be returned postpaid to Micro Diversions, Inc. SCREENSPLITTERs purchased through a dealer should be returned to the seller for service.